How Numerical Precision Affects Arithmetical Reasoning Capabilities of LLMs

Anonymous ACL submission

Abstract

Despite the remarkable success of Transformer-based large language models (LLMs) across various domains, understanding and enhancing their mathematical capabilities remains a significant challenge. In this paper, we conduct a rigorous theoretical analysis of LLMs' mathematical abilities, with a specific focus on their arithmetic performances. We identify numerical precision as a key factor that influences their effectiveness in arithmetical tasks. Our results show that Transformers operating with low numerical precision fail to address arithmetic tasks, such as iterated addition and integer multiplication, unless the model size grows super-polynomially with respect to the input length. In contrast, Transformers with standard numerical precision can efficiently handle these tasks with significantly smaller model sizes. We further support our theoretical findings through empirical experiments that explore the impact of varying numerical precision on arithmetic tasks, providing valuable insights for improving the mathematical reasoning capabilities of LLMs.

1 Introduction

Transformer-based LLMs, such as GPT (OpenAI, 2023), Claude (Anthropic, 2024), and LLAMA (Dubey et al., 2024), have achieved impressive performance across a broad range of natural language tasks (Basyal and Sanghvi, 2023; Shao et al., 2023; Zhu et al., 2024). Despite the great success, significant challenges remain when applying LLMs to mathematical problem-solving. Unlike many typical NLP tasks, which often depend on pattern recognition and statistical correlations (Blei et al., 2003), mathematical reasoning requires rigorous logical deduction in a specific order (Bubeck et al., 2023; Frieder et al., 2024). To address these challenges, various strategies have been proposed, including carefully designed prompting strategies (Wei et al., 2022b; Yamauchi et al., 2023; Imani et al., 2023) and inference-based searching method (Kang et al., 2024; Wu et al., 2024a; Snell et al., 2024; Brown et al., 2024). However, a comprehensive understanding of the intrinsic limitations that restrict the mathematical reasoning capabilities of LLMs remains elusive.

In principle, mathematical reasoning, built on basic arithmetical operations, requires accurate computation of intermediate results throughout the reasoning process (Bubeck et al., 2023; Lee et al., 2024). There exist works (Feng et al., 2023; Yang et al., 2024) exploring the arithmetic capabilities of LLMs with Chain of Thought (CoT) prompting (Wei et al., 2022b). However, these investigations often deviate from the tokenization strategies employed by modern LLMs (OpenAI, 2023; Dubey et al., 2024), where numbers are typically segmented into tokens of at most three digits. Under the assumption of Feng et al. (2023) and Yang et al. (2024), each distinct number occupies a unique position in the vocabulary, leading to an essential mismatch with practical implementations. Moreover, recent studies have demonstrated that LLMs operating with reduced numerical precision (e.g., `int4`) exhibit a significant decline in performance on mathematical tasks (Jin et al., 2024; Marchisio et al., 2024).

In this paper, we provide a rigorous theoretical investigation of the arithmetical abilities of LLMs under the autoregressive paradigm. Specifically, we adopt the tokenization approach used in modern LLMs, where numbers are processed and generated token by token, with each token representing only a small number of digits. Under these assumptions, we identify **numerical precision** as a key factor influencing their performance in arithmetical tasks. Our analysis focuses on three elementary arithmetic tasks: integer addition, iterated addition, and integer multiplication, which serve as elementary building blocks in solving complex real-world math problems.

Arithmetic Tasks	**Standard Precision**	**Low Precision**
Integer Addition $\mathrm{ADD}_p(n)$	Constant	$O(n^2)$
Iterated Addition $\mathrm{IterADD}_p(n,k)$	Constant	Super-polynomial
Integer Multiplication $\mathrm{Mul}_p(n,l)$	$O(n^2)$	Super-polynomial

Table 1: The model size *w.r.t.* the input size required for various arithmetic tasks on bounded-depth Transformers, under both standard and low numerical precision. Blue denotes the acceptable model size, and red represents the unaffordable model size.

To elucidate the role of numerical precision, we first examine the expressiveness of Transformers operating under low precision, such as `int8` and `int4`. We establish foundational impossibility results for low-precision Transformers, demonstrating that such models require super-polynomial size with respect to input length to solve iterated addition and integer multiplication (Theorems 4.2 and 4.3). Our proofs, grounded in complexity theory (Razborov, 1987; Arora and Barak, 2009), show that this limitation arises from the inability of individual neurons to store intermediate results during arithmetic computations. As a result, a significantly larger number of neurons is required to distribute the computation and avoid overflow.

We further demonstrate that increasing numerical precision is essential to addressing this limitation. Specifically, as numerical precision improves, the model size required to solve arithmetic tasks decreases significantly. In particular, we prove that a bounded-depth Transformer operating with standard precision (e.g., `float32`) can efficiently and reliably solve all three tasks under consideration. For both integer and iterated addition, the required model size remains constant and independent of the input length (Theorems 5.1 and 5.2), while for integer multiplication, the model size scales quadratically w.r.t the input length (Theorem 5.3). These results highlight that standard numerical precision is sufficient for LLMs to effectively perform arithmetic tasks. Our findings emphasize the practical importance of numerical precision in mathematical reasoning. While low-precision models may offer computational advantages, ensuring sufficient numerical precision is critical for tasks involving complex arithmetic. A summary of our main results is provided in Table 1.

In addition to theoretical analysis, we conduct extensive experiments to validate our conclusions. First, we evaluate the performance of Transformers trained from scratch on the aforementioned arithmetic tasks, systematically examining how problem size and numerical precision impact their capabilities. Furthermore, we also conduct experiments on LLAMA-3.1-8B Instruct (Dubey et al., 2024) to evaluate the performance of these arithmetic tasks under different numerical precision. Our empirical results demonstrate that both low-precision and standard-precision Transformers perform adequately on the integer addition task. However, as task complexity increases—particularly in iterated addition and integer multiplication—the decrease in precision in Transformers results in significant performance degradation. These findings align with our theoretical predictions and offer practical guidance for enhancing LLM performance in mathematical reasoning tasks.

2 Preliminary

An autoregressive Transformer, or decoder-only Transformer (Radford et al., 2019; Dai et al., 2019), is a neural network designed to model sequence-to-sequence mappings. For an input sequence $\boldsymbol{s}$ of length n, each input token s_i (for $i \in [n]$) is transformed into a d-dimensional vector $\boldsymbol{x}_i^{(0)} = \mathrm{Embed}(s_i) + \boldsymbol{p}_i \in \mathbb{R}^d$, where $\mathrm{Embed}(\cdot)$ represents the token embedding function, and $\boldsymbol{p}_i$ denotes learnable positional embeddings. The model then consists of L Transformer blocks, each following the form:

$$\boldsymbol{h}_i^{(l)} = \boldsymbol{x}_i^{(l-1)} + \mathrm{Attn}^{(l)}\left(\boldsymbol{x}_i^{(l-1)}; \{\boldsymbol{x}_j^{(l-1)} : j \le i\}\right),$$
$$\boldsymbol{x}_i^{(l)} = \boldsymbol{h}_i^{(l)} + \mathrm{FFN}^{(l)}(\boldsymbol{h}_i^{(l)}),$$

where $l \in [L]$. Here, $\mathrm{Attn}^{(l)}$ and $\mathrm{FFN}^{(l)}$ denote the multi-head self-attention layer and the feed-forward network of the l-th Transformer block:

$$\begin{aligned}
&\mathrm{Attn}^{(l)}(\boldsymbol{x}, \mathcal{S}) = \sum_{h=1}^{H} \left(\boldsymbol{W}_{\mathrm{O}}^{(l,h)}\right)^{\top} \cdot \mathrm{H}^{(l,h)}(\boldsymbol{x}, \mathcal{S}),\\
&\mathrm{H}^{(l,h)}(\boldsymbol{x}, \mathcal{S}) =\\
&\mathrm{softmax}_{\boldsymbol{z}\in\mathcal{S}}\left((\boldsymbol{W}_{\mathrm{K}}^{(l,h)}\boldsymbol{z})^{\top}(\boldsymbol{W}_{\mathrm{Q}}^{(l,h)}\boldsymbol{x})\right)\boldsymbol{W}_{\mathrm{V}}^{(l,h)}\boldsymbol{z},\\
&\mathrm{FFN}^{(l)}(\boldsymbol{x}) = \boldsymbol{W}_2^{(l)}\sigma(\boldsymbol{W}_1^{(l)}\boldsymbol{x}),
\end{aligned}$$

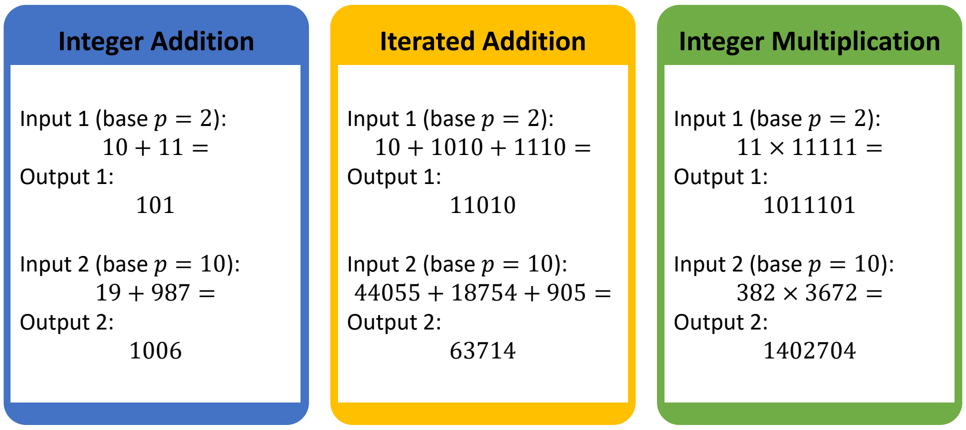

Figure 1: Examples for three elementary arithmetic tasks we consider in this paper: integer addition, iterated addition, and integer multiplication.

where $\boldsymbol{W}_{\mathrm{Q}}^{(l,h)}, \boldsymbol{W}_{\mathrm{K}}^{(l,h)}, \boldsymbol{W}_{\mathrm{V}}^{(l,h)}, \boldsymbol{W}_{\mathrm{O}}^{(l,h)} \in \mathbb{R}^{\lceil \frac{d}{H} \rceil \times d}$ are the query, key, value, and output matrices of the h-th head in the l-th layer. The weight matrices in the feed-forward network are denoted as $\boldsymbol{W}_1^{(l)}, \boldsymbol{W}_2^{(l)} \in \mathbb{R}^{d \times d}$. The activation function σ is chosen to be GeLU (Hendrycks and Gimpel, 2016), following the work of (Radford et al., 2019).

The computed embedding $\boldsymbol{x}_n^{(M)}$ is then used to predict the next token s_{n+1}, which is concatenated to the input to continue the sequence generation process. This process terminates when an `<EOS>` token is generated. Further discussions on related work are listed in Appendix A.

3 Problem Setup

This paper explores the arithmetic reasoning capabilities of LLMs by focusing on three elementary arithmetic tasks: integer addition, iterated addition, and integer multiplication under the autoregressive paradigm. Below, we define the integer representations used throughout the study and provide formal descriptions for each task.

Integer Representation and Tokenization. We consider all integers to be non-negative and represented in base-p notation, where $p \geq 2$ is a fixed base. Specifically, an integer with n digits is expressed as $(x_{n-1} \cdots x_0)_p$. To tokenize this sequence, we employ a tokenizer, denoted by $\boldsymbol{T}_c$, that partitions $\mathbf{x}$ into tokens, each containing at most c contiguous digits. Formally, let the sequence $\boldsymbol{t} = [t_{k-1}, \ldots, t_0] = \boldsymbol{T}_c([x_{n-1}, \ldots, x_0])$, where $k = \lceil \frac{n}{c} \rceil$ we have

$$t_i = \begin{cases} [x_{ic}, x_{ic+1}, \cdots, x_{ic+c-1}], & i < k-1; \\ [x_{ic}, x_{ic+1}, \cdots, x_{n-1}], & i = k-1. \end{cases} \tag{1}$$

During sequence generation, the Transformer model outputs the target tokens sequentially, strictly following the tokenization scheme of tokenizer $\boldsymbol{T}_c$. Unlike prior works (Feng et al., 2023; Yang et al., 2024), which represent entire integers as single tokens, this approach aligns with prevalent tokenization strategies employed by modern LLMs (OpenAI, 2023; Dubey et al., 2024) and enables Transformers to process and generate numbers token by token. Further discussion and illustrative examples of the tokenization scheme are provided in Appendix B.4.

Integer Addition. Let $\boldsymbol{a} = (a_{n_1-1} \cdots a_0)_p$ and $\boldsymbol{b} = (b_{n_2-1} \cdots b_0)_p$ denote two integers represented in base-p. Their sum is expressed as $\boldsymbol{s} = (s_n \cdots s_0)_p = \boldsymbol{a} + \boldsymbol{b}$. Let $\boldsymbol{T}_c$ represent the tokenizer. The input sequence is constructed by concatenating the tokenized representations of $\boldsymbol{a}$ and $\boldsymbol{b}$, i.e., $\boldsymbol{T}_c(\boldsymbol{a})$ and $\boldsymbol{T}_c(\boldsymbol{b})$, with the addition operator token '$+$' placed between them, and the equality operator token '$=$' appended at the end. The task is to generate the tokenized representation of the result, $\boldsymbol{T}_c(\boldsymbol{s})$, sequentially, one token at a time.

Iterated Addition. Now consider k integers in base-p: $\boldsymbol{a}_1 = (a_{1,n_1-1} \cdots a_{1,0})_p, \ \ldots, \ \boldsymbol{a}_k = (a_{k,n_k-1} \cdots a_{k,0})_p$, where $n = \max\{n_1, \ldots, n_k\}$. Their sum is denoted as $\boldsymbol{s} = (s_{n-1} \cdots s_0)_p = \sum_{i \in [k]} \boldsymbol{a}_i$, where $n = \max_{i \in [k]}\{n_k\} + \lceil \log k \rceil$. Let $\boldsymbol{T}_c$ denote the tokenizer. The input sequence is formed by concatenating the tokenized representations of these integers, separated by the addition operator token '$+$', followed by the equality operator token '$=$' appended at the end. The objective is for the Transformer to generate the tokenized representation of the sum, $\boldsymbol{T}_c(\boldsymbol{s})$, sequentially, one token at a time.

Integer Multiplication. The integer multiplication task involves computing the product of two integers, truncated to a predefined length l. Let $\boldsymbol{a} = (a_{n_1-1} \cdots a_0)_p$ and $\boldsymbol{b} = (b_{n_2-1} \cdots b_0)_p$ represent two integers in base-p, and let $n = \max\{n_1, n_2\}$.

Their product is given by $\boldsymbol{s} = (s_{2n-1} \cdots s_0)_p = \boldsymbol{a} \times \boldsymbol{b}$. Let $\boldsymbol{T}_c$ denote the tokenizer. The input sequence is constructed by concatenating the tokenized representations of $\boldsymbol{a}$ and $\boldsymbol{b}$, separated by the multiplication operator token '$\times$', with the equality operator token '$=$' appended at the end. The objective is to generate the tokenized representation of the product's least significant l digits, $\boldsymbol{T}_c([s_{l-1}, s_{l-2}, \ldots, s_0])$, where $l \leq 2n$.

Remark 3.1. We consider a generalized case of integer multiplication where overflow may occur if the result exceeds the given digit length. Standard integer multiplication is a special case of this framework when $l = n_1 + n_2$.

Figure 1 presents examples of these tasks. Integer addition is the simplest of these tasks and can be viewed as a specific instance of iterated addition. Furthermore, integer multiplication inherently involves the summation of several intermediate products. Consequently, we present these tasks in increasing order of complexity. In the subsequent sections, we use the notations $\mathrm{ADD}_p(n)$ to denote addition with at most n digits in base-p arithmetic, $\mathrm{IterADD}_p(n, k)$ for the iterated addition of k integers with at most n digits each in base-p, and $\mathrm{MUL}_p(n, l)$ for the multiplication of two integers with at most n digits in base-p, truncated to l digits.

4 Low-Precision Transformers Struggle with Basic Arithmetic Tasks

Recent studies (Marchisio et al., 2024; Jin et al., 2024) have shown that LLMs operating under low-precision constraints encounter significant challenges in performing basic mathematical tasks. In this section, we examine the expressive limitations of Transformers under such constraints and seek to explain the sharp decline in their arithmetical capabilities. Specifically, we demonstrate that Transformers restricted to low-precision arithmetic exhibit substantial difficulty in solving even elementary arithmetic problems.

To formalize these limitations, we build on the framework introduced by Li et al. (2024) and utilize the setting of a **constant-precision Transformer** (See formal definition in Appendix B.2). In this setting, the internal states of the model's neurons are constrained to represent real numbers using only c bits, where c is a small constant independent of the input sequence length. These numbers may be represented by floating point in IEEE 754 formats (Kahan, 1996) or fixed point formats. This configuration mirrors many practical deployment scenarios, in which LLMs often employ reduced-precision formats such as `float8`, `int8`, or even `int4`, particularly during inference (Han et al., 2015). Given that these models typically process input sequences comprising thousands of tokens, it is reasonable and realistic to assume that the numerical precision remains fixed at a small constant, independent of sequence length. Under the constant-precision setting, we examine the expressiveness of the Transformer model in elementary arithmetic problems.

Theorem 4.1. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. There exist constant-precision Transformers with constant depth (independent of n) and hidden dimension $d = O(n^2)$ that can solve the $\mathrm{ADD}_p(n)$ task.*

Theorem 4.1 suggests that the bounded-depth Transformers with reasonable hidden dimensions are capable of solving the integer addition task. However, as we will show in subsequent theorems, constant-precision Transformers exhibit pronounced limitations when considering more complex arithmetic problems. For the page limitation, we give the detailed proof of Theorem 4.1 in Appendix D.1.

Theorem 4.2. *Fix integers $p \geq 2$ and $c, L \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any polynomial f, there exist problem scales n and k such that no constant-precision autoregressive Transformer with L layers and hidden dimension $d < f(n, k)$ can correctly solve the $\mathrm{IterADD}_p(n, k)$ task.*

Theorem 4.3. *Fix integers $p \geq 2$ and $c, L \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any polynomial f, there exist problem scales n and l such that no constant-precision autoregressive Transformer with L layers and hidden dimension $d < f(n, l)$ can correctly solve the $\mathrm{MUL}_p(n, l)$ task.*

The detailed proof of Theorems 4.2 and 4.3 are presented in Appendices D.2 and D.3.

What accounts for this limitation? As presented in Appendix D, our proof is grounded in circuit complexity theory. By modeling the constant-precision Transformer as a computational circuit, we rigorously analyze its expressive limitations through the lens of circuit complexity (Merrill et al., 2022; Merrill and Sabharwal, 2023; Feng et al., 2023; Li et al., 2024). Specifically, Li et al. (2024) proves that the expressiveness of constant-precision Transformers with polynomial size and bounded depth is upper-bounded by the computation complexity class AC^0. In contrast, we demonstrate that the complexity of tasks such as $\mathrm{IterADD}$ and MUL exceeds that of AC^0, using reductions from `Majority`, a well-established problem that has been provably unsolvable by the circuits in AC^0 (Razborov, 1987; Smolensky, 1987). Consequently, these tasks are inherently hard for low-precision Transformers.

Practical Implications. While low-precision Transformers can effectively handle some of the simplest arithmetic tasks, such as basic integer addition, their capacity is severely limited when addressing more complex tasks. As demonstrated, low numerical precision, such as `int4` and `float8`, imposes fundamental constraints, preventing these models from solving problems that would require Transformers with super-polynomial size.

5 Standard-Precision Transformers Are Sufficient for Arithmetic Tasks

In Section 4, we demonstrated that low-precision Transformers struggle with arithmetic tasks due to their expressive limitations. In this section, we will show that increasing numerical precision is essential to overcoming this limitation. In particular, we focus on **standard-precision** Transformers and show that such models can overcome these limitations and solve arithmetic problems efficiently.

To formalize the notion of standard precision (e.g., `float32`), we follow Feng et al. (2023) and adopt the setting of a **logarithmic-precision Transformer** (See formal definition in Appendix B). In this setting, the Transformer's internal neurons can represent real numbers with up to $O(\log n)$ bits, where n denotes the maximum input sequence length. Given that modern LLMs often limit their context length to hundreds of thousands of tokens (OpenAI, 2023; Touvron et al., 2023; Anthropic, 2024), it is natural to treat 32 as the logarithmic scale corresponding to $100,000$. Hence, the logarithmic-precision setting reflects practical deployment scenarios.

We first establish that, under logarithmic precision, a Transformer with constant depth and dimension can solve both the integer addition and iterated addition tasks for arbitrarily large input lengths, as shown in Theorems 5.1 and 5.2. The detailed proof of Theorems 5.1 and 5.2 is presented in Appendices E.1 and E.2.

Theorem 5.1. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. There exists a logarithmic-precision Transformer with constant depth and hidden dimension (independent of n) that can generate the correct output for any input on the $\mathrm{ADD}_p(n)$ task.*

Theorem 5.2. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any integers n and k, there exists a logarithmic-precision Transformer with constant depth and hidden dimension d (independent of n and k) that can generate the correct output for any input on the $\mathrm{IterADD}_p(n,k)$ task.*

We now turn to integer multiplication. As established in Theorem 5.3, a logarithmic-precision Transformer with constant depth and polynomial hidden dimensions is capable of solving the integer multiplication task. The detailed proof of Theorem 5.3 is presented in Appendix E.3.

Theorem 5.3. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any integers n and $l \leq 2n$, there exists a logarithmic-precision Transformer with constant depth (independent of n and k) and hidden dimensions $O(n^2)$ that can generate the correct output for any input on the $\mathrm{MUL}_p(n,l)$ task.*

Theorems 5.1 to 5.3 demonstrate that, under standard precision, a bounded-depth Transformer with reasonable size can solve all elementary arithmetic tasks. Compared to the theoretical results for low-precision Transformers (Theorems 4.1 to 4.3), even a modest increase in numerical precision leads to a substantial improvement in expressiveness for arithmetic tasks.

The Reason for Increased Expressiveness. The transition from constant precision to logarithmic precision enables Transformers to process and represent large numbers effectively, thereby expanding their expressiveness beyond the capabilities of low-precision models. In particular, the expressiveness of a logarithmic-precision Transformer with polynomial size and bounded depth is upper-bounded by the computational complexity class TC^0 (Merrill and Sabharwal, 2023). Leveraging this increased precision, we constructively prove that logarithmic-precision Transformers are sufficient for solving these arithmetic tasks. These results underscore the critical role of numerical

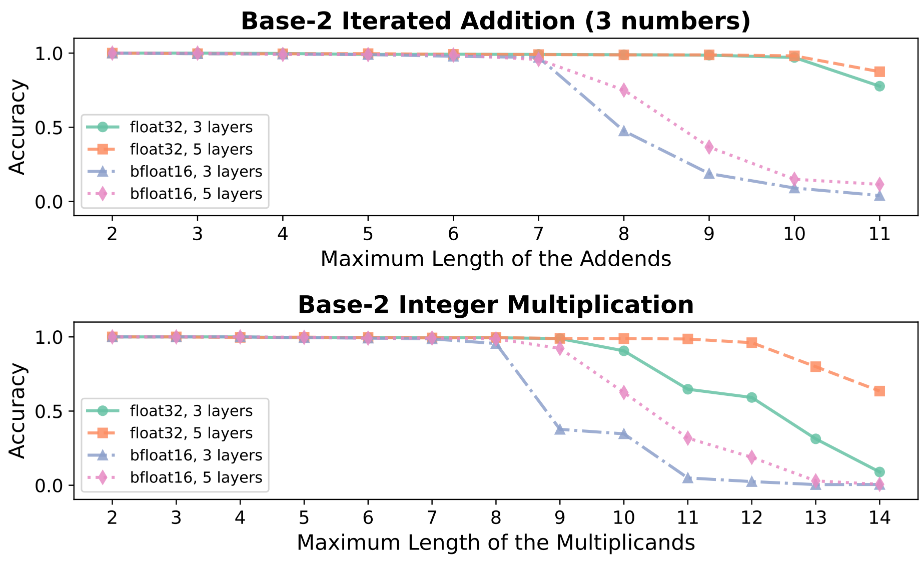

Figure 2: Model performance on different tasks in base-2. Within each sub-figure, the x-axis represents the maximum digits length and the y-axis represents the accuracy gained by each model. The figure indicates that, for all tasks, Transformers utilizing `float32` with 3 layers and 5 layers outperform their `bfloat16` counterparts.

precision in enhancing the expressiveness of Transformer architectures.

Practical Implications. Our theoretical results underscore the critical importance of numerical precision when deploying Transformers for arithmetic tasks. Under low-precision settings, a Transformer requires super-polynomial model size to solve even elementary arithmetic problems, which is impractical for real-world applications. While low-precision models may offer computational efficiency, they are likely to fail in scenarios that demand accurate numerical reasoning, such as mathematical problem-solving or scientific computing. However, a slight increase in precision—such as using `float32`—enables Transformers to handle more complex arithmetic operations while maintaining a reasonable hidden dimension. Thus, employing sufficient numerical precision is crucial for ensuring both accuracy and robustness in arithmetic tasks, and should be a key consideration when designing or deploying LLMs for applications involving complex arithmetic reasoning.

6 Experiments

In the preceding sections, we employ complexity theory to demonstrate that low-precision Transformers face significant challenges in performing elementary arithmetic tasks. To validate these theoretical insights, we conduct a series of experiments to compare the performance of Transformers under different precisions. The results provide empirical evidence that the model's ability to execute arithmetic operations drops as precision decreases, reinforcing our theoretical results.

6.1 Experimental Setup

Tasks and datasets. We evaluate three elementary arithmetic tasks: integer addition, iterated addition, and integer multiplication, as presented in Figure 1. Each task involves a series of experiments with base $p = 2, 10$ and varying choices of digit length n. For integer addition, we examine the addition of integers in both base-2 and base-10, with digit lengths $n \in \{4, 8, 16, 32, 64\}$. For iterated addition, we examine the addition of three numbers in base-2, with digit lengths $n \in [2, 11]$, as well as in base-10, with digit lengths $n \in [1, 4]$. Similarly, for integer multiplication, we run experiments in base-2 with digit lengths $n \in [2, 14]$, and in base-10 with digit length $n \in [2, 5]$. Both training data and test data are dynamically generated. We use a batch size of 512 with 100k steps, resulting in a total training dataset size of 51.2M. Further details regarding the data generation function and the construction of datasets are provided in Algorithms 4 and 5.

Training and Evaluation. All experiments use Transformers as the backbone. We trained models with 3 and 5 layers and evaluated their performance on each task. Detailed model and training configurations are listed in Tables 2 and 3. No prompts or chat templates were added to the dataset. The models were trained with cross-entropy loss over the answer tokens. During evaluation, the models were required to produce exact answers, with accuracy reported as the evaluation metric. For each task, accuracy was computed over 50k test samples. To assess the impact of numerical precision, experiments were conducted with `float32`

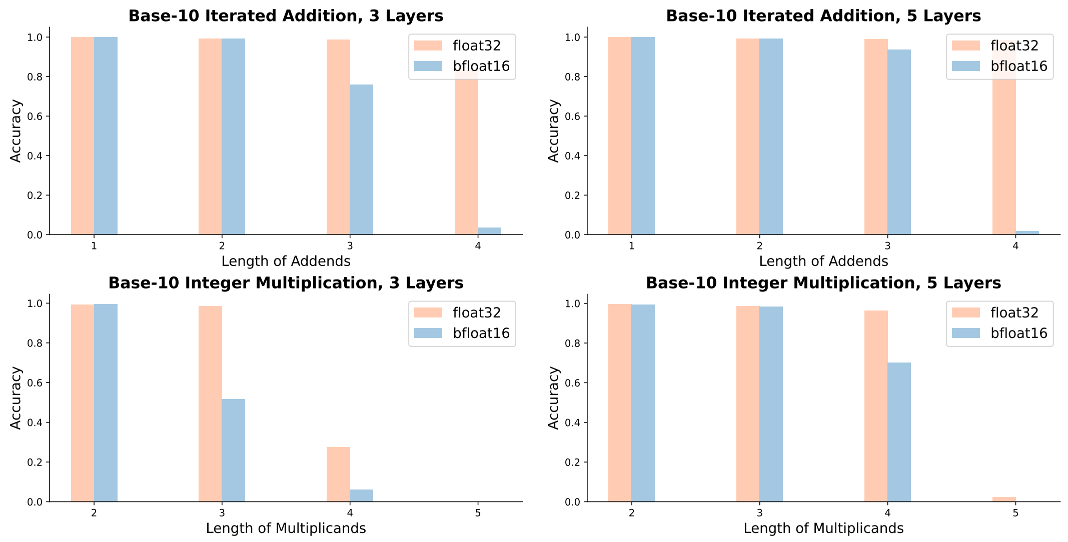

Figure 3: Model performance on iterated addition tasks involving three numbers and integer multiplication tasks. Each sub-figure presents a comparison of the performance between `float32` and `bfloat16`.

and `bfloat16`.

6.2 Experimental Results

Integer addition proved relatively simple, maintaining over 94% accuracy even as digit lengths increased to 32 across both base-2 and base-10 for both `float32` and `bfloat16` (see Appendix F.3).

The results for iterated addition and multiplication in base-2 are shown in Figure 2, while the corresponding base-10 results are presented in Figure 3. In each sub-figure, the x-axis represents the maximum digit length for addends or multiplicands, while the y-axis indicates test accuracy.

For iterated addition, accuracy under `bfloat16` declined significantly as the digit length increased, while `float32` consistently achieved near-perfect accuracy across all model depths. Specifically, in base-2, 16-bit precision exhibited a pronounced decline for digit lengths between 7 and 10, whereas 32-bit precision maintained high accuracy. In base-10, at digit lengths up to 10, `float32` achieved over 90% accuracy, whereas `bfloat16` struggled to produce correct results.

In the multiplication task, the gap between the two precisions became even more apparent as digit lengths increased. For example, at a digit length of 13 in base-2, 16-bit precision accuracy dropped sharply, signifying its inability to handle such inputs. Similarly, in base-10, 16-bit precision showed a marked reduction in accuracy, particularly for inputs with lengths of 3 in 3-layer models and lengths of 4 in 5-layer models. These results underscore the critical role of precision in achieving reliable performance for elementary arithmetic tasks, consistent with our theoretical findings.

6.3 Further Experiments on LLMs

To further substantiate our theoretical results, we conducted additional experiments on LLMs, specifically evaluating the LLAMA-3.1-8B Instruct model on elementary arithmetic tasks.

Task Description. For integer addition, we tested the addition of two base-10 integers with digit lengths ranging from 1 to 13. For iterated addition, we extended the task to include three and five base-10 numbers, with digit lengths spanning 1 to 9 and 1 to 5, respectively. For integer multiplication, we evaluated the multiplication of two base-10 numbers, with digit lengths varying from 1 to 5. Data generation followed the same procedure as earlier experiments, with details provided in Algorithms 4 and 5.

Model Configuration. All experiments used the LLAMA-3.1-8B Instruct model (Dubey et al., 2024). To study the effects of reduced precision, we evaluated the model under four settings:

- Original model operating under `bfloat16`
- Quantized model operating under `int4`
- Fine-tuned model using LoRA (`bfloat16`)
- Fine-tuned model using QLoRA (`int4`)

The baseline configuration employs the original LLaMA-3.1-8B Instruct model, which operates under `bfloat16` precision. To assess the effects of reduced precision, we applied 4-bit quantization using the AWQ algorithm (Lin et al., 2024). Further, we fine-tuned the model using LoRA and QLoRA (Hu et al., 2021; Dettmers et al., 2024). The fine-tuning configurations for LoRA and QLoRA are listed in Table 6 in Appendix F.4. For the LoRA fine-tuning experiments, model weights were maintained in `bfloat16`. In contrast, the QLoRA ex-

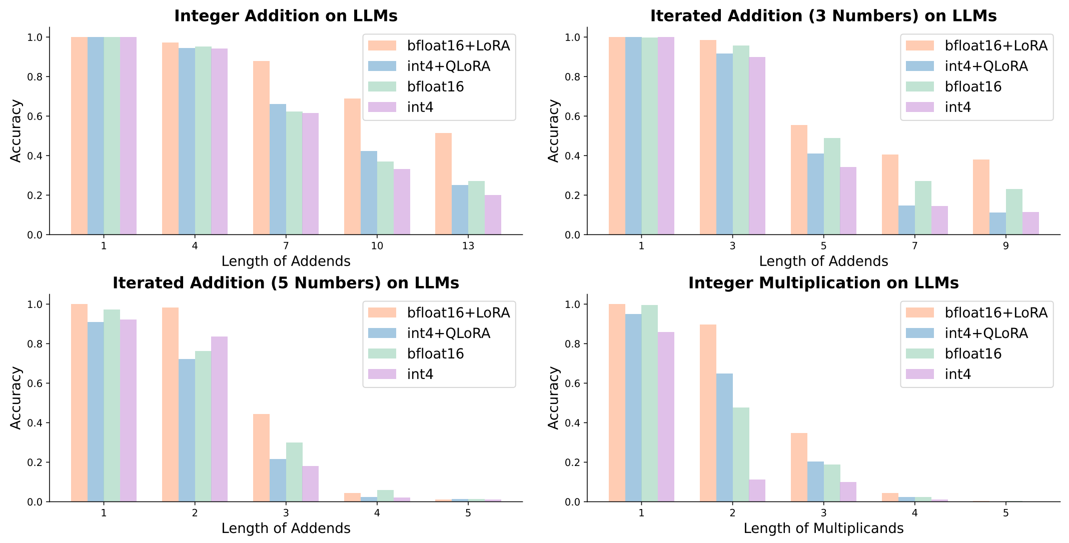

Figure 4: The performance of LLAMA-3.1-8B Instruct model on arithmetic tasks in base-10. In each sub-figure, we compare the original model in `bfloat16` and the quantized model in `int4`, alongside fine-tuned models, with LoRA using `bfloat16` and QLoRA using `int4`.

periments extended this setup by enabling 4-bit quantization, represented by the `int4`. Fine-tuning was performed individually for each task. Furthermore, we add a baseline of GPT-4o (OpenAI, 2023) as a reference, whose results are listed in Table 9.

Dataset for Fine-tuning. We generated the fine-tuning data for both multiplication and addition tasks, with both multiplicands and addends varying in length from 1 to 9. The dataset comprised a total of 60k samples, including 5k samples with lengths between 1 and 6, and 10k samples with lengths between 7 and 9. The generation process of the dataset is the same as in previous experiments. Furthermore, we add the few-shot learning prompt to the raw dataset and apply the LLaMA chat template for data preprocessing. The prompt for few-shot learning can be found in Tables 7 and 8.

Evaluation. For the evaluation, we employ a few-shot learning approach for inference. The prompts are the same as the prompts of fine-tuning dataset and can be found in Appendix F.1. The generation configurations for LLMs can be also found in Table 5 in Appendix F.4. During inference, the LLMs were tasked with producing exact solutions to the given arithmetic problems. Both the original model and the model fine-tuned with LoRA are evaluated using `bfloat16`, whereas the quantized model and the model fine-tuned with QLoRA are evaluated using `int4`. For each task, we evaluate the model on 1k samples to compute the accuracy serving as the evaluation metric.

The results of the experiments are shown in Figure 4. Each sub-figure presents the results of a task, where the x-axis denotes the maximum length of the addends or multiplicands, and the y-axis represents the test accuracy. For each task, reducing numerical precision in both the original and fine-tuned models leads to a significant decrease in accuracy. Specifically, in the iterated addition task for 3 numbers, accuracy drops by nearly 20% as the length of the addends increases. Similarly, for models fine-tuned with QLoRA and LoRA, lowering precision also results in a decline in accuracy. Furthermore, in some cases, even after fine-tuning a low-precision model with QLoRA, the performance does not surpass that of the original model with standard precision. These experimental findings support our theoretical results that numerical precision is a critical factor in the success of iterated addition and integer multiplication tasks. Overall, the results underscore the consistency between the precision requirements for these elementary arithmetic tasks and our theoretical predictions.

7 Conclusion

In this work, we have theoretically analyzed the impact of numerical precision on LLMs for arithmetical reasoning. By focusing on three elementary arithmetic tasks, integer addition, iterated addition, and integer multiplication, we demonstrate that the Transformers operating under standard precision can handle these tasks effectively. In contrast, Transformers with low precision struggle with complex arithmetic tasks, excelling only at integer addition. Extensive experimental results corroborate our theoretical findings, showing that standard precision models outperform low precision ones. We believe this study offers valuable insights for developing more powerful LLMs in mathematics.

8 Limitations

One limitation of this work is that we have not fully explored all key components of mathematical reasoning. While the arithmetic tasks considered are foundational, there remain other essential elements of mathematical reasoning whose dependence on numerical precision is still unclear. Additionally, our focus was exclusively on numerical precision, but we acknowledge that other factors are likely to play a significant role in applying LLMs to mathematical reasoning. We leave these explorations for future work.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta. Association for Computational Linguistics.

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2022. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*.

Silas Alberti, Niclas Dern, Laura Thesing, and Gitta Kutyniok. 2023. Sumformer: Universal approximation for efficient transformers. In *Topological, Algebraic and Geometric Learning Workshops 2023*, pages 72–86. PMLR.

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Learning from mistakes makes llm better reasoner. *Preprint*, arXiv:2310.20689.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Sanjeev Arora and Boaz Barak. 2009. *Computational complexity: a modern approach*. Cambridge University Press.

Lochan Basyal and Mihir Sanghvi. 2023. Text summarization using large language models: A comparative study of mpt-7b-instruct, falcon-7b-instruct, and openai chat-gpt models. *Preprint*, arXiv:2310.10449.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*.

David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in neural information processing systems*, volume 33, pages 1877–1901.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Vincent Cheng and Zhang Yu. 2023. Analyzing ChatGPT's mathematical deficiencies: Insights and contributions. In *Proceedings of the 35th Conference on Computational Linguistics and Speech Processing (ROCLING 2023)*, pages 188–193, Taipei City, Taiwan. The Association for Computational Linguistics and Chinese Language Processing (ACLCLP).

David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter bounds on the expressivity of transformer encoders. In *Proceedings of the 40th International Conference on Machine Learning*, pages 5544–5562.

Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. 2023. Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang (Lorraine) Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. 2023. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems*, volume 36, pages 70293–70332. Curran Associates, Inc.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. 2021. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. 2023. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Guhao Feng and Han Zhong. 2023. Rethinking model-based, policy-based, and value-based reinforcement learning via the lens of representation complexity. *arXiv preprint arXiv:2312.17248*.

Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Petersen, and Julius Berner. 2024. Mathematical capabilities of chatgpt. *Advances in neural information processing systems*, 36.

Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. What can transformers learn in-context? a case study of simple function classes. In *Advances in Neural Information Processing Systems*.

Siavash Golkar, Mariel Pettee, Alberto Bietti, Michael Eickenberg, Miles Cranmer, Geraud Krawezik, Francois Lanusse, Michael McCabe, Ruben Ohana, Liam Holden Parker, Bruno Régaldo-Saint Blancard, Tiberiu Tesileanu, Kyunghyun Cho, and Shirley Ho. 2024. xval: A continuous number encoding for large language models.

Sophia Gu. 2023. Llms as potential brainstorming partners for math and science problems. *Preprint*, arXiv:2310.10677.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171.

Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

Yiding Hao, Dana Angluin, and Robert Frank. 2022. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810.

Joy He-Yueya, Gabriel Poesia, Rose E. Wang, and Noah D. Goodman. 2023. Solving math word problems by combining language models with symbolic solvers. *Preprint*, arXiv:2304.09102.

Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

IEEE. 2019. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. MathPrompter: Mathematical reasoning using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 37–42, Toronto, Canada. Association for Computational Linguistics.

Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. 2024. A comprehensive evaluation of quantization strategies for large language models. *arXiv preprint arXiv:2402.16775*.

William Kahan. 1996. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11.

Jikun Kang, Xin Zhe Li, Xi Chen, Amirreza Kazemi, Qianyi Sun, Boxing Chen, Dong Li, Xu He, Quan He, Feng Wen, Jianye Hao, and Jun Yao. 2024. Mindstar: Enhancing math reasoning in pre-trained llms at inference time. *Preprint*, arXiv:2405.16265.

Tanishq Kumar, Zachary Ankner, Benjamin F. Spector, Blake Bordelon, Niklas Muennighoff, Mansheej Paul, Cengiz Pehlevan, Christopher Ré, and Aditi Raghunathan. 2024. Scaling laws for precision. *Preprint*, arXiv:2411.04330.

Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. 2024. Teaching arithmetic to small transformers. In *The Twelfth International Conference on Learning Representations*.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. 2024. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*.

Zhenwen Liang, Dian Yu, Xiaoman Pan, Wenlin Yao, Qingkai Zeng, Xiangliang Zhang, and Dong Yu. 2024. MinT: Boosting generalization in mathematical reasoning via multi-view fine-tuning. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 11307–11318, Torino, Italia. ELRA and ICCL.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024.

Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. 2023. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*.

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. 2024. Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts. In *The Twelfth International Conference on Learning Representations*.

Shengjie Luo, Shanda Li, Shuxin Zheng, Tie-Yan Liu, Liwei Wang, and Di He. 2022. Your transformer may not be as powerful as you expect. In *Advances in Neural Information Processing Systems*.

Yujun Mao, Yoon Kim, and Yilun Zhou. 2024. Champ: A competition-level dataset for fine-grained analyses of llms' mathematical reasoning capabilities. *Preprint*, arXiv:2401.06961.

Kelly Marchisio, Saurabh Dash, Hongyu Chen, Dennis Aumiller, Ahmet Üstün, Sara Hooker, and Sebastian Ruder. 2024. How does quantization affect multilingual llms? *arXiv preprint arXiv:2407.03211*.

Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, and Tom Goldstein. 2024. Transformers can do arithmetic with the right embeddings. *Preprint*, arXiv:2405.17399.

William Merrill and Ashish Sabharwal. 2023. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*.

William Merrill, Ashish Sabharwal, and Noah A Smith. 2022. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, and Ashwin Kalyan. 2022. LILA: A unified benchmark for mathematical reasoning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5807–5832, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. 2021. Investigating the limitations of transformers with simple arithmetic tasks. *Preprint*, arXiv:2102.13019.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2022. In-context learning and induction heads. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Xu Ouyang, Tao Ge, Thomas Hartvigsen, Zhisong Zhang, Haitao Mi, and Dong Yu. 2024. Low-bit quantization favors undertrained llms: Scaling laws for quantized llms with 100t training tokens. *Preprint*, arXiv:2411.17691.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. 2021. Attention is turing complete. *The Journal of Machine Learning Research*, 22(1):3463–3497.

Jorge Pérez, Javier Marinković, and Pablo Barceló. 2019. On the turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Syed Rifat Raiyan, Md Nafis Faiyaz, Shah Md. Jawad Kabir, Mohsinul Kabir, Hasan Mahmud, and Md Kamrul Hasan. 2023. Math word problem solving by generating linguistic variants of problem statements. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 362–378, Toronto, Canada. Association for Computational Linguistics.

Alexander A Razborov. 1987. Lower bounds for the size of circuits of bounded depth with basis f^; g. *Math. notes of the Academy of Sciences of the USSR*, 41(4):333–338.

Ankit Satpute, Noah Gießing, André Greiner-Petter, Moritz Schubotz, Olaf Teschke, Akiko Aizawa, and Bela Gipp. 2024. Can llms master math? investigating large language models on math stack exchange. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 2316–2320, New York, NY, USA. Association for Computing Machinery.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*.

Paulo Shakarian, Abhinav Koyyalamudi, Noel Ngu, and Lakshmivihari Mareedu. 2023. An independent evaluation of chatgpt on mathematical word problems (mwp). *Preprint*, arXiv:2302.13814.

Yunfan Shao, Linyang Li, Junqi Dai, and Xipeng Qiu. 2023. Character-LLM: A trainable agent for role-playing. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13153–13187, Singapore. Association for Computational Linguistics.

Ruoqi Shen, Sebastien Bubeck, Ronen Eldan, Yin Tat Lee, Yuanzhi Li, and Yi Zhang. 2024. Positional description matters for transformers arithmetic.

Roman Smolensky. 1987. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 77–82.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Pragya Srivastava, Manuj Malik, Vivek Gupta, Tanuja Ganu, and Dan Roth. 2024. Evaluating LLMs' mathematical reasoning in financial document question answering. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3853–3878, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. 2023. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pages 35151–35174. PMLR.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Colin Wei, Yining Chen, and Tengyu Ma. 2022a. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *Advances in Neural Information Processing Systems*, 35:12071–12083.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022b. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR.

Kaiyue Wen, Xingyu Dang, and Kaifeng Lyu. 2024. Rnns are not transformers (yet): The key bottleneck on in-context retrieval. *Preprint*, arXiv:2402.18510.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024a. An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2024b. Mathchat: Converse to tackle challenging math problems with llm agents. *Preprint*, arXiv:2306.01337.

Ryutaro Yamauchi, Sho Sonoda, Akiyoshi Sannai, and Wataru Kumagai. 2023. Lpml: Llm-prompting markup language for mathematical reasoning. *Preprint*, arXiv:2309.13078.

Kai Yang, Jan Ackermann, Zhenyu He, Guhao Feng, Bohang Zhang, Yunzhen Feng, Qiwei Ye, Di He, and Liwei Wang. 2024. Do efficient transformers really save computation? In *Forty-first International Conference on Machine Learning*.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. 2021. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3770–3785.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2024. MAmmoTH: Building math generalist models through hybrid instruction tuning. In *The Twelfth International Conference on Learning Representations*.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. 2019. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077*.

Chulhee Yun, Yin-Wen Chang, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2020. O (n) connections are expressive enough: Universal approximability of sparse transformers. In *Advances in Neural Information Processing Systems*, volume 33, pages 13783–13794.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2024a. Solving challenging math word problems using GPT-4 code interpreter with code-based self-verification. In *The Twelfth International Conference on Learning Representations*.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua M. Susskind, Samy Bengio, and Preetum Nakkiran. 2024b. What algorithms can transformers learn? a study in length generalization.

In *The Twelfth International Conference on Learning Representations*.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. 2024c. Transformers can achieve length generalization but not robustly. In *ICLR 2024 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. 2024. Multilingual machine translation with large language models: Empirical results and analysis. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2765–2781, Mexico City, Mexico. Association for Computational Linguistics.

A Related Work

A.1 LLMs for Mathematical Reasoning

Mathmetical Reasoning. Recent studies highlight the limitations of current LLMs in mathematical reasoning (Ahn et al., 2024; Srivastava et al., 2024). Satpute et al. (2024) demonstrated that advanced models like GPT-4 can generate relevant answers, but these answers are not always accurate. Additionally, Mao et al. (2024) found that current LLMs struggle even with verifying the solutions to mathematical problems. To enhance the mathematical capabilities of LLMs, several studies have carefully designed prompting strategies (Shakarian et al., 2023; Cheng and Yu, 2023; Gu, 2023; Lu et al., 2024) or finetuned LLMs on mathematics-related datasets (An et al., 2024; Liang et al., 2024; Raiyan et al., 2023; Mishra et al., 2022; Yue et al., 2024). Other approaches include inference-based searching methods (Kang et al., 2024), the application of external tools (Yamauchi et al., 2023; He-Yueya et al., 2023; Chen et al., 2023), and the introduction of simulated interaction processes (Wu et al., 2024b) or self-verification mechanisms (Wang et al., 2023; Zhou et al., 2024a).

Arithmetical Reasoning. Bubeck et al. (2023) highlighted arithmetical reasoning as a key component of true mathematical ability. However, Saxton et al. (2019); Dziri et al. (2023) identified significant challenges that LLMs encounter when solving elementary arithmetic tasks, such as multi-digit addition and multiplication. A common approach to mitigate these difficulties is to reverse the output digit order (Shen et al., 2024), or both the input and output digit order simultaneously (Lee et al., 2024). Other studies have focused on developing improved positional encodings (Golkar et al., 2024; McLeish et al., 2024) or positional tokens (Nogueira et al., 2021) that are more suitable for arithmetic tasks. Zhou et al. (2024b,c) further examined the length extrapolation capabilities of LLMs in solving basic arithmetic problems, emphasizing the importance of data formats and positional embeddings for better generalization.

A.2 Computational Powers of Transformers

Another more relevant line of work investigates the theoretical expressive power of Transformers from a computational perspective.

Universal Approximation. Early theoretical work on Transformers primarily focused on their function approximation capabilities. Yun et al. (2019) demonstrated that Transformers can universally approximate any continuous sequence-to-sequence functions, given sufficient size. This universality result has since been extended to various Transformer variants, such as Sparse Transformers (Yun et al., 2020), Linear Transformers (Alberti et al., 2023), and Transformers with relative positional encodings (RPE) (Luo et al., 2022). Additionally, previous studies established that infinite-precision Transformers are Turing-complete (Pérez et al., 2019, 2021), while Wei et al. (2022a) showed that finite-precision Transformers are approximately Turing-complete. Although these results highlight Transformers' computational capacity, our work develops expressiveness results under more practical settings, exploring the differences in expressiveness across varying levels of numerical precision.

Formal Language Learning. Another line of research focuses on the ability of Transformers to learn formal languages. Liu et al. (2023) explored how Transformers simulate finite state automata, while Bhattamishra et al. (2020); Yao et al. (2021) studied their ability to recognize counter languages and Dyck languages, respectively. On the negative side, Hahn (2020) showed that Transformers are not capable of learning distributions over languages. In addition to affirmative results, several works have characterized the limitations of Transformers from the perspective of formal language modeling (Hahn, 2020; Bhattamishra et al., 2020; Weiss et al., 2021; Yao et al., 2021; Chiang et al., 2023) or circuit simulation (Hao et al., 2022; Merrill et al., 2022; Merrill and Sabharwal, 2023). However, few of these studies focus on the autoregressive Transformers commonly used in LLMs, which we investigate in this paper.

Chain-of-Thought and In-Context Learning. Chain-of-Thought prompting (Wei et al., 2022b) plays a crucial role in tasks requiring complex reasoning structures, and several studies aim to understand its underlying mechanisms. For instance, Feng et al. (2023); Li et al. (2024) analyzed CoT from an expressiveness perspective, and Yang et al. (2024); Wen et al. (2024) examined CoT across more different model variants. In-context learning (Brown et al., 2020; Garg et al., 2022) is another powerful aspect of LLMs. Some theoretical work has shown that in-context learning can be explained through gradient descent (Akyürek et al., 2022; Dai et al., 2023; Von Oswald et al., 2023), while others attribute it to the induction heads mechanism (Elhage et al., 2021; Olsson et al., 2022).

A.3 Scaling Laws of Precision

Concurrent works (Kumar et al., 2024; Ouyang et al., 2024) explore the impact of numerical precision on scaling laws, particularly in the contexts of training and quantization. Kumar et al. (2024) introduced "precision-aware" scaling laws, demonstrating that low-precision training effectively reduces a model's "effective parameter count" but may still be compute-optimal for larger models. Their framework unifies the effects of both training and post-training quantization. Ouyang et al. (2024) examined quantization-induced degradation (QiD), showing that larger or undertrained models exhibit greater robustness to low-bit quantization, whereas fully trained models experience significant performance degradation. While both studies underscore precision as a critical dimension in scaling laws, they leave theoretical gaps in understanding the role of precision for LLMs. Our work focuses on addressing these gaps by analyzing the impact of numerical precision on elementary arithmetic reasoning tasks.

B Additional Background and Preliminary

B.1 Circuit Complexity

Circuit complexity classes capture various aspects of computational complexity, typically bounding circuit width and depth. For a more detailed introduction, we refer to Arora and Barak (2009).

We begin by defining Boolean circuits. A Boolean circuit over a basis of gates is represented as a finite-size directed acyclic graph (DAG), where each vertex corresponds to either a basis function (or gate) or an input bit. Some internal nodes are designated as outputs, and the *fan-in* of a vertex is defined as its in-degree. Building on this definition, we can define the complexity classes NC^i, AC^i, and TC^i:

- NC^i: This class consists of constant fan-in, polynomial-sized circuits made up of AND, OR, and NOT gates, with a depth of $O(\log^i n)$.
- AC^i: This class includes unbounded fan-in, polynomial-sized circuits composed of AND, OR, and NOT gates (with NOT gates allowed only on inputs), also having a depth of $O(\log^i n)$.
- TC^i: This class extends AC^i by allowing majority gates.

The relationships among the NC, AC, and TC hierarchies are as follows:

$$\mathsf{NC}^i \subset \mathsf{AC}^i \subset \mathsf{TC}^i \subset \mathsf{NC}^{i+1}, \ \mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{TC}^0.$$

B.2 Constant-precision Transformer

Previous work has investigated the expressiveness of constant-precision Transformers (Li et al., 2024), utilizing a simplified version of the IEEE 754 standards (IEEE, 2019). Our constant-precision setting is analogous, and we will introduce the floating-point representations we consider here.

Definition B.1. A $(e + 2s + 1)$-floating point representation includes e exponent bits, $2s$ precision bits, and one sign bit. The numbers representable under this representation are defined as follows:

$$\mathbb{F}_{e,s} := \{S \cdot 2^{-s+E} \mid -2^{-2s} + 1 \leq S \leq 2^{2s} - 1, -2^{e-1} \leq E \leq \max(2^{e-1} - 1, 0), S, E \in \mathbb{Z}\}.$$

For any $x \in \mathbb{R}$, its representation under this floating-point format is determined by rounding to the nearest value in $\mathbb{F}$. In the event of a tie, we select the number with the smaller absolute value.

In this paper, we focus on the case where $e = 0$, which means all representable numbers take the form $S \cdot 2^{-s}$, with $S \in \mathbb{Z}$ such that $-2^{-2s} + 1 \leq S \leq 2^{2s} - 1$. However, this is necessary only for Theorem 4.1, while Theorems 4.2 and 4.3 do not depend on specific numerical representations.

Definition B.2 (Constant-Precision Transformer)**.** A *constant-precision Transformer* is a Transformer in which each neuron and activation are restricted to using a constant number of bits for computation.

Li et al. (2024) demonstrated that constant-precision Transformers with constant depth belong to the complexity class AC^0.

B.3 Logarithmic-precision Transformer

A key limitation of constant-precision representation is that it fails to capture the input size n within a single neuron. To address this, we consider logarithmic precision, allowing for $O(\log n)$ bits for numerical representations.

Definition B.3 (Logarithmic-Precision Transformer)**.** A *logarithmic-precision Transformer* is a Transformer in which each neuron and activation are allowed to use $O(\log n)$ bits for computation, where n denotes the size of the input.

Logarithmic-precision Transformers possess several advantageous properties (Feng et al., 2023; Feng and Zhong, 2023):

- For floating-point representations with $O(\log n)$ bits, any real number $x \in O(\mathrm{poly}(n))$ can be represented with $O(\mathrm{poly}(1/n))$ error.
- Each neuron in the Transformer can only store $O(\log n)$ bits of information, which means it cannot retain all input data. Consequently, computation must be distributed across the network, aligning with the operational principles of Transformers.

Previous work (Merrill et al., 2022; Merrill and Sabharwal, 2023) has shown that logarithmic-precision Transformers fall within the complexity class TC^0.

B.4 Tokenization Scheme

In this section, we formalize the tokenization scheme adopted in this paper and provide the necessary definitions and examples to establish a foundation for the subsequent analysis.

Definition B.4 (Tokenizer $\boldsymbol{T}_c$)**.** Let $\mathbf{x} = (x_{n-1} \cdots x_0)_p$ denote an n-digit integer in base p. The tokenizer $\boldsymbol{T}_c$ maps $\mathbf{x}$ into $k = \lceil \frac{n}{c} \rceil$ tokens, represented as $\boldsymbol{t} = [t_{k-1}, \cdots, t_0]$, where

$$t_i = \begin{cases} [x_{ic}, x_{ic+1}, \cdots, x_{ic+c-1}], & i < k; \\ [x_{ic}, x_{ic+1}, \cdots, x_{n-1}], & i = k. \end{cases}$$

Furthermore, for any operator (e.g., “$+$”, “$\times$”, “$=$”), the tokenizer $\boldsymbol{T}_c$ assigns each operator a single token.

Example B.5. Consider the 5-digit integer 13215. Under the tokenizer $\boldsymbol{T}_3$, it is tokenized into $[13, 215]$.

A key property of this tokenization scheme is that, for any fixed base p and tokenizer $\boldsymbol{T}_c$, the resulting tokenized sequence can be reinterpreted as an arithmetic expression in base p^c. Specifically, there exists a one-to-one mapping τ between the vocabulary of the base-p tokenizer $\boldsymbol{T}_c$ and the vocabulary of the base-p^c tokenizer $\boldsymbol{T}_1$, such that

$$\tau(\boldsymbol{T}_c([a_{c-1}, \cdots, a_0]_p)) = \boldsymbol{T}_1\left(\left[\sum_{i\in[c]} a_i p^i\right]\right).$$

Proposition B.6. *Let $\boldsymbol{a}$ be an integer. If $\mathbf{t} = \boldsymbol{T}_c(\boldsymbol{a}) = [t_{k-1}, \cdots, t_0]$ and $\mathbf{t}' = \boldsymbol{T}_1(\boldsymbol{a}) = [t'_{k-1}, \cdots, t'_0]$, then for all i, we have $\tau(t_i) = t'_i$.*

This property is particularly significant as it allows us to abstract away the specific effects of the tokenizer $\boldsymbol{T}_c$ and focus exclusively on the case where the tokenizer is $\boldsymbol{T}_1$. This simplification is leveraged in proving the main theorems presented in this paper.

Example B.7 illustrates how a tokenized sequence in base-10, generated using the tokenizer $\boldsymbol{T}_3$, can be equivalently interpreted as a sequence in base-1000.

Example B.7. Consider the base-10 arithmetic expression $44505 + 9416 = 53921$. When tokenized using $\boldsymbol{T}_3$, the sequence becomes $[44, 505, +, 9, 416, =, 53, 921]$. This tokenized representation can then be reinterpreted as an arithmetic expression in base-1000.

C Technical Lemmas

C.1 Technical Lemmas for Logarithmic Precision MLP

In this subsection, we present several foundational results concerning logarithmic precision multi-layer perceptrons (MLPs), as introduced in (Feng et al., 2023). For brevity, proofs of these results are omitted here but are available in the appendix of (Feng et al., 2023).

Lemma C.1 (Feng et al., 2023, Lemma C.1)**.** *Let $\epsilon > 0$. There exists a two-layer MLP $f : \mathbb{R}^2 \to \mathbb{R}$ with four hidden units and* GeLU *activation, such that for any $a, b \in [-M, M]$, the inequality $|f(a, b) - ab| \le \epsilon$ holds. Furthermore, the ℓ_∞ norm of f is bounded by $O(\mathrm{poly}(M, 1/\epsilon))$.*

Lemma C.2 (Feng et al., 2023, Lemma C.2)**.** *Let $\boldsymbol{g} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ be a two-layer MLP with* ReLU *activation and ℓ_∞ norm bounded by M. Then, for any $\epsilon > 0$, there exists a two-layer MLP $\boldsymbol{f}$ of the same size with* GeLU *activation such that for all $\boldsymbol{x} \in \mathbb{R}^{d_1}$, the inequality $\|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{g}(\boldsymbol{x})\|_\infty \le \epsilon$ is satisfied. Moreover, the ℓ_∞ norm of $\boldsymbol{f}$ is bounded by $O(\mathrm{poly}(M, 1/\epsilon))$.*

Lemma C.3 (Feng et al., 2023, Lemma C.4)**.** *Consider the selection function $\boldsymbol{g} : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ defined as*

$$\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t) = \begin{cases} \boldsymbol{x} & \text{if } t > 0, \\ \boldsymbol{y} & \text{otherwise}. \end{cases}$$

For any $\epsilon > 0$, $\alpha > 0$, and $M > 0$, there exists a two-layer MLP $\boldsymbol{f}$ with $2d + 2$ hidden units and GeLU *activation such that, for all $\boldsymbol{x} \in [-M, M]^d$, $\boldsymbol{y} \in [-M, M]^d$, and $t \in (-\infty, -\alpha] \cup [\alpha, +\infty)$, the inequality $\|\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t) - \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t)\|_\infty \le \epsilon$ holds. Furthermore, the ℓ_∞ norm of $\boldsymbol{f}$ is bounded by $O(\mathrm{poly}(M, 1/\alpha, 1/\epsilon))$.*

C.2 Technical Lemmas for Logarithmic Precision Attention Layer

Feng et al. (2023) investigated the expressive power of the standard attention layer and introduced two fundamental operations: **COPY** and **MEAN**, demonstrating that a standard attention layer with logarithmic precision can perform these operations under certain regularity conditions. In this subsection, we restate their results and extend the discussion to a specialized operation referred to as **SINGLE COPY**.

Consider a sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, where each $\boldsymbol{x}_i = (\tilde{\boldsymbol{x}}_i, r_i, 1) \in [-M, M]^{d+2}$, and M is a fixed constant. Let the attention matrices be $\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V} \in \mathbb{R}^{d' \times (d+2)}$, and define the following transformed vectors:

$$\boldsymbol{q}_i = \boldsymbol{Q}\boldsymbol{x}_i, \quad \boldsymbol{k}_j = \boldsymbol{K}\boldsymbol{x}_j, \quad \boldsymbol{v}_j = \boldsymbol{V}\boldsymbol{x}_j.$$

For any scalars $0 < \rho, \delta < M$, define the *matching set* as:

$$\mathcal{S}_i = \{j \le i : |\boldsymbol{q}_i \cdot \boldsymbol{k}_j| \le \rho\}.$$

Using this matching set, we define the following operations:

- **COPY**: The output is a sequence of vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$, where

  $$\boldsymbol{u}_i = \boldsymbol{v}_{\mathrm{pos}(i)}, \quad \text{with } \mathrm{pos}(i) = \mathrm{argmax}_{j \in \mathcal{S}_i} r_j.$$

  The output $\boldsymbol{u}_i$ is undefined if $\mathcal{S}_i = \varnothing$.

- **MEAN**: The output is a sequence of vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$, where

  $$\boldsymbol{u}_i = \mathrm{mean}_{j \in \mathcal{S}_i} \boldsymbol{v}_j = \frac{1}{|\mathcal{S}_i|} \sum_{j \in \mathcal{S}_i} \boldsymbol{v}_j.$$

  The output $\boldsymbol{u}_i$ is undefined if $\mathcal{S}_i = \varnothing$.

- **SINGLE COPY**: The output is a sequence of vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$, where

  $$\boldsymbol{u}_i = \boldsymbol{v}_{\mathrm{pos}(i)}, \quad \text{with } \mathrm{pos}(i) \text{ being the unique element in } \mathcal{S}_i.$$

  The output $\boldsymbol{u}_i$ is undefined if $|\mathcal{S}_i| \ne 1$.

We now impose the following regularity assumption to ensure the feasibility of the operations under consideration:

Assumption C.4 (Regularity Assumption for Attention)**.** For any input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, the matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ and scalars ρ, δ satisfy the following conditions:

- For any $i, j \in [n]$, either $|\boldsymbol{q}_i \cdot \boldsymbol{k}_j| \leq \rho$ or $\boldsymbol{q}_i \cdot \boldsymbol{k}_j \leq -\delta$.
- For any $i, j \in [n]$, either $i = j$ or $|r_i - r_j| \geq \delta$.
- The infinity norm of the value matrix $\boldsymbol{V}$ satisfies $\|\boldsymbol{V}\|_\infty \leq 1$.

Under this assumption, we demonstrate that a logarithmic precision attention layer with $O(d)$ embedding dimension and a single attention head can perform the operations defined in Section C.2.

Lemma C.5 (Feng et al., 2023, Lemma C.7)**.** *Suppose Assumption C.4 holds and $\rho \leq \frac{\delta^2}{8M}$. For any $\epsilon > 0$, there exists an attention layer with a single attention head and $O(d)$ embedding dimension that can approximate the* COPY *operation. Furthermore, the ℓ_∞ norm of the parameters is bounded by $O(\text{poly}(M, 1/\delta, \log(n), \log(1/\epsilon)))$.*

Formally, for any input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, let the attention layer outputs be $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n$. Then, for any $i \in [n]$ such that $\mathcal{S}_i \neq \varnothing$, the following holds:

$$\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty \leq \epsilon,$$

where $\boldsymbol{u}_i$ is the target output of the COPY *operation as defined in Section C.2.*

Lemma C.6 (Feng et al., 2023, Lemma C.8)**.** *Suppose Assumption C.4 holds and $\rho \leq \frac{\delta\epsilon}{16M\ln(4Mn/\epsilon)}$. For any $0 < \epsilon \leq M$, there exists an attention layer with a single attention head and $O(d)$ embedding dimension that can approximate the* MEAN *operation. Furthermore, the ℓ_∞ norm of the parameters is bounded by $O(\text{poly}(M, 1/\delta, \log(n), \log(1/\epsilon)))$.*

Formally, for any input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, let the attention layer outputs be $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n$. Then, for any $i \in [n]$ such that $\mathcal{S}_i \neq \varnothing$, the following holds:

$$\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty \leq \epsilon,$$

where $\boldsymbol{u}_i$ is the target output of the MEAN *operation as defined in Section C.2.*

The proofs of Lemmas C.5 and C.6 are omitted here for brevity. Complete proofs can be found in the appendix of Feng et al. (2023).

Lemma C.7. *Suppose Assumption C.4 holds and $\delta - \rho \geq c\rho$ for some constant $c > 0$. For any $\epsilon > 0$, there exists an attention layer with a single attention head and $O(d)$ embedding dimension that can approximate the* SINGLE COPY *operation. Furthermore, the ℓ_∞ norm of the parameters is bounded by $O(\text{poly}(M, 1/\delta, 1/c, \log(n), \log(1/\epsilon)))$.*

Formally, for any input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, let the attention layer outputs be $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n$. Then, for any $i \in [n]$ such that $|\mathcal{S}_i| = 1$, the following holds:

$$\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty \leq \epsilon,$$

where $\boldsymbol{u}_i$ is the target output of the SINGLE COPY *operation, as defined in Section C.2.*

Proof. We construct the query, key, and value vectors as follows:

- Query: $\lambda \boldsymbol{q}_i \in \mathbb{R}^d$
- Key: $\boldsymbol{k}_i \in \mathbb{R}^d$
- Value: $\boldsymbol{v}_i \in \mathbb{R}^d$

where $\lambda > 0$ is a constant to be determined. Denote $a_{i,j}$ as the attention score, defined as:

$$a_{i,j} = \frac{\exp(\lambda(\boldsymbol{q}_i \cdot \boldsymbol{k}_j))}{\sum_{j'} \exp(\lambda(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'}))}.$$

Since $\delta - \rho \geq c\rho$, it follows that $\delta - \rho \geq \frac{c}{c+1}\delta$. Setting

$$\lambda = \frac{(c+1)\ln\left(\frac{2nM}{\epsilon}\right)}{c\delta},$$

which is bounded by $O(\text{poly}(M, 1/\delta, 1/c, \log(n), \log(1/\epsilon)))$, we derive the following bounds for $a_{i,\text{pos}(i)}$:

$$\begin{aligned}
a_{i,\text{pos}(i)} &\geq \frac{\exp(-\lambda\rho)}{\exp(-\lambda\rho) + (n-1)\exp(-\lambda\delta)} && (2)\\
&= \frac{1}{1 + (n-1)\exp(-\lambda(\delta - \rho))}\\
&\geq 1 - (n-1)\exp(-\lambda(\delta - \rho)) && (3)\\
&\geq 1 - n\exp\left(-\ln\left(\frac{2nM}{\epsilon}\right)\right)\\
&= 1 - \frac{\epsilon}{2M}.
\end{aligned}$$

Here, Equation (2) follows from Assumption C.4 and the condition $|\mathcal{S}_i| = 1$, which ensures that for $j' \neq \text{pos}(i)$, $\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'} \leq -\delta$; Equation (3) uses the approximation $\frac{1}{1+x} \geq 1 - x$ for $x \geq 0$.

Thus, we can bound the error as follows:

$$\begin{aligned}
\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty &= \left\| \sum_j a_{ij}\boldsymbol{v}_j - \boldsymbol{v}_{\text{pos}(i)} \right\|_\infty\\
&\leq M\|\boldsymbol{V}\|_\infty \cdot \left(1 - a_{i,\text{pos}(i)} + \sum_{j \neq \text{pos}(i)} a_{ij}\right)\\
&= M\|\boldsymbol{V}\|_\infty \cdot (2 - 2a_{i,\text{pos}(i)})\\
&\leq \epsilon,
\end{aligned}$$

where the last inequality follows from the bound $a_{i,\text{pos}(i)} \geq 1 - \frac{\epsilon}{2M}$ and the constraint $\|\boldsymbol{V}\|_\infty \leq 1$ from Assumption C.4. This concludes the proof. □

C.3 Technical Lemmas for Constant Precision Calculations

In this section, we establish technical lemmas that underpin constant precision calculations. Assume a system with $2s$-bit fixed-point precision and no exponent bits, and let $B_s = 2^s - 2^{-s}$. The largest representable value in this system is B_s, while the smallest is $-B_s$.

Lemma C.8 (Li et al., 2024, Lemmas E.1 and E.2)**.** *For any $s \in \mathbb{N}_+$, it holds that* $\exp(-B_s) = 0$ *and* $\exp(B_s) = B_s$.

Proof. First, observe that $\exp(B_s) \geq \mathrm{e}B_s > 2^{s+1}$. Consequently, $\exp(-B_s) \leq 2^{-s-1}$, implying $\exp(-B_s) = 0$ due to the truncation to zero under the given precision. For the second claim, note that $\exp(B_s) \geq B_s + 1 > B_s$, which enforces $\exp(B_s) = B_s$ under the constant precision constraints. □

Lemma C.9. *For any $s \in \mathbb{N}_+$, we have* $\text{GeLU}(-B_s) = 0$.

Proof. To prove this, it suffices to show that $B_s\Phi(-B_s) \leq 2^{-s-1}$, where Φ denotes the cumulative distribution function (CDF) of the standard Gaussian distribution.

Case 1: $s = 1$. In this case, $B_s = \frac{3}{2}$. Thus,

$$B_s\Phi(-B_s) \leq \frac{3}{2}\Phi(-1) \leq \frac{3}{2} \cdot \frac{1-0.68}{2} < \frac{1}{4}.$$

Case 2: $s \geq 2$. For larger s, we proceed as follows:

$$\begin{aligned} B_s\Phi(-B_s) &= \frac{B_s}{\sqrt{2\pi}} \int_{B_s}^{+\infty} \mathrm{e}^{-\frac{x^2}{2}}\,\mathrm{d}x \\ &\leq \frac{B_s}{\sqrt{2\pi}} \int_{B_s}^{+\infty} \mathrm{e}^{-\frac{B_s x}{2}}\,\mathrm{d}x \\ &\leq \sqrt{\frac{2}{\pi}}\mathrm{e}^{-\frac{B_s^2}{2}} \leq \frac{2\sqrt{2}}{\sqrt{\pi}(B_s^2+2)} \\ &\leq \frac{2\sqrt{2}}{\sqrt{\pi}\cdot 2^{2s}} \leq \frac{1}{2^{s+1}}. \end{aligned}$$

This completes the proof. $\square$

D Proofs for Section 4

In this section, we present the formal proofs of the theorems stated in Section 4. Before delving into the proofs, we revisit the role of the tokenizer $\boldsymbol{T}_c$. As established in Appendix B.4 and Proposition B.6, it suffices to focus on the case of $\boldsymbol{T}_1$, where both the inputs and outputs are tokenized into single digits. This simplification is key to the subsequent analysis and constructions.

D.1 Proof for Theorem 4.1

Theorem 4.1. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. There exist constant-precision Transformers with constant depth L (independent of n) and hidden dimension $d = O(n^2)$ that can solve the $\mathrm{ADD}_p(n)$ task.*

To aid readability, we first describe an algorithm to perform the $\mathrm{ADD}_p(n)$ task (Algorithm 1) and prove its correctness. Subsequently, we construct a Transformer with the specified configuration in Theorem 4.1 that simulates Algorithm 1.

Algorithm 1: $\mathrm{ADD}_p(n)$ Algorithm

```
Input  : Two $p$-adic numbers $\boldsymbol{a}, \boldsymbol{b}$ with lengths $n_1$ and $n_2$, respectively.
Output : The sum of the inputs, $\boldsymbol{o}$, represented as a $p$-adic number with $(n+1)$ digits, where
         $n = \max(n_1, n_2)$.
1 Initialize $a_n = 0$ and $b_n = 0$;
2 foreach $i \in \{0, \cdots, n-1\}$ do
3     Compute the carry-on bits $\boldsymbol{c}$;
4     $i_\wedge = \max\{j \leq i \mid a_j + b_j \geq p\}$;
5     $i_\vee = \max\{j \leq i \mid a_j + b_j \leq p-2\}$;
6     $c_i = \mathbf{1}_{i_\wedge > i_\vee}$;
7 end
8 Compute the output digits $\boldsymbol{o}$: $o_i = (a_i + b_i + c_{i-1}) \bmod p$;
```

Lemma D.1 (An algorithm to perform $\mathrm{ADD}_p(n)$)**.** *Algorithm 1 outputs $\boldsymbol{o} = \boldsymbol{a} + \boldsymbol{b}$ for all inputs $\boldsymbol{a}, \boldsymbol{b}$.*

Proof. Consider two n-bit p-adic numbers $\boldsymbol{a}$ and $\boldsymbol{b}$. The carry-over bits $\boldsymbol{c} = (c_n, \cdots, c_1)$ can be computed recursively as follows:

$$\begin{aligned}
c_{-1} &= 0, \\
c_0 &= \mathbf{1}_{a_0+b_0 \geq p}, \\
c_1 &= (c_0 \cdot \mathbf{1}_{a_1+b_1 \geq p-1}) \vee \mathbf{1}_{a_1+b_1 \geq p}, \\
&\cdots, \\
c_i &= (c_{i-1} \cdot \mathbf{1}_{a_i+b_i \geq p-1}) \vee \mathbf{1}_{a_i+b_i \geq p}.
\end{aligned} \tag{4}$$

To avoid the recursive computation, the carry-over bits can be expressed in closed form as:

$$\begin{aligned}
i_\wedge &= \max\{j \leq i \mid a_j + b_j \geq p\}, \\
i_\vee &= \max\{j \leq i \mid a_j + b_j \leq p-2\}, \\
c_i &= \mathbf{1}_{i_\wedge > i_\vee}.
\end{aligned} \tag{5}$$

Alternatively, the carry-over bits can be expressed equivalently as:

$$c_i = \bigvee_{0 \leq j \leq i} \left[\mathbf{1}_{a_j+b_j \geq p} \wedge \bigwedge_{j \leq k \leq i} \mathbf{1}_{a_k+b_k \geq p-2} \right]. \tag{6}$$

In Equation (5), $i_\wedge$ identifies the largest bit index less than or equal to i that contributes a carry to higher bits, while $i_\vee$ identifies the largest bit index less than or equal to i such that the carry generated below $i_\vee$ does not propagate beyond $i_\vee$. Thus, the carry-over bit $c_i = 1$ if and only if $i_\wedge > i_\vee$.

After computing the carry-over bits, the sum of the input integers can be computed as:

$$\begin{aligned}
o_0 &= (a_0 + b_0) \bmod p, \\
o_1 &= (a_1 + b_1 + c_0) \bmod p, \\
&\cdots \\
o_i &= (a_i + b_i + c_{i-1}) \bmod p, \\
o_n &= c_{n-1}.
\end{aligned} \tag{7}$$

Therefore, the output $\boldsymbol{o}$ is exactly the sum of the two input numbers, and Algorithm 1 correctly computes $\text{ADD}_p(\boldsymbol{a}, \boldsymbol{b})$ for all $\boldsymbol{a}, \boldsymbol{b} \in \{0,1\}^n$. □

Next, we provide the proof for Theorem 4.1.

Proof for Theorem 4.1. We now demonstrate that a constant-precision Transformer, with constant depth L, a fixed number of attention heads, and a hidden dimension of size $O(n^2)$, is capable of simulating Algorithm 1. Consequently, this model can accurately produce the correct output for any pair of input integers $\boldsymbol{a}$ and $\boldsymbol{b}$.

Initial Embeddings: The total length of the input sequence is at most $2(n+1)$. We categorize the tokens into two distinct classes: (1) *number tokens* representing digits $(0, 1, \cdots, p-1)$, and (2) *auxiliary tokens* for operations and control flow ("+", "=", `<SOS>`, and `<EOS>`).

The embeddings for each token are initialized as follows:

- **Embedding of input token a_i:**

$$\boldsymbol{u}^0_{a,i} = (a_i \boldsymbol{e}_{i+1}, \mathbf{0}, -1, \mathbf{0}, 0, 1, 1).$$

- **Embedding of input token b_i:**

$$\boldsymbol{u}^0_{b,i} = (\mathbf{0}, b_i \boldsymbol{e}_{i+1}, -1, \mathbf{0}, 0, 2, 1).$$

- **Embedding of output token o_i:**

$$\boldsymbol{u}^0_{o,i} = (\mathbf{0}, \mathbf{0}, o_i, \boldsymbol{e}_{i+1}, 0, 3, -1).$$

- **Embedding of the "+" token:**

$$\boldsymbol{u}^0_{+} = (\mathbf{0}, \mathbf{0}, -1, \mathbf{0}, 0, 4, -1).$$

- **Embedding of the "=" token:**

$$\boldsymbol{u}^0_{=} = (\mathbf{0}, \mathbf{0}, -1, \mathbf{0}, 1, 5, -1).$$

- **Embedding of the <SOS> token:**

$$\boldsymbol{u}^0_{\texttt{<SOS>}} = (\mathbf{0}, \mathbf{0}, -1, \mathbf{0}, 0, 6, -1).$$

- **Embedding of the <EOS> token:**

$$\boldsymbol{u}^0_{\texttt{<EOS>}} = (\mathbf{0}, \mathbf{0}, 0, \mathbf{0}, 0, 3, -1).$$

In each of these embeddings:

- $\boldsymbol{e}_i \in \mathbb{R}^{n+1}$ is a one-hot vector representing the positional encoding of the token (e.g., digit a_i) in the sequence.
- **0** is a vector of zeros of appropriate dimensions.

Block 1. The first block of the Transformer performs the **COPY** operation, which copies the values of a_i, b_i to the positions of the output tokens. This is achieved using the attention mechanism. The query, key, and value are set as follows:

- **Query:** $\boldsymbol{q} = B_s$
- **Key:** $\boldsymbol{k} = \boldsymbol{u}^0[3n + 6]$, i.e., $\boldsymbol{k} = 1$ for input number tokens, and $\boldsymbol{k} = -1$ otherwise.
- **Value:** $\boldsymbol{v} = \boldsymbol{u}^0[1, \cdots, 2n + 2]$, i.e.,

$$\boldsymbol{v} = \begin{cases} (a_i \boldsymbol{e}_{i+1}, \mathbf{0}) & \text{for input } \boldsymbol{a}, \\ (\mathbf{0}, b_i \boldsymbol{e}_{i+1}) & \text{for input } \boldsymbol{b}, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Since we operate under constant precision, we carefully analyze the attention values. The attention value (before normalization) is B_s for tokens a_i, b_i and $-B_s$ otherwise. By Lemma C.8, we know $\exp(B_s) = B_s$ and $\exp(-B_s) = 0$. The normalization term for attention is $2nB_s = B_s$, so the attention weights are 1 for tokens a_i, b_i and 0 otherwise. As a result, the attention output at the positions of the output tokens is always $(a_0, \cdots, a_n, b_0, \cdots, b_n)$.

Block 2. The second block of the Transformer uses MLPs to compute the output $\boldsymbol{o}$ based on Algorithm 1. The calculations proceed in the following steps:

- **Compute $r_i = a_i + b_i$ for $i = 0, \cdots, n$.**

  This can be implemented using an MLP with constant hidden dimension. To avoid overflow of r_i, we require $B_s \geq 2p$.

- **Compute $f_i = \mathbf{1}_{r_i \geq p}$ and $g_i = \mathbf{1}_{r_i \geq p-2}$.**

  Using Lemma C.9, we can calculate:

$$f_i = \frac{\text{GeLU}[B_s \cdot (2r_i - 2p + 1)]}{\text{GeLU}(B_s)}, \quad g_i = \frac{\text{GeLU}[B_s \cdot (2r_i - 2p + 5)]}{\text{GeLU}(B_s)}.$$

  Here, we require $B_s \geq 4p$ to avoid overflow of $2r_i - 2p + 1$.

- **Compute c_i using the formula:**

$$c_i = \bigvee_{0 \leq j \leq i} \left[\mathbf{1}_{a_j + b_j \geq p} \wedge \bigwedge_{j \leq k \leq i} \mathbf{1}_{a_k + b_k \geq p-2} \right] = \bigvee_{0 \leq j \leq i} \left[f_j \wedge \bigwedge_{j \leq k \leq i} g_k \right].$$

  Notice that:

$$\bigvee_{1 \leq i \leq \gamma} \alpha_i = \frac{\text{GeLU}\left[B_s \cdot (\sum_{i=1}^{\gamma} \alpha_i)\right]}{\text{GeLU}(B_s)}, \quad \bigwedge_{1 \leq i \leq \gamma} \alpha_i = 1 - \bigvee_{1 \leq i \leq \gamma} (1 - \alpha_i).$$

  These formulas imply that the $\vee$ and $\wedge$ operations can be implemented using a constant-depth, constant-precision MLP with constant hidden dimension. Therefore, c_i can be computed using $O(n)$ hidden dimension.

- **Compute $o_i = a_i + b_i + c_{i-1}$ for $i = 0, \cdots, n$.**

  This computation can also be implemented with a constant hidden dimension. Again, we require $B_s \geq 2p$ to avoid overflow of o_i.

Since we need to compute r_i, f_i, g_i, c_i for all i, the hidden dimension of this block is $O(n^2)$.

Block 3. This block filters out the token o_i from $\boldsymbol{o}$. Specifically, for the token o_{i+1}, where $i \in \{0, \cdots, n-1\}$, we predict the next token o_i.

First, we calculate the positional embedding $\boldsymbol{e}_{i+1}$ using $\boldsymbol{e}_{i+2}$ from the positional embedding of $\boldsymbol{m}_{o,i+1}^0$. Then, we compute o_i as:

$$o_i = \langle \boldsymbol{e}_{i+1}, \boldsymbol{o} \rangle.$$

Using the property $x = \text{GeLU}(x) - \text{GeLU}(-x)$, this can be expanded as:

$$o_i = \sum_{j=1}^{n+1} \boldsymbol{e}_{i+1}[j]\boldsymbol{o}[j] = \sum_{j=1}^{n+1} \left[\text{GeLU}(\boldsymbol{e}_{i+1}[j] - B_s(2 - 2\boldsymbol{o}[j])) - \text{GeLU}(-\boldsymbol{e}_{i+1}[j] - B_s(2 - 2\boldsymbol{o}[j]))\right].$$

Thus, o_i can be calculated using $O(n)$ hidden dimension. The final output from this layer is given by:

$$\boldsymbol{e}_{o,i+1}^3 = \begin{cases} (o_i, \boldsymbol{e}_{i+1}) & \text{if } i > 0, \\ (0, \mathbf{0}) & \text{if } i = 0. \end{cases}$$

Predict Next Token. Given the output embeddings from the last Transformer layer, $\boldsymbol{e}_{o,i}^3$, and the word embeddings, the Transformer predicts the next token by finding the nearest word embedding.

Precision Requirements. In this construction, we require $B_s \geq 4p$, which guarantees that constant precision is sufficient for all computations. □

D.2 Proof for Theorem 4.2

Theorem 4.2. *Fix integers $p \geq 2$ and $c, L \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any polynomial f, there exist problem scales n and k such that no constant-precision autoregressive Transformer with L layers and hidden dimension $d < f(n,k)$ can correctly solve the* $\text{IterADD}_p(n,k)$ *task.*

Proof. Assume, for the sake of contradiction, that there exist integers $p \geq 2$, L, and a polynomial f, such that for all problem scales n and k, there exists a constant-precision autoregressive Transformer with L layers and hidden dimension $d \leq f(n,k)$ that can solve the $\mathrm{IterADD}_p(n,k)$ task correctly.

We now consider the majority function $\mathrm{Maj}(b_1, \dots, b_k)$, where $b_i \in \{0,1\}$. To establish the contradiction, we construct a reduction from $\mathrm{Maj}(b_1, \dots, b_k)$ to $\mathrm{IterADD}_p(2, k')$, where $k' = p^{\lceil \log_p k \rceil} \leq pk$. Specifically, let $a_i = b_i(p^2 - 1)$ for $i = 1, \dots, k$, and define the remaining terms as follows:

$$a_{k+1} + \dots + a_{k'} = p^{\lceil \log_p k \rceil + 1} - (p^2 - 1)\left\lceil \frac{k}{2} \right\rceil.$$

This construction is feasible because:

$$p^{\lceil \log_p k \rceil + 1} - (p^2 - 1)\left\lceil \frac{k}{2} \right\rceil \leq (p^{\lceil \log_p k \rceil} - k)(p^2 - 1),$$

which holds for $p \geq 2$. Consequently, the following equivalence relationships hold:

$$\mathrm{Maj}(b_1, \dots, b_k) = 1 \iff \sum_{i=1}^{k} b_i \geq \left\lceil \frac{k}{2} \right\rceil \iff \sum_{i=1}^{k'} a_i \geq p^{\lceil \log_p k \rceil + 1} \iff o_{\lceil \log_p k \rceil + 1} > 0,$$

where $o_{\lceil \log_p k \rceil + 1}$ is the output token corresponding to the final layer of the Transformer.

Now, observe that a bounded-depth, fixed-precision decoder-only Transformer with polynomial hidden dimension, which generates a single token, operates within the complexity class AC^0. However, by the reduction above, solving $\mathrm{IterADD}_p(2, k')$ implies the ability to compute Maj. This leads to a contradiction, as $\mathrm{Maj} \notin \mathsf{AC}^0$.

Thus, no constant-precision autoregressive Transformer with L layers and hidden dimension $d \leq f(n,k)$ can solve the $\mathrm{IterADD}_p(n,k)$ task in general. □

D.3 Proof for Theorem 4.3

Theorem 4.3. *Fix integers $p \geq 2$ and $c, L \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any polynomial f, there exist problem scales n and l such that no constant-precision autoregressive Transformer with L layers and hidden dimension $d < f(n,l)$ can correctly solve the $\mathrm{MUL}_p(n,l)$ task.*

Proof. Assume, for contradiction, that there exist integers $p \geq 2$, L, and a polynomial f, such that for all problem scales n and l, there exists a constant-precision autoregressive Transformer with L layers and hidden dimension $d \leq f(n,l)$ that can correctly solve the $\mathrm{MUL}_p(n,l)$ task.

Now, consider the majority function $\mathrm{Maj}(c_1, \dots, c_k)$, where $c_i \in \{0,1\}$. We construct a reduction from $\mathrm{Maj}(c_1, \dots, c_k)$ to $\mathrm{MUL}_p(n,l)$. Specifically, let

$$n = \left(\lceil \log_p k \rceil + 1\right)\left(p^{\lceil \log_p k \rceil} + \left\lfloor \frac{k}{2} \right\rfloor\right) = O(k \log k), \quad l = n + \lceil \log_p k \rceil = O(k \log k).$$

We extend c_i by defining

$$k' = p^{\lceil \log_p k \rceil} + \left\lfloor \frac{k}{2} \right\rfloor, \quad c_{k+1} = \dots = c_{k'} = 1,$$

and construct the sequences $\boldsymbol{a}$ and $\boldsymbol{b}$ as follows:

$$\boldsymbol{a} = c_1 \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil} c_2 \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil} \cdots c_{k'} \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil}, \quad \boldsymbol{b} = 1 \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil} 1 \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil} \cdots 1 \underbrace{0 \cdots 0}_{\lceil \log_p n \rceil}.$$

Under this construction, the following equivalences hold:

$$\mathrm{Maj}(c_1, \dots, c_k) = 1 \iff c_1 + \dots + c_k \geq \left\lceil \frac{k}{2} \right\rceil \iff c_1 + \dots + c_{k'} \geq p^{\lceil \log_p k \rceil} \iff o_{l-1} > 0,$$

where o_{l-1} denotes the first output token.

Now, observe that a bounded-depth, fixed-precision decoder-only Transformer with polynomial hidden dimension, which generates a single token, operates within the complexity class AC^0. However, by the reduction above, solving $\mathrm{MUL}_p(n,l)$ implies the ability to compute Maj. This leads to a contradiction, as $\mathrm{Maj} \notin \mathsf{AC}^0$.

Hence, no constant-precision autoregressive Transformer with L layers and hidden dimension $d \leq f(n,l)$ can correctly solve $\mathrm{MUL}_p(n,l)$ task for any problem scale n and l. □

E Proofs for Section 5

In this section, we provide the formal proofs of the theorems stated in Section 5. Before proceeding with the proofs, we revisit the role of the tokenizer $\boldsymbol{T}_c$. As established in Appendix B.4 and Proposition B.6, it is sufficient to focus on the case of $\boldsymbol{T}_1$, where both the input and output sequences are tokenized into single digits. This simplification is crucial for the subsequent analysis and constructions. Notably, this reasoning parallels the argument presented at the beginning of Appendix D (Proof of Section 4).

E.1 Proof for Theorem 5.1

Theorem 5.1. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. There exists a logarithmic-precision Transformer with constant depth and hidden dimension (independent of n) that can generate the correct output for any input on the $\mathrm{ADD}_p(n)$ task.*

Proof. The result in Theorem 5.1 follows as a special case of Theorem 5.2. Specifically, by setting $k = 2$ in Theorem 5.2, the proof is complete. Observe that in this case, $m = \lceil \log_p k \rceil = 1$, which implies that the combination of neighboring bits is unnecessary. □

E.2 Proof for Theorem 5.2

Theorem 5.2. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any integers n and k, there exists a logarithmic-precision Transformer with constant depth and hidden dimension d (independent of n and k) that can generate the correct output for any input on the $\mathrm{IterADD}_p(n,k)$ task.*

For ease of understanding, we first present an algorithm to compute $\mathrm{IterADD}_p(n,k)$ (Algorithm 2) and prove its correctness. Subsequently, we demonstrate the construction of a constant-size Transformer with logarithmic precision to simulate Algorithm 2.

Lemma E.1 (Algorithm for $\mathrm{IterADD}_p(n,k)$)**.** *Algorithm 2 computes $\boldsymbol{o} = \boldsymbol{a}_1 + \cdots + \boldsymbol{a}_k$ for any inputs $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$.*

Proof. The initial four steps of the algorithm transform p-adic addition into p^m-adic addition. This transformation allows the sum of k numbers to be represented as $\sum_i s_i p^{im}$, where s_i are intermediate coefficients.

At this stage, $s_i \in [0, kp^m)$. To ensure the final results are accurate, we must account for carry-over effects such that the outputs $\tilde{o}_i$ remain within the range $[0, p^m - 1]$. Each s_i can be decomposed as $s_i = b_i p^m + q_i$, where $q_i \in [0, p^m - 1]$ and $b_i < k \leq p^m$. Consequently, the overflow b_i propagates only *directly* to the next subsequent digit q_{i+1}. Notably, $q_{i+1} + b_i \leq 2(p^m - 1)$.

Let $\boldsymbol{c}$ denote the vector recording carry-over effects at each position i. The carry-over can be computed iteratively as:

$$\begin{aligned}
c_{-1} &= 0, \\
c_0 &= \mathbf{1}_{q_0 + b_{-1} \geq p^m} \ (b_{-1} := 0), \\
c_1 &= \left(c_0 \cdot \mathbf{1}_{q_1 + b_0 \geq p^m - 1}\right) \vee \mathbf{1}_{q_1 + b_0 \geq p^m}, \\
&\cdots \\
c_i &= \left(c_{i-1} \cdot \mathbf{1}_{q_i + b_{i-1} \geq p^m - 1}\right) \vee \mathbf{1}_{q_i + b_{i-1} \geq p^m}.
\end{aligned} \tag{8}$$

Algorithm 2: $\mathrm{IterADD}_p(n, k)$ Algorithm

```
Input  : $k$ $p$-adic numbers $\boldsymbol{a}_1, \cdots, \boldsymbol{a}_k$, each of maximum length $n$
Output : The sum of the inputs $\boldsymbol{o}$
1  $m = \lceil \log_p k \rceil$;
2  Compute the sum of each bit: $r_j = \sum_{i \in [k]} a_{ij}$ for $j = 0, \cdots, n-1$;
3  Combine neighboring $m$ bits:
       $$s_i = \sum_{j=0}^{m-1} r_{ik+j} p^j$$
   for $i = 0, \cdots, \lfloor n/m \rfloor$;
4  Decompose $s_i$: $s_i = b_i p^m + q_i$, where $q_i \in [0, p^m - 1]$ and $b_i, q_i \in \mathbb{N}$;
5  Initialize $c_0 = 0$;
6  foreach $i = 0, \cdots, \lfloor n/m \rfloor$ do
7      Compute carry bits $\boldsymbol{c}$:
8      $i_\wedge = \max\{j \le i \mid q_j + b_{j-1} \ge p^m\}$;
9      $i_\vee = \max\{j \le i \mid q_j + b_{j-1} \le p^m - 2\}$;
10     $c_i = \mathbf{1}_{i_\wedge > i_\vee}$;
11 end
12 Compute the $p^m$-adic outcome $\tilde{\boldsymbol{o}}$: $\tilde{o}_i = (q_i + b_{i-1} + c_{i-1}) \bmod p^m$ for $i = 0, \cdots, \lfloor n/m \rfloor + 1$;
13 Covert $p^m$-adic $\tilde{\boldsymbol{o}}$ to $p$-adic $\boldsymbol{o}$:
       $$o_i = \left\lfloor \frac{\tilde{o}_j \bmod p^{(l+1)}}{p^l} \right\rfloor$$
   for $i = jk + l$, where $l \in \{0, \cdots, k-1\}$, $j \in \mathbb{Z}$;
```

To avoid recursive computation, the carry-over can also be derived using:

$$\begin{aligned} i_\wedge &= \max\{j \le i \mid q_j + b_{j-1} \ge p^m\}, \\ i_\vee &= \max\{j \le i \mid q_j + b_{j-1} \le p^m - 2\}, \\ c_i &= \mathbf{1}_{i_\wedge > i_\vee}. \end{aligned} \tag{9}$$

Alternatively, this can be expressed as:

$$c_i = \bigvee_{0 \le j \le i} \left[\mathbf{1}_{q_j + b_{j-1} \ge p^m} \wedge \bigwedge_{j \le k \le i} \mathbf{1}_{q_k + b_{k-1} \ge p^m - 2} \right]. \tag{10}$$

Here, $i_\wedge$ identifies the highest bit contributing a carry to the i-th position, while $i_\vee$ identifies the highest bit below i such that carry propagation from bits below $i_\vee$ does not affect higher bits. Thus, $c_i = 1$ if and only if $i_\wedge > i_\vee$.

Once the carry-over vector $\boldsymbol{c}$ is determined, the p^m-adic sum can be computed as:

$$\begin{aligned} &\tilde{o}_0 = q_0, \\ &\tilde{o}_1 = (q_1 + b_0 + c_0) \bmod p^m, \\ &\cdots \\ &\tilde{o}_i = (q_i + b_{i-1} + c_{i-1}) \bmod p^m. \end{aligned} \tag{11}$$

Finally, to convert the p^m-adic representation back to a p-adic number $\tilde{\boldsymbol{o}}$, we perform the following modulus operation:

$$o_i = \left\lfloor \frac{\tilde{o}_j \bmod p^{(l+1)}}{p^l} \right\rfloor,$$

for $i = jk + l$, where $l \in \{0, \ldots, k-1\}$ and $j \in \mathbb{Z}$.

Therefore, the output $\boldsymbol{o}$ is precisely the sum of the k input p-adic numbers, and the algorithm in Algorithm 2 correctly computes $\text{IterADD}_p(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k)$ for all $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$. $\square$

Now we present the proof of Theorem 5.2.

Proof of Theorem 5.2. We demonstrate that a log-precision transformer, with constant depth, a fixed number of attention heads, and constant embedding dimensions, is capable of simulating Algorithm 2. As a result, this model can reliably produce the correct output for any input integers $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$.

Initial Embeddings: The total length of the input sequence is at most $k(n+1)$. Tokens in the sequence are divided into two categories: numeric tokens $(0, 1, \ldots, p-1)$ and auxiliary tokens ($+$, $=$, <SOS>, <EOS>). Given the parameters k and n, we define the parameter $m = \lceil \log_p k \rceil$, as specified in Algorithm 2. The embeddings for these tokens are constructed as follows:

- **Embedding of input token $a_{i,j}$:**

$$\boldsymbol{e}^0_{i,j} = \left(a_{i,j}, 0, 0, i, j, j \bmod m, \lfloor \frac{j}{m} \rfloor, p^{j \bmod m}, p^{-(j \bmod m)}, \text{ape}_{i,j}\right).$$

- **Embedding of the i-th "$+$" token:**

$$\boldsymbol{e}^0_{i,+} = \left(0, 1, 0, i, -1, -1, -1, 0, 0, \text{ape}_{i,+}\right).$$

- **Embedding of the "$=$" token:**

$$\boldsymbol{e}^0_{=} = \left(0, 1, 0, k+1, -1, -1, -1, 0, 0, \text{ape}_{=}\right).$$

- **Embedding of the <SOS> token:**

$$\boldsymbol{e}^0_{\texttt{<SOS>}} = \left(0, 1, 0, 0, -1, -1, -1, 0, 0, \text{ape}_{\texttt{<SOS>}}\right).$$

- **Embedding of the <EOS> token:**

$$\boldsymbol{e}^0_{\texttt{<EOS>}} = \left(0, 0, 1, 0, -1, -1, -1, 0, 0, \text{ape}_{\texttt{<EOS>}}\right).$$

- **Embedding of output token o_i:**

$$\boldsymbol{e}^0_{o,i} = \left(o_i, 0, 0, 0, i, i \bmod m, \lfloor \frac{i}{m} \rfloor, p^{i \bmod m}, p^{-(i \bmod m)}, \text{ape}_{o,i}\right).$$

Here, $\text{ape}_{\ldots}$ represents the absolute positional encoding. In this construction, the first three dimensions of each embedding correspond to the word embedding, while the remaining six dimensions capture the positional embedding.

Block 1. The first block of the Transformer computes the following quantities:

1. $l_{i,j}$: The number of preceding tokens (inclusive) $a_{i',j'}$ satisfying $i' = i$ and $\lfloor \frac{j'}{m} \rfloor = \lfloor \frac{j}{m} \rfloor$. The value $l_{i,j}$ is defined only for input number tokens $a_{i,j}$. If undefined, we set $l = -1$.

2. $f_{i,j}$: Defined as $f_{i,j} = 1$ if no preceding tokens (exclusive) $a_{i',j'}$ exist such that $\lfloor \frac{j'}{m} \rfloor = \lfloor \frac{j}{m} \rfloor$; otherwise, $f_{i,j} = 0$. This value is defined only for input number tokens $a_{i,j}$, and if undefined, we set $f = -1$.

To compute $l_{i,j}$, we define the query, key, value, and r in Appendix C.2 as follows:

- Query: $\boldsymbol{q}_{i,j} = \left(-1, 2i, -i^2, -1, 2\lfloor \frac{j}{m} \rfloor, -\lfloor \frac{j}{m} \rfloor^2\right)$.

- Key: $\boldsymbol{k}_{i',j'} = \left(i'^2, i', 1, \lfloor\frac{j'}{m}\rfloor^2, \lfloor\frac{j'}{m}\rfloor, 1\right)$.
- Value: $\boldsymbol{v}_{i',j'} = (\text{ape}_{i',j'})$.
- r: $r_{i',j'} = -\text{ape}_{i',j'}$.

Using Lemma C.1, the components of the query and key can be computed by preceding MLP layers. The result of the dot product is given by:

$$\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = -\left(\lfloor\frac{j'}{m}\rfloor - \lfloor\frac{j}{m}\rfloor\right)^2 - (i'-i)^2.$$

This implies that $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = 0$ if $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor$ and $i = i'$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle \leq -1$ otherwise. By Lemma C.5, one attention head can be used to copy the absolute position j'' of the first token satisfying these conditions. The value of $l_{i,j}$ is then computed as $j - j'' + 1$.

To compute $f_{i,j}$, we redefine the query, key, and r as follows:

- Query: $\boldsymbol{q}_{i,j} = \left(-1, 2\lfloor\frac{j}{m}\rfloor, -\lfloor\frac{j}{m}\rfloor^2\right)$.
- Key: $\boldsymbol{k}_{i',j'} = \left(\lfloor\frac{j'}{m}\rfloor^2, \lfloor\frac{j'}{m}\rfloor, 1\right)$.
- Value: $\boldsymbol{v}_{i',j'} = (\text{ape}_{i',j'})$.
- r: $r_{i',j'} = -\text{ape}_{i',j'}$.

In this case, the dot product is given by:

$$\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = -\left(\lfloor\frac{j'}{m}\rfloor - \lfloor\frac{j}{m}\rfloor\right)^2.$$

This yields $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = 0$ if $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle \leq -1$ otherwise. By Lemma C.5, one attention head can copy the absolute position j'' of the first token satisfying these conditions. The value $f_{i,j}$ is then determined by checking whether $j'' = j$. Specifically, we evaluate:

$$\mathbf{1}_{j''=j} = \text{ReLU}[1 - (j - j'')^2],$$

which allows $f_{i,j}$ to be computed by a constant-width MLP via Lemma C.1.

Finally, for undefined values, l and f are set to -1 in the MLP stage using conditional selection (Lemma C.3) based on positional embedding information. In summary, the embeddings generated in this block are $\boldsymbol{e}^1 = (l, f)$, and these embeddings are concatenated with the original input embeddings.

Block 2. The second block of the Transformer is designed to compute the first three lines of Algorithm 2. Specifically, this block utilizes the attention mechanism to aggregate the adjacent m bits and derive s_i. For each token $a_{i,j}$, let $t_{i,j}$ denote the number of preceding tokens (including $a_{i,j}$ itself) $a_{i',j'}$ such that $\lfloor\frac{j}{m}\rfloor = \lfloor\frac{j'}{m}\rfloor$. This block computes the following values:

1. $\frac{1}{t_{i,j}}$, where $t_{i,j}$ is as defined above. If $t_{i,j}$ is undefined, its value is set to -1.
2. $c_{i,j}$, the mean value computed over $a_{i',j'}p^{j' \bmod m}$ for previous tokens (including $a_{i,j}$) $a_{i',j'}$, where $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor$. If this value is undefined, it is also set to -1.

Using these, we derive $s_w = c_{i,mw}t_{i,mw}$, where i is the largest index such that the length of $\boldsymbol{a}_i$ exceeds mk.

To compute the first value, the query, key, and value vectors are defined as follows:

- Query: $\boldsymbol{q}_{i,j} = \left(-1, 2\lfloor\frac{j}{m}\rfloor, -\lfloor\frac{j}{m}\rfloor^2\right)$.

- Key: $\boldsymbol{k}_{i',j'} = \left(\lfloor\frac{j'}{m}\rfloor^2, \lfloor\frac{j'}{m}\rfloor, 1\right)$.
- Value: $\boldsymbol{v}_{i',j'} = (f_{i',j'})$.

Using Lemma C.1, the components of the query and key can be computed by preceding MLP layers. The result of the dot product is given by: $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = -\left(\lfloor\frac{j'}{m}\rfloor - \lfloor\frac{j}{m}\rfloor\right)^2$. This result implies that $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = 0$ when $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle \leq -1$ otherwise. By the definition of $f_{i,j}$ and Lemma C.6, the output of the attention mechanism is $\frac{1}{t_{i,j}}$, as required.

To compute the second value, we redefine the query, key, and value vectors as follows:

- Query: $\boldsymbol{q}_{i,j} = \left(-1, 2\lfloor\frac{j}{m}\rfloor, -\lfloor\frac{j}{m}\rfloor^2\right)$.
- Key: $\boldsymbol{k}_{i',j'} = \left(\lfloor\frac{j'}{m}\rfloor^2, \lfloor\frac{j'}{m}\rfloor, 1\right)$.
- Value: $\boldsymbol{v}_{i',j'} = (a_{i',j'} p^{j' \bmod m})$.

Similar to the computation of the first value, we use Lemma C.1 to compute the components of the query and key. In this case, the dot product is given by: $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = -\left(\lfloor\frac{j'}{m}\rfloor - \lfloor\frac{j}{m}\rfloor\right)^2$. Thus, $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = 0$ when $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle \leq -1$ otherwise. By applying Lemma C.6, the attention output is $c_{i,j}$, as required.

Finally, for undefined values, we assign -1 during the MLP stage by employing conditional selection, as outlined in Lemma C.3, utilizing information encoded in the positional embeddings.

In summary, the new embeddings generated in this block can be expressed as $\boldsymbol{e}^2 = \left(\frac{1}{t}, c\right)$. These embeddings are subsequently concatenated with the original embeddings.

Block 3. The third block of the Transformer computes the value of $c_{i,j}t_{i,j}$. This is achieved by first determining $t_{i,j}$ via the attention layer and $\frac{1}{t_{i,j}}$ from the previous block. Subsequently, $c_{i,j}t_{i,j}$ is computed using Lemma C.1.

Notice that $t_{i,j}$ does not exceed the absolute positional value of the current token. We define the query, key, and value vectors as follows:

- **Query:** $\boldsymbol{q}_{i,j} = \left(\frac{1}{t_{i,j}^2}, -\frac{2}{t_{i,j}}, 1\right)$
- **Key:** $\boldsymbol{k}_{i',j'} = \left(\mathrm{ape}_{i',j'}^2, \mathrm{ape}_{i',j'}, 1\right)$
- **Value:** $\boldsymbol{v}_{i',j'} = (\mathrm{ape}_{i',j'})$

These vectors can be constructed using Lemma C.1. It follows that the inner product

$$\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = -\left(\frac{\mathrm{ape}_{i',j'}}{t_{i,j}} - 1\right)^2,$$

which implies $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle = 0$ if $\mathrm{ape}_{i',j'} = t_{i,j}$ and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'}\rangle \leq -\frac{1}{n^2k^2}$ otherwise, given that $t_{i,j} \leq nk$. By leveraging Lemma C.6, the attention output is confirmed to be $t_{i,j}$, as required.

Finally, the computation of $c_{i,j}t_{i,j}$ is performed via the subsequent MLP layer. In summary, the embeddings generated in this block are represented as $\boldsymbol{e}^3 = (ct)$. These new embeddings are concatenated with the original embeddings to produce the final output of this block.

Block 4. This block of the Transformer corresponds to the fourth step in Algorithm 2, which decomposes $c_{i,j}t_{i,j}$ as $b_{i,j}p^m + q_{i,j}$. It is important to observe that $b_{i,j} \leq i$, meaning that $b_{i,j}$ does not exceed the absolute positional index of the current token. To achieve this decomposition, we define the query, key, and value as follows:

- **Query:** $\boldsymbol{q}_{i,j} = (-(c_{i,j}t_{i,j} + \frac{1}{2})^2, 2p^m(c_{i,j}t_{i,j} + \frac{1}{2}), -p^{2m})$

- **Key:** $\boldsymbol{k}_{i',j'} = \left(1, \text{ape}_{i',j'} - \frac{1}{2}, (\text{ape}_{i',j'} - \frac{1}{2})^2\right)$
- **Value:** $\boldsymbol{v}_{i',j'} = \text{ape}_{i',j'}$

The above components can be computed using Lemma C.1. Consequently, the inner product of the query and key is given by:

$$\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = -\left[c_{i,j}t_{i,j} - \left(\text{ape}_{i',j'} - \frac{1}{2}\right)p^m + \frac{1}{2}\right]^2.$$

This expression implies that $|\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle| \leq \left(\frac{p^m-1}{2}\right)^2$ if $\text{ape}_{i',j'} = \lfloor \frac{c_{i,j}t_{i,j}}{p^m} \rfloor$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle \leq -\left(\frac{p^m+1}{2}\right)^2$ otherwise. By applying Lemma C.7, we obtain:

$$c = \frac{(p^m+1)^2 - (p^m-1)^2}{(p^m-1)^2} \geq \frac{4}{p^m}.$$

From this, it follows that $1/c = O(p^m) = O(k)$. Hence, we can design the query, key, and value such that the attention output satisfies $\lfloor \frac{c_{i,j}t_{i,j}}{p^m} \rfloor = b_{i,j}$. Finally, $q_{i,j}$ can be computed as $q_{i,j} = c_{i,j}t_{i,j} - p^m b_{i,j}$ using the subsequent MLP. The embeddings generated by this block are thus given by $\boldsymbol{e}^4 = (b, q)$.

Block 5. This block of the Transformer computes $q_{w+1} + b_w$ for s_w. Recall that $s_w = c_{i,mw}t_{i,mw}$, where i is the largest index such that the length of $\boldsymbol{a}_i$ is greater than mw. The goal is to compute these values at their corresponding positions.

First, we use the attention mechanism to copy q_{w+1} for the token $\boldsymbol{a}_i$ defined above. Note that it is always possible to retrieve the correct value because the position associated with the correct value of s_{w+1} precedes that of s_k. To achieve this, we utilize the attention mechanism to copy from the position containing the value s_{w+1}. This is implemented by appropriately configuring the query, key, value, and r as described in Appendix C.2:

- Query: $\boldsymbol{q}_{i,j} = \left(-1, 2\lfloor\frac{j}{m}\rfloor, -\lfloor\frac{j}{m}\rfloor^2, -1\right)$.
- Key: $\boldsymbol{k}_{i',j'} = \left((\lfloor\frac{j'}{m}\rfloor - 1)^2, \lfloor\frac{j'}{m}\rfloor - 1, 1, (j' \bmod m)^2\right)$.
- Value: $\boldsymbol{v}_{i',j'} = (q_{i',j'}, \lfloor\frac{j'}{m}\rfloor)$.
- r: $r_{i',j'} = \text{ape}_{i',j'}$.

The values required for the query or key can be computed using previous MLPs, as shown in Lemma C.1. The dot product $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle$ evaluates to

$$\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = -\left(\lfloor\frac{j'}{m}\rfloor - \lfloor\frac{j}{m}\rfloor - 1\right)^2 - (j' \bmod m)^2.$$

This implies $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle = 0$ if $\lfloor\frac{j'}{m}\rfloor = \lfloor\frac{j}{m}\rfloor + 1$ and $j' \bmod m = 0$, and $\langle \boldsymbol{q}_{i,j}, \boldsymbol{k}_{i',j'} \rangle \leq -1$ otherwise.

Using Lemma C.5, one attention head suffices to copy the values $q_{i',j'}$ and $\lfloor\frac{j'}{m}\rfloor$ from the last token satisfying the conditions. Consequently, the first dimension of the attention output equals q_{w+1} if an input number with length greater than $m(w+1)$ exists, as required. Otherwise, q_{w+1} should be zero, and the attention output remains undefined. These two cases can be distinguished by inspecting the second dimension of the attention output: if no input number has a length greater than $m(w+1)$, the second dimension of the attention output is at most $\lfloor\frac{j}{m}\rfloor$. Using Lemma C.3, we can identify these cases and set $q_{w+1} = 0$ when necessary.

Finally, a subsequent MLP computes the correct value of $q_{w+1} + b_w$ for s_w. Additionally, this MLP calculates the indicators $\mathbf{1}_{q_{w+1}+b_w \geq p^m}$, $\mathbf{1}_{q_{w+1}+b_w \leq p^m-2}$, $\mathbf{1}_{b_w \geq p^m}$, and $\mathbf{1}_{b_w \leq p^m-2}$ using the formulation:

$$\mathbf{1}_{q_{w+1}+b_w \geq p^m} = \text{ReLU}[q_{w+1} + b_w - (p^m - 1)] - \text{ReLU}[q_{w+1} + b_w - p^m],$$

as described in Lemma C.2.

To summarize, the embeddings generated in this block are as follows:

- For positions with the correct value of s_w:

$$e^5 = (q_{w+1} + b_w, b_w, \mathbf{1}_{q_{w+1}+b_w \geq p^m}, \mathbf{1}_{q_{w+1}+b_w \leq p^m-2}, \mathbf{1}_{b_w \geq p^m}, \mathbf{1}_{b_w \leq p^m-2}, w).$$

- For all other positions:

$$e^5 = (-1, -1, -1, -1, -1, -1, -1).$$

  This can be achieved by filtering out infeasible values using Lemma C.3.

Block 6. This block of the Transformer computes the following values for positions with the correct value of s_w:

- The smallest $w_1 \geq w$ such that $\mathbf{1}_{q_{w+1}+b_w \geq p^m} = 1$.
- The smallest $w_2 \geq w$ such that $\mathbf{1}_{q_{w+1}+b_w \leq p^m-2} = 1$.

Both calculations rely on the standard COPY operation, which can be implemented using Lemma C.5. To ensure the validity of w_1 and w_2 (i.e., the existence of such indices), we COPY the values $\mathbf{1}_{q_{w_1+1}+b_{w_1} \geq p^m}$ and $\mathbf{1}_{q_{w_2+1}+b_{w_2} \leq p^m-2}$, verifying that they equal 1. Invalid values can then be filtered out using an MLP, as described in Lemma C.3.

The embeddings generated in this block are as follows:

- For positions with the correct value of s_w: $e^6 = (w_1, w_2)$.
- For all other positions: $e^6 = (-1, -1)$. (This can be achieved by filtering infeasible values using Lemma C.3.)

Block 7. The last block of the Transformer executes the final four steps of Algorithm 2. This layer calculates the carry-over bits $\boldsymbol{c}$ and p^m-adic representation of the final output $\boldsymbol{o}$ via the attention mechanism and the MLP, subsequently converting the p^m-adic number into a p-adic number.

The computation of carry-on bits, as described in Equation (9) within Algorithm 2, adheres to the following equations:

$$\begin{aligned} i_\wedge &= \max\{w \leq i \mid q_w + b_{w-1} \geq p^m\}, \\ i_\vee &= \max\{w \leq i \mid q_w + b_{w-1} \leq p^m - 2\}, \\ c_i &= \mathbf{1}_{i_\wedge > i_\vee}. \end{aligned} \tag{12}$$

In the attention layer, operations are restricted to output tokens and other tokens will maintain the embeddings via the residual connection and the filter operation by MLP. Let's consider the token $o_{(i+1)m+j+1}$, where $j \in \{0, \cdots, m-1\}$, we want to predict the next token $o_{(i+1)k+j}$. The model executes the COPY operation, duplicating the previous embeddings to extract $q_{i+1} + b_i$, $i_\wedge$, and $i_\vee$. The extraction is similar to previous blocks, but here we only need to focus on positions with correct value of s_w. To find out the value of $i_\wedge, i_\vee$, we first COPY the embedding of the position with the correct value of s_i, and find the minimum w' which shares the same value of w_1, w_2 with s_i. Again, this can be implemented by several COPY operation with Lemma C.5.

The carry-over bit c_i and the p^m-adic results $\tilde{o}_{i+1}$ are then computed as follows:

$$c_i = \mathbf{1}_{i_\wedge > i_\vee}, \ \tilde{o}_{i+1} = b_i + c_i + q_{i+1}.$$

This computation is facilitated by a constant-size MLP. Subsequently, for the output token $\tilde{o}_{(i+1)k+j}$, the result $o_{(i+1)k+j} = \tilde{o}_{i+1} \bmod p^{j+1}$ is required. We first calculate $\tilde{o}_{i+1}/p^{j+1}$ using the positional embedding and Lemma C.1, then calculate $\lfloor \tilde{o}_{i+1}/p^{j+1} \rfloor$ using the similar fashion to what we did in Block 4, and then calculate $\tilde{o}_{i+1} \bmod p^{j+1}$ using MLP. Finally, we can get the value of $\lfloor \frac{\tilde{o}_{i+1} \bmod p^{j+1}}{p^j} \rfloor$ using the similar fashion to what we did in Block 4.

Upon outputting the token o_0, the model anticipates the <EOS> token, employing an MLP to filter the hidden embeddings and output the word embedding for <EOS>. Thus, the final output from this layer is characterized by the equation:

$$\boldsymbol{e}_{o,i}^7 = \begin{cases} (o_{i-1}, i, 0) & \text{if } i > 0, \\ (-1, -1, 1) & \text{if } i = 0. \end{cases}$$

Predict Next Token. Given the output embeddings of the last transformer layer $\boldsymbol{e}_{o,i}^7$, and the word embeddings, the transformer can simply predict the next token by finding the nearest word embeddings.

In this construction, the norm of the parameters is bounded by $poly(n, k)$, therefore, this construction can be implemented by a log-precision transformer with arbitrarily small error. □

E.3 Proof for Theorem 5.3

Theorem 5.3. *Fix integers $p \geq 2$ and $c \in \mathbb{N}^*$. Consider the tokenizer $\boldsymbol{T}_c$ defined in Eq. (1) for processing the input and output sequences. For any integers n and $l \leq 2n$, there exists a logarithmic-precision Transformer with constant depth (independent of n and k) and hidden dimensions $O(n^2)$ that can generate the correct output for any input on the $\mathrm{MUL}_p(n, l)$ task.*

Here, we first describe an algorithm to perform $\mathrm{MUL}_p(n, l)$ (Algorithm 3) and prove the correctness of Algorithm 3. Then, we construct a Transformer with the configurations in Theorem 5.3 capable for simulating Algorithm 3.

Lemma E.2 (An algorithm to perform $\mathrm{MUL}_p(n, l)$)**.** *Algorithm 3 outputs $\boldsymbol{o} = \boldsymbol{ab} \bmod p^l$ for all inputs $\boldsymbol{a}, \boldsymbol{b}$.*

Proof. It's easy to verify $\sum_i s_i p^{im}$ accurately represents the product of $\boldsymbol{a}, \boldsymbol{b}$. For the subsequent steps, the proof is the same as that of Lemma E.1 since they share the same procedures. □

Next, we provide the proof for Theorem 5.3.

Proof for Theorem 5.3. Now, we demonstrate that a log-precision transformer, with a constant depth, a fixed number of attention heads, and $O(n^2)$ embedding dimensions, is capable of simulating Algorithm 3. Consequently, this model can accurately generate correct output for any input integers $\boldsymbol{a}, \boldsymbol{b}$.

Initial Embeddings: The total length of the input sequence is no longer than $2(n+1)$. We categorize the tokens into two classes: number tokens $(0, 1, \cdots, p-1)$ and auxiliary tokens ($+$, $=$, <SOS> and <EOS>). Given the parameters k, n, we determine the parameter $m = \lceil \log_p k \rceil + 1 \geq 2$, as specified in Algorithm 3. The embeddings for these classes are defined as follows:

- **Embedding of input token a_i:** $\boldsymbol{u}_{a,i}^0 = \big(a_i \boldsymbol{e}_{i+1}, 0, -1, -1, 0, 1, i, 0, \mathrm{ape}_{a,i}\big)$.
- **Embedding of input token b_i:** $\boldsymbol{u}_{b,i}^0 = \big(0, b_i \boldsymbol{e}_{i+1}, -1, -1, 0, 2, i, 0, \mathrm{ape}_{b,i}\big)$.
- **Embedding of the "$\times$" token:** $\boldsymbol{u}_{\times}^0 = (-1, -1, -1, -1, -1, 4, -1, 0, \mathrm{ape}_{\times})$.
- **Embedding of the "$=$" token:** $\boldsymbol{u}_{=}^0 = (-1, -1, -1, -1, -1, 5, -1, 0, \mathrm{ape}_{=})$.
- **Embedding of the <SOS> token:** $\boldsymbol{u}_{\texttt{<SOS>}}^0 = (-1, -1, -1, -1, -1, 6, -1, 0, \mathrm{ape}_{\texttt{<SOS>}})$.
- **Embedding of the <EOS> token:** $\boldsymbol{u}_{\texttt{<EOS>}}^0 = (-1, -1, -1, -1, -1, 7, -1, 0, \mathrm{ape}_{\texttt{<EOS>}})$.
- **Embedding of output token o_i:** $\boldsymbol{u}_{o,i}^0 = (-1, -1, o_i, \boldsymbol{e}_{\lfloor i/m \rfloor}, -1, 3, i, p^{-(i \bmod m)}, \mathrm{ape}_{o,i})$.

where $\boldsymbol{e}_i \in \mathbb{R}^n$ is one-hot vector, and $\mathrm{ape}_{\dots}$ is absolute positional embedding. In this construction, the first $3n+3$ dimensions of each initial embedding represent the word embedding, while the last three dimensions accounts for the position embedding.

Block 1. The first block of the Transformer executes the first three lines of Algorithm 3. To be specific, we first aggregate the input number $\boldsymbol{a}, \boldsymbol{b}$ to the positions of b_0, and then calculate the values of r_j.

To aggregate the input number $\boldsymbol{a}, \boldsymbol{b}$ to the positions of b_0, we set the query, key and value as follows:

Algorithm 3: $\mathrm{MUL}_p(n,l)$ Algorithm

```
Input   : Two $p$-adic numbers $\boldsymbol{a}, \boldsymbol{b}$ no longer than $n$ bits, truncating length $l$
Output : $\boldsymbol{o} := \boldsymbol{ab} \bmod p^l$
1  $m = \lceil \log_p n \rceil + 1$;
2  Compute the product of each pair of bits: $d_{i,j} = a_i b_j$;
3  Compute each bit as
       $$r_j = \sum_{k=\max(0,j-(n-1))}^{\min(n-1,j)} d_{k,j-k}$$
   for $j = 0, \cdots, 2n-1$;
4  Combine neighboring $m$ bits:
       $$s_i = \sum_{j=0}^{m-1} r_{ik+j} p^j$$
   for $i = 0, \cdots, \lfloor (2n-1)/m \rfloor$;
5  Decompose $s_i$ by $s_i = b_i p^m + q_i$, where $q_i \in [0, p^m - 1]$ and $b_i, q_i \in \mathbb{N}$;
6  $b_{-1} = 0$;
7  foreach $i = 0, \cdots, \lfloor (2n-1)/m \rfloor$ do
8      $f_i = \mathbf{1}_{q_i + b_{i-1} \geq p^m}$;
9      $g_i = \mathbf{1}_{q_i + b_{i-1} \geq p^m - 2}$;
10 end
11 Compute the carry-on bits $\boldsymbol{c}$:
       $$c_i = \bigvee_{0 \leq j \leq i} \left( f_j \wedge \bigwedge_{j \leq k \leq i} g_k \right)$$
   for $i = 0, \cdots, \lfloor (2n-1)/m \rfloor$;
12 Compute the $p^m$-adic outcome $\tilde{\boldsymbol{o}}$: $\tilde{o}_i = (q_i + b_{i-1} + c_{i-1}) \bmod p^m$ for $i = 0, \cdots, \lfloor (2n-1)/m \rfloor$;
13 Covert $p^m$-adic $\tilde{\boldsymbol{o}}$ to $p$-adic $\boldsymbol{o}$:
       $$o_i = \left\lfloor \frac{\tilde{o}_j \bmod p^{(l+1)}}{p^l} \right\rfloor$$
   for $i = jk + l$ where $l \in \{0, \cdots, k-1\}$, $j \in \mathbb{Z}$;
```

- Query: $\boldsymbol{q} = (\boldsymbol{e}^0[2n+2])$, i.e., $\boldsymbol{q} = (0)$ for input number $\boldsymbol{a}, \boldsymbol{b}$, and $\boldsymbol{q} = (-1)$ otherwise.
- Key: $\boldsymbol{k} = (1)$.
- Value: $\boldsymbol{v} = \boldsymbol{e}^0[1, \cdots, 2n]$.

Thus $\langle \boldsymbol{q}, \boldsymbol{k} \rangle = 0$ for key of input number tokens, and $\langle \boldsymbol{q}, \boldsymbol{k} \rangle \le -1$ otherwise. By Lemma C.6, the attention output is

$$\frac{1}{\text{ape}_{b,0} - 2}(a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}).$$

By Lemma C.1, we can use the subsequent MLP to get $(a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1})$ given the value of $\text{ape}_{b,0}$. Then we can calculate all $d_{i,j}$ using the MLP, which requires $O(n^2)$ hidden dimension by Lemma C.1.

Finally, we calculate $(r_{2n-1}, \cdots, r_0)$ by

$$r_j = \sum_{k=\max(0,j-(n-1))}^{\min(n-1,j)} d_{k,j-k}.$$

Block 2. This block of the Transformer uses several MLPs to executes line 4-12 of Algorithm 3. All the calculations below are also calculated at the position of b_0, subsequent to what we did in Block 1.

- For the calculation of s_i, it's easy to get the values via $(r_{2n-1}, \cdots, r_0)$.
- For the calculation of b_i, q_i, notice that $b_i \le p^m \le np^2$, thus we can use

  $$b_i = \sum_{j=0}^{np^2} \text{ReLU}(s_i - p^m)$$

  for each b_i, which requires $O(n^2)$ hidden dimension in total by Lemma C.2. Then $q_i = s_i - b_i p^m$, which can be easily implemented by MLP as well.
- For the calculation of f_i, g_i, we can get those values by

  $$\begin{aligned} f_i &= \text{ReLU}[q_i + b_{i-1} - (p^m - 1)] - \text{ReLU}[q_i + b_{i-1} - p^m], \\ g_i &= \text{ReLU}[q_i + b_{i-1} - (p^m - 2)] - \text{ReLU}[q_i + b_{i-1} - (p^m - 1)] \end{aligned}$$

  and Lemma C.2, which requires $O(n)$ hidden dimension in total.
- For the calculation of c_i, notice that

  $$\bigwedge_{1 \le i \le \gamma} \alpha_i = \text{ReLU}\left(\sum_{i=1}^{\gamma} \alpha_i - \gamma + 1\right), \quad \bigvee_{1 \le i \le \gamma} \alpha_i = 1 - \text{ReLU}\left(1 - \sum_{i=1}^{\gamma} \alpha_i\right).$$

  Combining with Lemma C.2, we can calculate the value of each c_i with $O(n)$ hidden dimension.
- Finally, for the calculation of $\tilde{o}_i$, we can use the similar fashion of the calculation of q_i. Since $q_i + b_{i-1} + c_{i-1} < 2p^m$, we can calculate each $\tilde{o}_i$ using constant hidden dimension, which implies we can calculate $\tilde{\boldsymbol{o}}$ using $O(n)$ hidden dimension in total.

Block 3. The last block of the Transformer executes the last step of Algorithm 3. Let's consider the token $o_{(i+1)m+j+1}$, where $j \in \{0, \cdots, m-1\}$, we want to predict the next token $o_{(i+1)k+j}$. We first COPY the value of $\tilde{\boldsymbol{o}}$ from the position of b_0, then extracts $\tilde{o}_{i+1}$ by $\tilde{o}_{i+1} = \langle \tilde{\boldsymbol{o}}, \boldsymbol{e}_{i+1} \rangle$ using the positional embedding of $\boldsymbol{u}^0_{o,i}$.

Subsequently, for the output token $o_{(i+1)k+j}$, the result $o_{(i+1)k+j} = \tilde{o}_{i+1} \bmod p^{j+1}$ is required. We first calculate o_{i+1}/p^{j+1} using the positional embedding and Lemma C.1, then calculate $\lfloor \tilde{o}_{i+1}/p^{j+1} \rfloor$

using the similar fashion to what we did when calculating s_i, b_i in Block 2. Since $\tilde{o}_{i+1} < 2p^m \le np^2$, this can be implemented by a MLP with $O(n)$ hidden dimension. Then we can calculate $\tilde{o}_{i+1} \bmod p^{j+1}$ using MLP. Similarly, we can finally get the value of $\lfloor \frac{\tilde{o}_{i+1} \bmod p^{j+1}}{p^j} \rfloor$ using a MLP with $O(n)$ hidden dimension.

Upon outputting the token o_0, the model anticipates the <EOS> token, employing an MLP to filter the hidden embeddings and output the word embedding for <EOS>. Thus, the final output from this layer is characterized by the equation:

$$\boldsymbol{e}_{o,i}^3 = \begin{cases} (o_{i-1}, i, 3) & \text{if } i > 0, \\ (-1, -1, 7) & \text{if } i = 0. \end{cases}$$

Predict Next Token. Given the output embeddings of the last transformer layer $\boldsymbol{e}_{o,i}^3$, and the word embeddings, the transformer can simply predict the next token by softmax.

In this construction, the norm of the parameters is bounded by $O(n^2)$, therefore, this construction can be implemented by a log-precision transformer with arbitrarily small error. □

F Experimental Details

In this section, we present the experimental details.

F.1 Datasets

The iterated addition and integer addition data are generated according to Algorithm 4. The multiplication data are generated according to Algorithm 5. Both datasets are used online for training and testing.

Algorithm 4: Iterated Addition Data Generation

```
Function large_number_add($a, b, base$):
    Input: $a$: List of digits of the first number
           $b$: List of digits of the second number
           $base$: The numerical base
    Output: $result$: List of digits of the sum of $a$ and $b$
    carry $\leftarrow$ 0, result $\leftarrow$ []
    max_length $\leftarrow$ max(length(a), length(b))
    for $i \leftarrow 0$ to max_length - 1 do
        sum $\leftarrow$ carry
        if $i <$ length(a) then
            sum $\leftarrow$ sum + a[i]
        end
        if $i <$ length(b) then
            sum $\leftarrow$ sum + b[i]
        end
        carry $\leftarrow$ floor(sum / base)
        result.append(sum $\bmod$ base)
    end
    if carry $\neq 0$ then
        result.append(carry)
    end
    return result
Function get_data($batch, length, num\_count, base$):
    Input:
    $batch$: Number of samples
    $length$: Maximum length of addends
    $num\_count$: Number of addends
    $base$: The numerical base
    Output: tokenized_data: Tensor of generated sequences
    data $\leftarrow$ random integers in range [0, base)  with shape  (batch, length, num_count)
    tokenized_data $\leftarrow$ []
    for $i \leftarrow 0$ to $batch - 1$ do
        numbers $\leftarrow$ data[i, :, :]
        strip leading zeros of numbers and get stripped_numbers
        for num in numbers do
            sum_digits $\leftarrow$ large_number_add(sum_digits, num, base)
        end
        reverse stripped_numbers and sum_digits
        add token of '+' and '=' and '<EOS>' to form sequence pad the sequence into the same
          length
        tokenized_data.append(sequence)
    end
    convert tokenized_data to tensor
    return tokenized_data
```

1807

Algorithm 5: Integer Multiplication Data Generation

```
Function large_number_mult($a, b, base$):
    Input: $a$: List of digits of the first number
           $b$: List of digits of the second number
           $base$: The numerical base
    Output: $result$: List of digits of the product of $a$ and $b$
    result ← [0] * (length(a) + length(b))
    for $i \leftarrow 0$ to $length(a) - 1$ do
        carry ← 0
        for $j \leftarrow 0$ to $length(b) - 1$ do
            product ← $a[i] * b[j]$ + result$[i + j]$ + carry
            carry ← floor(product / base)
            result$[i + j]$ ← product mod base
        end
        if carry > 0 then
            result[i + length(b)] ← result[i + length(b)] + carry
        end
    end
    strip leading zeros from result
    return result
Function get_mult_data($batch, length, base$):
    Input:
    $batch$: Number of samples
    $length$: Maximum length of multiplicands
    $base$: The numerical base
    Output: tokenized_data: Tensor of generated sequences
    data ← random integers in range [0, base) with shape (batch, length, 2)
    tokenized_data ← []
    for $i \leftarrow 0$ to $batch - 1$ do
        num_1 ← data[i, :, 0]
        num_2 ← data[i, :, 1]
        strip leading zeros of numbers and get stripped_numbers
        product_digits ← large_number_mult(num_1, num_2, base)
        reverse stripped_numbers and product_digits
        add token of '×' and '=' and '<EOS>' to form sequence pad the sequence into the same
          length
        tokenized_data.append(sequence)
    end
    convert tokenized_data to tensor
    return tokenized_data
```

F.2 Model Training

The experiments were conducted on a single NVIDIA GeForce RTX 4090 GPU over a duration of two weeks, investigating the differences in performance between standard precision and low precision operations. To avoid some unexpected issues of hardware, we also conduct the same experiments on NVIDIA A100 GPUs, and the results are consistent with the results on NVIDIA GeForce RTX 4090 GPU. We try 3 different seeds and select the maximum accuracy for each task.

The model configuration in our experiments is presented in Table 2, and the training configuration is presented in Table 3.

Model Configuration	
Model Depth	$\{3, 5\}$
Hidden Dimension	256
Attention Heads	4
Positional Embeddings	RoPE
Activation	NewGeLU

Table 2: Model Configuration for Transformer in Experiments.

Training Configuration	
Epochs	1
Learning Rate	1e-3
Optimizer	AdamW
β_1	0.9
β_2	0.999
Weight Decay	0.01
Learning Rate Scheduler	Cosine Scheduler with Warmup
Numerical Precision	$\{$`float32`, `bfloat16`$\}$

Table 3: Training Configuration in Experiments.

F.3 Integer Addition Results

The results of the experiments are presented in Table 4.

	Base-2		**Base-10**	
Length	`float32` **Accuracy**	`bfloat16` **Accuracy**	`float32` **Accuracy**	`bfloat16` **Accuracy**
8	99.8%	99.6%	99.4%	99.0%
16	99.3%	98.4%	99.2%	98.1%
24	98.9%	96.3%	99.2%	97.4%
32	99.3%	95.9%	99.2%	94.1%

Table 4: Evaluation of integer addition accuracy across various length with both 32-bit and 16-bit precision.

F.4 Fine-tuing Configuration, Generation Configuration, and Prompt For LLM

The fine-tuning configuration and generation configuration for LLMs is listed in Tables 5 and 6. The detailed prompts for the three elementary arithmetic tasks are listed in the Tables 7 and 8 and generation configuration can be found in the Table 5.

Generation Configuration	
TopK	50
TopP	0.95
Temperature	0.1

Table 5: Generation Configuration for LLAMA 3.1 8B Instruct in arithmetic tasks.

Fine-tuning Configuration	
Rank	8
Scaling Factor	16
Dropout Rate	0.05
Epochs	1
Learning Rate	2e-4
Optimizer	AdamW
β_1	0.9
β_2	0.999
Weight Decay	0.01
Learning Rate Scheduler	Cosine Scheduler with Warmup
Warmup Ratio	0.1
Numerical Precision	{`bfloat16`, `int4`}

Table 6: Generation Configuration for LLAMA 3.1 8B Instruct in arithmetic tasks.

Prompt for LLAMA 3.1 8B Instruct in Integer Addition and Iterated Addition tasks.
Please directly calculate the following arithmetic expression in base <base> with the following format: <Expression> = <Result> It is important that you should not show any intermediate steps in your calculation process. The final answer should be computed in one step and provided the final result immediately without any explanation. Here are some examples 32 + 78= 110 1234 + 4567 + 2134 + 4567 = 12502 2135 + 523 + 2135 + 523 = 5316 2314 + 4567 + 2314 + 4567 = 13762 Arithmetic Expression: <Expression>

Table 7: Prompt for LLAMA 3.1 8B Instruct in Integer Addition and Iterated Addition tasks.

Prompt for LLAMA 3.1 8B Instruct in Integer Multiplication task.
Please directly calculate the following arithmetic expression in base <base>. It is important that you should not show any intermediate steps in your calculation process. The final answer should be computed in one step and provided the final result immediately without any explanation. Here are some examples Examples: 32 * 56 = 1792 867 * 467 = 404889 123 * 456 = 56088 Arithmetic Expression: <Expression>

Table 8: Prompt for LLAMA 3.1 8B Instruct in Integer Multiplication task.

F.5 Reference Results for LLMs

We also provide the results of GPT-4o and GPT-4o-mini as a baseline for these arithmetic tasks base-10 for reference. The results are presented in Table 9.

Task	Length	GPT-4o	GPT-4o-mini
Addition of 2 numbers	1	100.0%	100.0%
	4	99.9%	98.8%
	7	97.5%	51.4%
	10	96.3%	46.0%
	13	93.3%	44.0%
Addition of 3 numbers	1	100.0%	100.0%
	3	99.8%	99.6%
	5	98.9%	73.4%
	7	69.2%	9.1%
	9	38.5%	5.8%
Addition of 5 numbers	1	100.0%	100.0%
	2	100.0%	99.4%
	3	100.0%	89.5%
	4	88.4%	31.1%
	5	86.8%	24.7%
Multiplication of 2 numbers	1	100.0%	100.0%
	2	100.0%	97.5%
	3	76.6%	44.7%
	4	21.5%	7.6%
	5	4.1%	0.7%

Table 9: The Performance of GPT-4o and GPT-4o-mini on the arithmetic tasks.