Transformers Can Do Arithmetic with the Right Embeddings

Sean McLeish[1*], **Arpit Bansal**[1*], **Alex Stein**[1], **Neel Jain**[1], **John Kirchenbauer**[1], **Brian R. Bartoldson**[2], **Bhavya Kailkhura**[2], **Abhinav Bhatele**[1], **Jonas Geiping**[3], **Avi Schwarzschild**[4], **Tom Goldstein**[1]

[1] University of Maryland, [2] Lawrence Livermore National Laboratory, [3] ELLIS Institute Tübingen, Max Planck Institute for Intelligent Systems, Tübingen AI Center, [4] Carnegie Mellon University

Abstract

The poor performance of transformers on arithmetic tasks seems to stem in large part from their inability to keep track of the exact position of each digit inside of a large span of digits. We mend this problem by adding an embedding to each digit that encodes its position relative to the start of the number. In addition to the boost these embeddings provide on their own, we show that this fix enables architectural modifications such as input injection to improve performance even further.

With positions resolved, we can study the logical extrapolation ability of transformers. Can they solve arithmetic problems that are larger and more complex than those in their training data? We find that training on only 20 digit numbers with a single GPU for one day, we can reach state-of-the-art performance, achieving up to 99% accuracy on 100 digit addition problems. Finally, we show that these gains in numeracy also unlock improvements on other multi-step reasoning tasks including sorting and multiplication.

1 Introduction

Much of the recent work on Large Language Models (LLMs) focuses on their ability to solve problems in natural language and code generation. Despite progress in these domains, transformers still struggle to perform complex multi-step and algorithmic reasoning tasks in a zero shot setting without resorting to tool use. To study algorithmic reasoning in a sterile laboratory setting, the academic community focuses on simple arithmetic test problems like addition. Addition is simple enough that modest-sized LLMs can (in principle) be trained from scratch to do it without running into capacity and training budget limitations, yet complex enough that even large industrial models fail on large numbers without a code interpreter (Loeber, 2024).

Prior studies indicate that addition is hard for transformers (Lee et al., 2023; Shen et al., 2023; Zhou et al., 2023, 2024). Our experiments indicate that this difficulty stems from their inability to clearly represent the exact position of a digit within a long sequence of digits. To address this problem, we propose a simple modification to the data representation that directly addresses this shortcoming. Our *Abacus Embeddings* are simple learned positional embeddings that are used to encode positions within each span of numerical tokens. Combining Abacus Embeddings and standard positional embeddings, we observe dramatic improvements in accuracy such that models trained with at most 20 digit operands can generalize to problems with 120 digit operands. This represents a state-of-the-art generalization factor of $6\times$, with the previous state of the art being only $2.5\times$. To the best of our knowledge, these are the longest sequences on which learned addition has ever been demonstrated.

[*]Equal Contribution, correspondence to: `smcleish@umd.edu`, `bansal01@umd.edu`.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

We also study several other methods of improving arithmetic and generalization in transformers. We find that incorporating *input injection*—skip connections inserted between the input layer and each decoder layer—can reduce generalization errors by 50% over the Abacus Embedding baseline. We also find that together with our embeddings looped transformer architectures, which contain recurrent layers in which the same parameters are re-used multiple times, can achieve near-perfect generalization on addition problems we consider. These results are shown in Appendix Section A.4

Since our proposed methods solve large addition problems successfully, we evaluate whether the same approaches can be used to improve other kinds of algorithmic learning. In Appendix Section A.3, we explore multiplication problems of up to 15 digit numbers and sorting over arrays of up to 10 numbers, making this the first study of extreme length generalization techniques for addition that transfer to other algorithmic tasks. Our contributions can be summarized as follows.

- We propose a new positional embedding called *Abacus Embeddings* to better capture the significance of each digit, which leads to near-perfect in-distribution generalization.
- We show that when we combine Abacus Embeddings with input injection and looped transformers performance further improves, increasing from 92.9% to 99.1% in out-of-distribution accuracy, an 87% reduction in error compared to using the embeddings with standard architectures alone.

2 Related Work

Arithmetic. Solving arithmetic with next token prediction is a difficult problem that attracts a lot of attention (e.g. Saxton et al., 2019). However, in zero-shot settings, even incredibly strong commercial API models struggle with very large addition problems (e.g. up to 100 digits) without access to tools. Among attempts to improve arithmetic performance of transformer-based models, reversing the digits so the arguments are written with the least significant digit first is popular (Lee et al., 2023; Shen et al., 2023; Zhou et al., 2023, 2024). Furthermore, changing the data format by adding explicit index characters improves model capability for addition (Zhou et al., 2023, 2024; Olsson et al., 2022).

Weight Sharing. Weight sharing and recurrence can be used to make models adaptive and help generalize to harder problems (Dehghani et al., 2018; Sukhbaatar et al., 2019; Lan et al., 2020; Ibarz et al., 2022). Schwarzschild et al. (2021) and Bansal et al. (2022) explore an end-to-end learning approach using recurrent convolutional neural networks to learn algorithms from input-output pairs, tackling algorithmic tasks like prefix sums, mazes, and chess. Weight sharing for algorithmic reasoning is also helpful with transformers and we use the *looped transformer* in some of our experiments below. A looped transformer has a transformer block called recurrently on its own output lending itself to executing iterative algorithms (Giannou et al., 2023; Yang et al., 2023a; de Luca & Fountoulakis, 2024).

Positional Embeddings. Indicating the position of tokens in a sequence to transformer models is critical for language modeling (Vaswani et al., 2017). Absolute positional embeddings (APE) are learned embeddings that are added to token embeddings before the first layer of the transformer (Vaswani et al., 2017). However, these absolute embeddings inhibit length generalization (Press et al., 2022). Kazemnejad et al. (2023) show that decoder layers can still learn positional information with no explicit positional embeddings. No positional embeddings (NoPE) can achieve good length generalization performance for small algorithmic tasks and even outperform some specialized embeddings. The latest and most useful for arithmetic is Functional Interpolation for Relative Position Embeddings (FIRE) (Li et al., 2023). FIRE shows the strongest length generalization to date, which leads to length generalization by $2.5\times$ on addition (Zhou et al., 2024) when combined with randomized embeddings (Ruoss et al., 2023). We go into more detail on positional embeddings in Appendix A.1.1. In this work, we focus on NoPE and FIRE embeddings since these are the best performers for addition in reversed format among existing embeddings (Zhou et al., 2024).

3 Achieving Length Generalization for Addition

We focus on two main hypotheses: (1) the positional information for individual digits within numbers is being lost and (2) recurrence can improve the reasoning abilities of transformer architectures on

Figure 1: Visualization of data formats and positional embeddings. *Abacus Embeddings* give the same positional embeddings to all digits of the same significance.

multi-step arithmetic reasoning problems. We briefly discuss the training and evaluation setup before describing each of our improvements in detail.

Experimental Setup. We train decoder-only causal language models to solve addition problems. Following prior work (Zhou et al., 2023, 2024; Shen et al., 2023; Kazemnejad et al., 2023; Lee et al., 2023), inputs are formatted least significant digit first, e.g. $98282 + 3859172 = 2787472$. Unlike prior work, we do not add any padding between digits (Shen et al., 2023) and do not pad any numbers with zeros, neither in the case of carry digits (Zhou et al., 2024), nor to make all operands the same length (Shen et al., 2023). We train on all combinations of operand lengths less than or equal to i and j where i and j are the maximum lengths of the first and second operands, respectively. For this study all training sets have 20 million samples and $i = j$, hence we can use one number to define the dataset i, where i is the maximum length of either operand. For further details on data construction and training we refer to Appendix A.6.

We report model accuracy for each (i, j) length pair and unlike most existing work, we also include accuracy for pairs where $i \neq j$ to highlight all instances of extrapolation. This extensive tabulation is costly and makes inference the main computational burden of this study. We measure accuracy in the strict sense where only exact matches of all output digits are counted as correct, i.e. if a single digit is incorrect then the example is marked as wrong and we refer to this as *exact match accuracy*. We have the following three evaluation categories: (i) in distribution (ID) where the models are tested on problems up to the maximum size seen during training; (ii) out of distribution (OOD) where the models are tested on problems greater than the maximum size seen during training but both operands are at most 100 digits; (iii) and extreme out of distribution (100+ digit OOD) where the models are tested on problems where both operands are of the same length and are both more than 100 digits and less than 160 digits. In the 100+ OOD setting, we only analyze problems where the operands are the same length ($i = j$) due to inference costs at this scale.

We consider two standard transformer architectures. First, we use a standard autoregressive transformer model (ST) where multiple decoder layers are stacked in a feedforward manner. Second, we enhance this standard transformer model by incorporating *input injection* (ST w/ II), where the embedded inputs are added to the input of each decoder layer (Ma et al., 2022; Bansal et al., 2022; Anil et al., 2022a). We visually describe the architectures in the Appendix Figure 19.

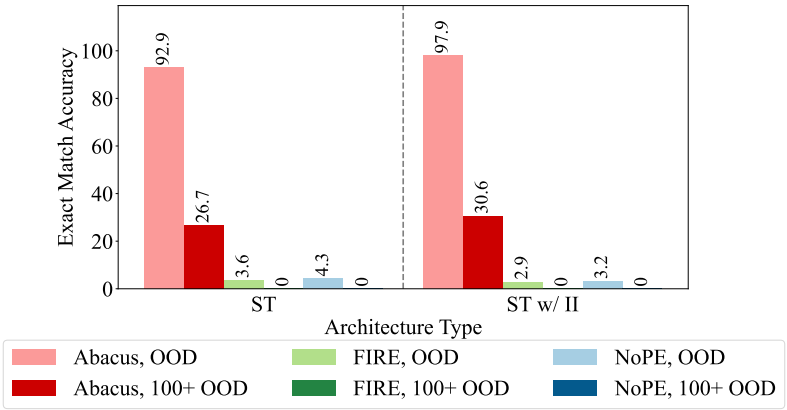

Figure 2: Mean exact match accuracy of three models of depth sixteen on size 20 data, varying the architecture and embeddings. Abacus Embeddings improve accuracy for addition over FIRE and NoPE.

3.1 Abacus Embeddings Help Align Digits

From prior work and our own initial experiments, we observe that even when input numbers are presented least-significant digit first and training data is stratified and abundant (several million examples), standard transformers struggle to learn multi-digit addition. We also observe that humans do long addition by first aligning the digits of the same significance into columns. Thus, our first hypothesis is that the significance of each digit (i.e. each digit's position relative to the beginning of the number) is not easy for transformers to represent, and that this sub-problem presents more of a hurdle than the actual addition itself.

Prior work addresses this by proposing explicit index hints in the inputs and outputs of the addition, for example $a6b7c5 + a1b6c3 = a7b3c9$; finding that transformers perform much better on addition with the information provided by such hints (Zhou et al., 2023, 2024). However, index hints of this form increase the input context length required and *double* the output length and inference cost of solving a given addition problem. Furthermore, Zhou et al. (2024) find that the ability of models trained with index hints to generalize is sensitive to the particular random initialization.

To address the limitations of transformers at representing positional information, we design a specially built positional embedding that encodes the location of each digit relative to the start of the current number. We call this *Abacus Embeddings*. We apply the same positional embedding to all digits of the same significance, providing an explicit signal that the model can use to align digits. We visually describe these embeddings in Figure 1.[2]

We take inspiration from *Randomized Embeddings* (Ruoss et al., 2023) but instead of using random ascending indices to represent positions in a sample, we use consecutive ascending indices with a random starting position to allow for length generalization. Specifically, during training we give consecutive positional embeddings to each digit in a number, starting from a randomly chosen offset value from $U[1, k]$, where k is a hyperparameter. Unless otherwise stated the default value for k in this study is 100. For example, if the input is 123, the positional encodings are $\beta, \beta + 1, \beta + 2$ where $\beta \sim U[1, 100]$, which are then passed through a learned embedding matrix. The value sampled from $U[1, k]$ is the same for all numbers in a batch, meaning all digits of the same significance obtain the same positional embedding. This training scheme allows the model to see a wide range of positional embeddings, even when training sequences are short. At test time, we set $\beta = 1$.

Abacus Embeddings Solve Addition. Abacus Embeddings improve generalization performance up to 100 digits and beyond for standard transformer architectures. In Figure 2, we highlight the comparative boost Abacus Embeddings have over standard transformer architectures and embeddings for performing addition, taking the mean accuracy of three models in all cases. Additionally, In Appendix A.5.4, we present 2D grid plots for several other experiments that are depicted as bar charts in the main text. Zhou et al. (2024) find that operand lengths of up to forty digits are required during training for good generalization to 100 digit addition during testing (albeit not robustly). We find that with our Abacus Embeddings, we can achieve similar accuracy and larger extrapolation using a standard model with input injection trained on maximum operand sizes of 20 digits.

As Abacus Embeddings are a variant of absolute positional embeddings, technically they cannot generalize beyond the relative positions seen during training. However the hyperparameter k that randomizes the starting offset used for each individual addition example can be increased to enable generalization by training a larger range of embeddings for a given computational budget. Relatedly, Appendix Figure 8 shows that training on larger datasets improves performance, even for operands with fewer than 100 digits.

4 Discussion

Across our experiments, we find that our novel Abacus Embeddings improve performance dramatically both when applied to standard transformers as well as recurrent variants. We hope that our work deepens the community's understanding of these problems and paves the way for further advancements in the algorithmic reasoning capabilities of large language models.

[2] In Appendix A.2, we motivate these embeddings further with experiments demonstrating their utility in solving a bitwise OR task and show their performance on multiplication and sorting in Appendix A.3.

References

Anil, C., Pokle, A., Liang, K., Treutlein, J., Wu, Y., Bai, S., Kolter, J. Z., and Grosse, R. B. Path independent equilibrium models can better exploit test-time computation. *Advances in Neural Information Processing Systems*, 35:7796–7809, 2022a.

Anil, C., Wu, Y., Andreassen, A., Lewkowycz, A., Misra, V., Ramasesh, V., Slone, A., Gur-Ari, G., Dyer, E., and Neyshabur, B. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems*, 35:38546–38556, 2022b.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Bansal, A., Schwarzschild, A., Borgnia, E., Emam, Z., Huang, F., Goldblum, M., and Goldstein, T. End-to-end algorithm synthesis with recurrent networks: Logical extrapolation without overthinking. *Advances in Neural Information Processing Systems*, 35, 2022.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Chi, T.-C., Fan, T.-H., Ramadge, P., and Rudnicky, A. Kerple: Kernelized relative positional embedding for length extrapolation. In *Advances in Neural Information Processing Systems*, 2022.

Chi, T.-C., Fan, T.-H., Rudnicky, A., and Ramadge, P. Dissecting transformer length extrapolation via the lens of receptive field analysis. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13522–13537, 2023.

de Luca, A. B. and Fountoulakis, K. Simulation of graph algorithms with looped transformers. *arXiv preprint arXiv:2402.01107*, 2024.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, L. Universal transformers. In *International Conference on Learning Representations*, 2018.

Dziri, N., Lu, X., Sclar, M., Li, X. L., Jian, L., Lin, B. Y., West, P., Bhagavatula, C., Bras, R. L., Hwang, J. D., et al. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2023.

Geiping, J. and Goldstein, T. Cramming: Training a language model on a single gpu in one day. In *International Conference on Machine Learning*, pp. 11117–11143. PMLR, 2023.

Giannou, A., Rajput, S., Sohn, J.-y., Lee, K., Lee, J. D., and Papailiopoulos, D. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Golkar, S., Pettee, M., Eickenberg, M., Bietti, A., Cranmer, M., Krawezik, G., Lanusse, F., McCabe, M., Ohana, R., Parker, L., et al. xval: A continuous number encoding for large language models. *arXiv preprint arXiv:2310.02989*, 2023.

Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A. J., Bošnjak, M., Vitvitskyi, A., Rubanova, Y., et al. A generalist neural algorithmic learner. In *Learning on graphs conference*, pp. 2–1. PMLR, 2022.

Jelassi, S., d'Ascoli, S., Domingo-Enrich, C., Wu, Y., Li, Y., and Charton, F. Length generalization in arithmetic transformers. *arXiv preprint arXiv:2306.15400*, 2023.

Kazemnejad, A., Padhi, I., Ramamurthy, K. N., Das, P., and Reddy, S. The impact of positional encoding on length generalization in transformers. *arXiv preprint arXiv:2305.19466*, 2023.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

Lee, N., Sreenivasan, K., Lee, J. D., Lee, K., and Papailiopoulos, D. Teaching arithmetic to small transformers. *arXiv preprint arXiv:2307.03381*, 2023.

Li, S., You, C., Guruganesh, G., Ainslie, J., Ontanon, S., Zaheer, M., Sanghai, S., Yang, Y., Kumar, S., and Bhojanapalli, S. Functional interpolation for relative positions improves long context transformers. *arXiv preprint arXiv:2310.04418*, 2023.

Loeber, J. #16: Notes on Arithmetic in GPT-4, February 2024. URL https://loeber.substack.com/p/16-notes-on-arithmetic-in-gpt-4.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Ma, X., Zhou, C., Kong, X., He, J., Gui, L., Neubig, G., May, J., and Zettlemoyer, L. Mega: moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022.

McLeish, S., Schwarzschild, A., and Goldstein, T. Benchmarking chatgpt on algorithmic reasoning. *arXiv preprint arXiv:2404.03441*, 2024.

Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T., Mann, B., Askell, A., Bai, Y., Chen, A., et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023. URL https://api.semanticscholar.org/CorpusID:257532815.

Peng, B., Quesnelle, J., Fan, H., and Shippole, E. Yarn: Efficient context window extension of large language models. *International Conference on Learning Representations*, 2024.

Press, O., Smith, N., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=R8sQPpGCv0.

Qian, J., Wang, H., Li, Z., Li, S., and Yan, X. Limitations of language models in arithmetic and symbolic induction. *arXiv preprint arXiv:2208.05051*, 2022.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Rodionov, G. and Prokhorenkova, L. Discrete neural algorithmic reasoning. *arXiv preprint arXiv:2402.11628*, 2024.

Ruoss, A., Delétang, G., Genewein, T., Grau-Moya, J., Csordás, R., Bennani, M., Legg, S., and Veness, J. Randomized positional encodings boost length generalization of transformers. *arXiv preprint arXiv:2305.16843*, 2023.

Saxton, D., Grefenstette, E., Hill, F., and Kohli, P. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557*, 2019.

Schwarzschild, A., Borgnia, E., Gupta, A., Huang, F., Vishkin, U., Goldblum, M., and Goldstein, T. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. *Advances in Neural Information Processing Systems*, 34, 2021.

Shaw, P., Uszkoreit, J., and Vaswani, A. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Shazeer, N. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Shen, R., Bubeck, S., Eldan, R., Lee, Y. T., Li, Y., and Zhang, Y. Positional description matters for transformers arithmetic. *arXiv preprint arXiv:2311.14737*, 2023.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Sukhbaatar, S., Grave, E., Bojanowski, P., and Joulin, A. Adaptive attention span in transformers. In Korhonen, A., Traum, D., and Màrquez, L. (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 331–335, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1032. URL https://aclanthology.org/P19-1032.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pp. 22084–22102. PMLR, 2022.

Wang, H., Ma, S., Dong, L., Huang, S., Zhang, D., and Wei, F. DeepNet: Scaling Transformers to 1,000 Layers. *arXiv:2203.00555 [cs]*, March 2022. URL http://arxiv.org/abs/2203.00555.

Yang, L., Lee, K., Nowak, R., and Papailiopoulos, D. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023a.

Yang, Z., Ding, M., Lv, Q., Jiang, Z., He, Z., Guo, Y., Bai, J., and Tang, J. Gpt can solve mathematical problems without a calculator. *arXiv preprint arXiv:2309.03241*, 2023b.

Zhai, X., Kolesnikov, A., Houlsby, N., and Beyer, L. Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 12104–12113, 2022.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J., Bengio, S., and Nakkiran, P. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

Zhou, Y., Alon, U., Chen, X., Wang, X., Agarwal, R., and Zhou, D. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024.

A Appendix

Limitations There are some intrinsic limitations that accompany any study involving language model training from scratch under compute constraints. However, the primary point of relevance for this study is that although we show the compatibility of Abacus Embeddings with FIRE and RoPE embeddings, we do not actually explore any natural language tasks. In the future, a larger scale study including natural language would be needed to understand further how Abacus Embeddings would perform on heterogeneous tasks comprising both numerical and natural language inputs.

A.1 Extended Related Works

A.1.1 Positional Embeddings.

To address this issue of absolute embeddings not generalizing, Shaw et al. (2018) propose relative embeddings (RPE) which are embedded during the attention computation, a mechanism further simplified by Raffel et al. (2020). Others further modify relative embeddings to improve length generalization including Sandwich (Chi et al., 2023), Kerple (Chi et al., 2022), and Alibi (Press et al., 2022) positional embeddings. Rotary Positional Embeddings (RoPE) (Su et al., 2024) are commonly used in state-of-the-art open source transformers (e.g. Touvron et al., 2023). However, RoPE does limit the length generalization as models are trained only using rotations based on training data length (Kazemnejad et al., 2023; Press et al., 2022). For improved length generalization, one can add post-training extensions (Peng et al., 2024).

FIRE embeddings are additive embeddings in the attention mechanism: $A_{RPE}(X) = XW_Q(XW_K)^T + B$ where $B_{i,j} = f_\theta\left(\frac{\log(c(i-j)+1)}{\log(c\max(i,L)+1)}\right)$ and c, L are learnable parameters. Li et al. (2023) show empirically that these embeddings allow for length generalization and theoretically show they are capable of representing many other embedding types. Ruoss et al. (2023) propose using a random subset of a larger set of possible positions during training so that larger positional embeddings are trained. Zhou et al. (2024) use randomized FIRE (Ruoss et al., 2023; Li et al., 2023) embeddings to achieve length generalization on arithmetic tasks, which use randomized positions as input to the small multi layer perceptron used in FIRE embeddings.

A.1.2 Arithmetic and Algorithmic Reasoning.

Golkar et al. (2023) approach arithmetic by embedding real numbers by scaling a single fixed token-embedding for numbers. Moreover, Dziri et al. (2023) show multiplication is a hard problem for GPT-3 (Brown et al., 2020) even when finetuned on this task. Dziri et al. (2023) further show that GPT-4 (OpenAI, 2023) struggles to obtain high in-distribution accuracy on multiplication, even with a scratchpad. However, Lee et al. (2023) find that with a detailed scratchpad, small transformers can perform multiplication in-distribution. Arithmetic is a subset of the larger class of algorithmic reasoning problems that focus on the ability to learn and execute algorithms and generalize to longer problems (Anil et al., 2022b; Jelassi et al., 2023; Yang et al., 2023b; Veličković et al., 2022; Rodionov & Prokhorenkova, 2024). The more general algorithmic reasoning field includes work on various architectures and data modalities aimed at learning algorithms from data. Veličković et al. (2022) and Rodionov & Prokhorenkova (2024), for example, train neural networks to execute specific algorithmic tasks by training on input-output pairs as well as intermediate steps and hints. Additionally, recent work aims to improve reasoning in LLMs (Zhou et al., 2023), but McLeish et al. (2024) demonstrate that LLMs, even with code interpreters, are less than perfect at algorithmic reasoning tasks, indicating a crucial need for advancements in our methodologies. This paper takes a step towards improving LLM arithmetic and algorithmic capabilities without tool use.

A.2 Bitwise OR on Binary Vectors

A necessary condition to perform addition is aligning digits of the same significance. We begin by examining positional embeddings for exactly this task. To do this we analyze the bitwise OR task, where the model has to output left aligned position wise OR of two binary vectors. We present samples from the dataset in Section A.2.1, these are left aligned to be representative of the task of aligning digits for reversed addition.

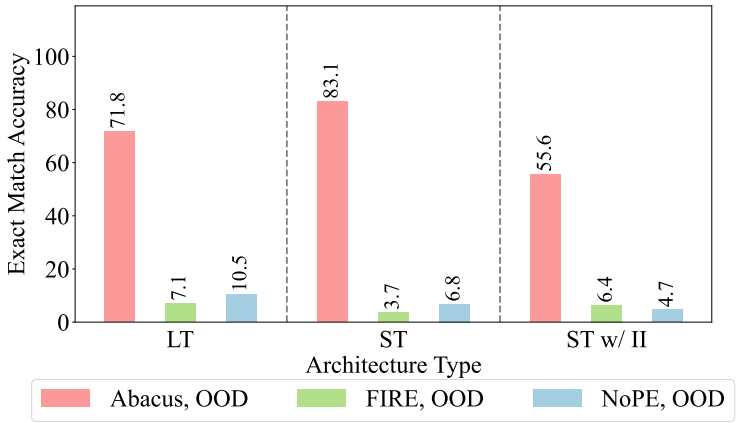

Figure 3: Accuracy of models on the bitwise OR task when trained on data with size up to 20, varying over different positional embeddings and architectures. Abacus Embeddings heavily improve performance on this task.

We train standard transformer, standard transformer with input injection and looped transformer models on the position wise or task, on a dataset where the maximum length of either input vector is twenty. This result is shown in Figure 3. Here we see that the Abacus Embeddings allow all models to generalize further on this task than the other embeddings which prior work for addition focuses on. As with addition, we see that looped transformers perform better than the standard architectures with FIRE or NoPE embeddings. We do note that these accuracies are not as high we report for addition. We hypothesize this is because the model is having to repeatedly predict the same token multiple times, this has been thought to be the cause of errors in prior addition work(Qian et al., 2022). When we analyzed the errors in this task we found they were predominantly caused by the model outputting one too few or too many zeros.

A.2.1 Example Data

$$000010 \oplus 00000000000000 = 00001000000000$$

$$000100 \oplus 0000000 = 0001000$$

$$001 \oplus 00000 = 00100$$

A.3 Pushing the Limits of Algorithmic Reasoning for Transformers

While there is an emphasis on addition as a difficult problem in existing work, our methods perform so well that we look beyond addition and apply our tools to even more difficult problems, including multiplication and sorting.

A.4 Recurrence In Transformers Boosts Performance

With positional embeddings addressed, next we explore whether recurrent architectures can further improve the ability of transformers to perform multi-digit addition. We use the term *recurrent block* to refer to a set of decoder layers with distinct weights and *recurrences* to refer to the number of times the recurrent block is repeated. We use the term *effective depth* to mean the number of layers used in a transformer, whether their weights are unique or not. Unless otherwise stated, we use a maximally recurrent architecture, i.e. only one unique layer recurred to achieve the effective depth. We also employ input injection, skip-connections that propagate a copy of the input to each layer in the network.

The Benefits of Recurrence. We explore the effect of varying the size of the recurrent block while keeping the effective depth fixed. We perform this ablation by halving the number of layers in the recurrent block and doubling the number of recurrences, sweeping from a model with sixteen layers in

the block and a single recurrence (16×1, i.e. a standard transformer), through to one layer in the block with sixteen recurrences (1×16). Analyzing Figure 4, we see further performance improvements are possible in some cases with the combination of both recurrence and Abacus Embeddings. In particular, a model with two recurrences (8×2) incurs half the error of the purely non-recurrent model (16×1) for OOD problems and enjoys increased accuracy on 100+ OOD problems. Although the experiments presented in Figure 4 are a fair comparison across depth, the purely standard transformer models have many more parameters than their recurrent counterparts.

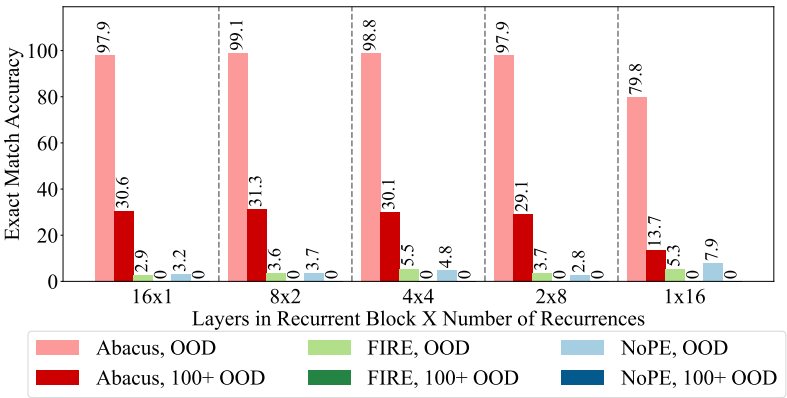

Figure 4: Varying the size of the recurrent block, while maintaining an effective depth of 16 and training on size 20 data. We see that a recurrent model with eight layers in the recurrent block and two recurrences is the most accurate of all effective depth 16 models, halving the error rate of a standard model with input injection in the OOD evaluation when using Abacus Embeddings.

A.4.1 Integer Multiplication

We now study a harder task, multiplication of natural numbers, where the length of the output may be the sum of the lengths of the operands. Compared to addition, where the output is at most one digit more than the longest operand, multiplication has longer-distance dependency and the output length scales much faster as problem size increases.

To adapt from addition to multiplication, we make some small changes to our set-up. First, we remove the input injection from inside the recurrent block and second, we divide the gradients in the recurrent block by the number of recurrences, down-weighing the gradient update from batches with many recurrences (Bansal et al., 2022). (We analyze the impact of these design decisions for addition models in Appendix Figure 16.) We only examine looped transformers as the compute required for training and hyperparameter search for multiplication is far greater than for addition, limiting us to a much smaller scale analysis.

Abacus Embeddings help looped transformers reach near-perfect accuracy in-distribution for multiplication. In Figure 5, we show how the training distribution, surrounded by the red square fully saturates with Abacus Embeddings. In fact, models with our Abacus Embeddings achieve higher in distribution accuracy on 15 digit multiplication than prior work (Shen et al., 2023) and do not require padding each operand to the same length with zeros. In particular, we highlight that the specific problems that models trained with FIRE embeddings struggle to solve are the hardest problems in the training set and Abacus Embeddings outperform them in this key area (see the lower right corner of the red boxes in Figure 5).

A.4.2 Array Sorting

While both addition and multiplication accept only two operands, we now analyze the task of sorting arrays of multiple variable length numbers, a more challenging testbed for evaluating the generalization abilities of our Abacus Embeddings. We present each sorting problem using alphabetical indices for each (reversed) number in an input array where the expected output is the alphabetical indices in ascending order. For example, $a : 64957, b : 99963, c : 10218, d : 7141, e : 05781 = d, e, b, a, c$. We train with arrays of up to 10 numbers each having up to 10 digits and then evaluate with arrays of

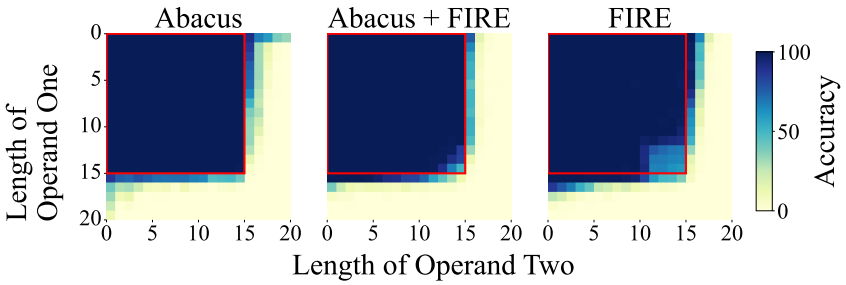

Figure 5: Exact match accuracy of looped transformer models trained on multiplication, with four layers in the recurrent block and four recurrences. The red square denotes in distribution testing on up to 15 digit operands. We see the models with Abacus Embeddings achieve near perfect in distribution accuracy. Combining Abacus Embeddings with FIRE also improves in distribution accuracy on the hardest in distribution problems (bottom right), comparing to the FIRE-only baseline.

Table 1: Exact match accuracy for sorting with various positional embeddings. All results are percentages of the test set and all models here are standard transformers with eight layers.

	FIRE	Abacus	Abacus + FIRE
OOD (number length - 30)	55.32	**68.63**	67.28
OOD (array length - 30)	**21.35**	9.67	21.11
All OOD (30 × 30)	3.73	2.65	**4.48**
All OOD (20 × 20)	14.65	9.78	**16.91**

Table 2: Accuracy for sorting with various architectures for sorting. ST denotes standard transformer, ST w/ II denotes standard transformer with input injection, and LT denotes looped transformer models. The standard transformer has the best exact match accuracy. When measuring the accuracy on identifying only the minimum element of the array, looped transformers outperform all others. All results are percentages of the test set.

	ST	ST w/ II	LT
All OOD (exact string match)	**4.48**	3.84	2.60
All OOD (min. elem. only)	49.73	60.09	**68.51**

up to 30 numbers each having up to 30 digits. We give more detail on the sorting data construction process in Appendix A.6.

In this setting, we explore two axes of generalization. First, we increase the maximum possible length of the input numbers to 30 digits while maintaining the maximum array length to 10 and refer to this scenario as "OOD (number length - 30)." Second, we increase the number of inputs in the array to be sorted to 30 while keeping the maximum digit length of each number at 10 and term this scenario "OOD (array length - 30)." Finally, we consider a scenario where both axes are increased simultaneously, referred to as "all OOD."

In Table 1, we illustrate the performance of a standard transformer (eight layers) trained with different embeddings—FIRE, Abacus, and their combination. Again, our results demonstrate that the combined embedding approach enhances the model's ability to generalize, surpassing the performance of either embedding alone in the "all OOD" setting. However, in Table 2, we observe mixed results when pairing the Abacus+FIRE Embeddings combination with different model architectures with effective depth eight. For sorting, different architectures appear to be better suited to different types of extrapolation, for example the looped transformer is best at extrapolating for finding the minimum element but not for sorting the whole array.

Overall, the superior sorting performance of the Abacus Embeddings underscores their potential utility across a broader spectrum of algorithmic tasks beyond basic arithmetic. Abacus Embeddings may be instrumental in use cases requiring transformer models to perform a variety of complex positional, numerical, and/or relational reasoning tasks.

A.4.3 Abacus and Relative Embeddings

As Abacus Embeddings are only applied to numbers, to incorporate Abacus Embeddings into a general purpose model, they must be compatible with other relative embeddings to maintain good downstream performance on non-arithmetic tasks. We examine these types of combinations here and conclude that Abacus Embeddings complement techniques that are good for natural language well, suggesting that these combinations could be powerful for large-scale general models.

Although Abacus Embeddings are implicitly combined with NoPE (no positional embeddings) embeddings for all experiments seen so far, most state-of-the-art open source models use Rotary Embeddings. Rotary Embeddings are weak for length generalization. We show that combining Abacus Embeddings with RoPE does, in fact, yield improvement in operand length generalization. However, in Figure 6, we demonstrate the true potential for integrating Abacus Embeddings into a more general system, showing that the combination of Abacus Embeddings with FIRE unlocks generalization well beyond the problems that FIRE embeddings can solve on their own.

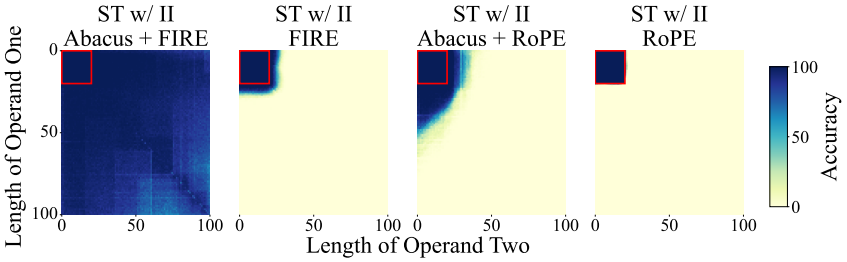

Figure 6: Exact match accuracy of standard transformer of depth 16 with input injection, trained on up to size 20 data. The red square denotes in distribution testing. Combining Abacus Embeddings with FIRE or RoPE embeddings improves out of distribution accuracy for addition, over the baseline models without Abacus Embeddings.

A.5 Further Addition Results

A.5.1 The Impact of Recurrence without Abacus

In Figure 7, we compare all architecture variants using both FIRE and NoPE embeddings trained on addition over operands with up to 40 digits. Despite having approximately $10\times$ fewer parameters than the other models, we see that the looped transformer (recurrent, with input injection and progressive loss), achieves the best out of distribution performance using either position embedding. In Figure 8 in the Appendix, we show this result is robust across multiple training data sizes.

With recurrent models, we can choose to vary the number of recurrences for each forward pass while training. This tends to improve generalization to harder tasks at test time and is also referred to as *progressive loss* computation (Bansal et al., 2022). This loss function is a convex combination of the loss values from two forward passes, one with the nominal number of recurrences (so 16 for a 1×16 model) and one with a random smaller number of recurrences.

A.5.2 Addition Models Trained on Varying Data Sizes

Across Figure 8, we see that increasing the size of the operands in the training set allows for better generalization above one hundred digits for all models. This is partially due to the sampling method for training Abacus Embeddings. As the offset randomization hyperparameter $k = 100$ is fixed across experiments, there are more embeddings trained if the operands seen during training are longer. The size of the OOD set below 100 is reduced as the size of the operands seen during training increases, as the ID category now includes this data. However, this does still show that the size of the operands seen during training directly impacts the generalization, with larger training sizes allowing for better generalization.

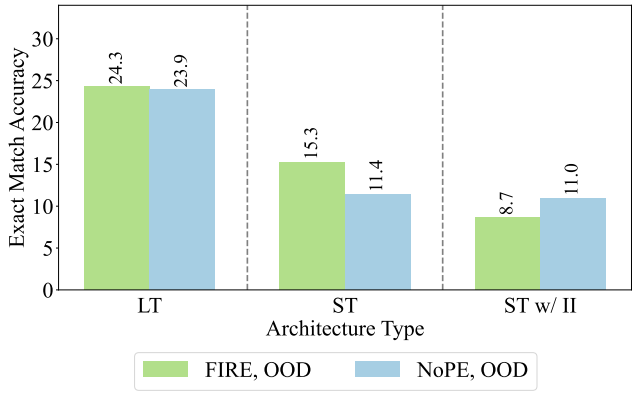

Figure 7: Mean exact match accuracy of three models of effective depth sixteen on size 40 data, varying over NoPE or FIRE embeddings and architectures. Recurrent looped transformer models improve accuracy for addition for both the FIRE and NoPE embeddings.

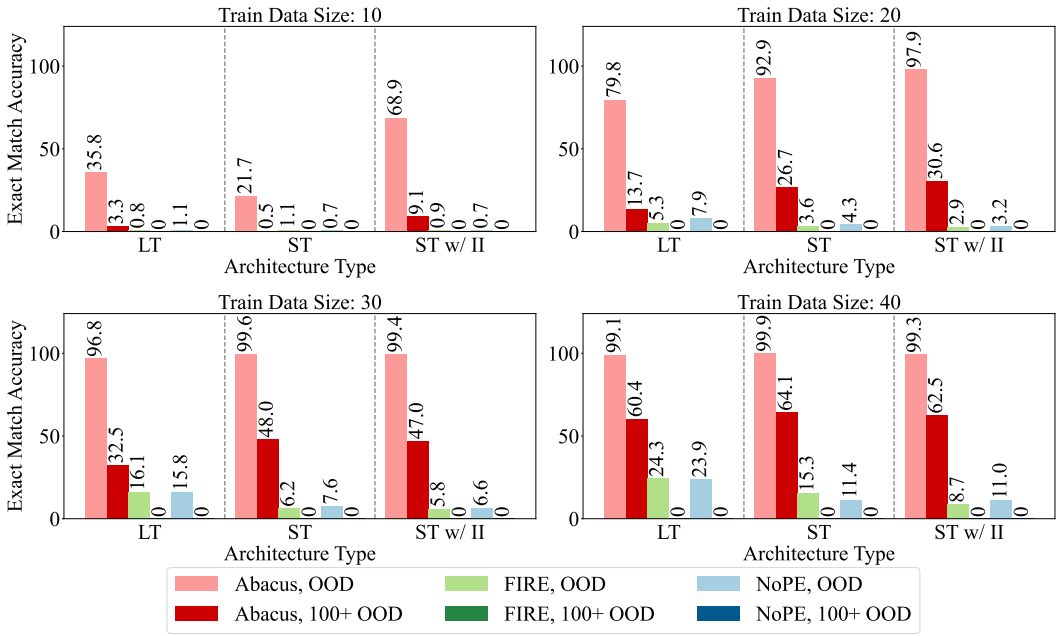

Figure 8: Mean exact match accuracy of three models of effective depth sixteen, varying the training data and architecture. We omit from the plot the in distribution accuracies as these are all 100% or very close to 100% for all models, this can be verified by the dark blue inside of all of the red squares in Section A.5.4. Models trained on larger operands achieve higher OOD accuracy.

A.5.3 Extreme Length Generalization for Addition

Absolute positional embeddings must be learned during training otherwise they are unusable at test time. This limits our Abacus Embeddings which are trained with the offset randomization hyperparameter $k = 100$. One possible way to resolve this generalization problem is to increase the value of k during testing. In Figure 9, we show the exact match accuracy of five looped transformer models, with eight layers in the recurrent block and two recurrences trained on size 20 data with Abacus Embeddings and $k = 101$, generalizing to 120 digit addition. We only show the accuracy for operands of the same length in Figure 9, seeing these models consistently achieve accuracies of 95% and above. We see this across the paper this method is much more robust than that presented by Zhou et al. (2024).

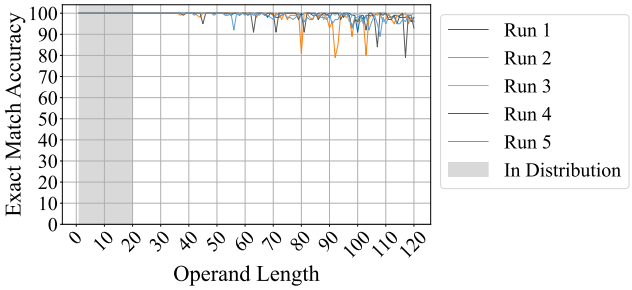

Figure 9: Exact match accuracy of five models trained on size 20 data, generalizing well to 120 digit addition, an extrapolation of $6\times$. Only showing the accuracy for operands of the same length.

A.5.4 Addition Full 100 x 100 Plots

Here we present the mean accuracy as heatmaps for the main addition experiments shown throughout the paper. Figure 10 (left) corresponds to Top Left of Figure 8. Figure 10 (right) corresponds to Top Right of Figure 8 and Figure 2. Figure 11 (left) corresponds to Bottom Left Figure 8. Figure 11 (right) corresponds to Bottom Right Figure 8 and Figure 7. Figure 12 corresponds to Figure 4. All of these figures show the Abacus Embeddings ability to generalize in both dimensions of the addition problem.

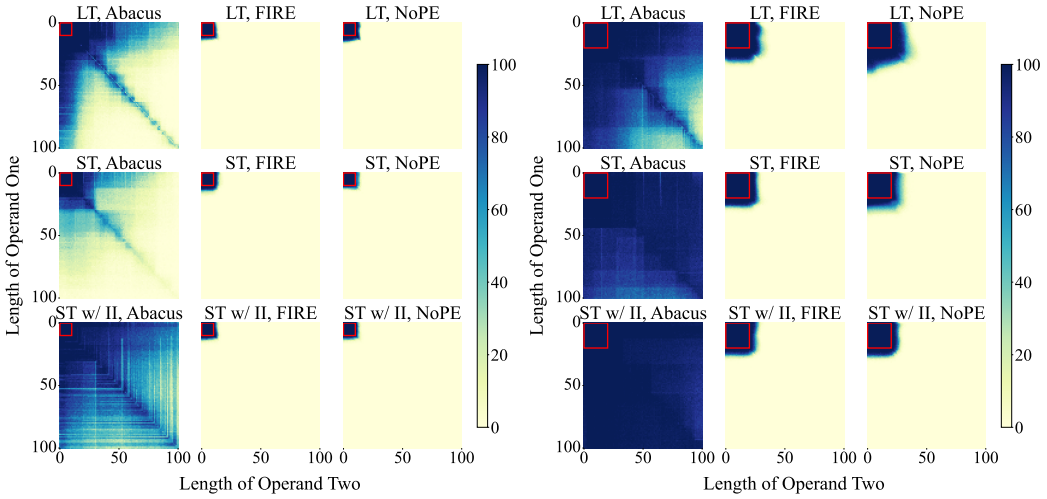

Figure 10: Full 100×100 exact match accuracy plots, taking the mean over three models. **Left:** Size 10 training data, corresponding to Top Left of Figure 8; **Right:** Size 20 training data, corresponding to Top Right of Figure 8 and Figure 2.

A.6 Datasets

Addition: We sample equally, with replacement, from all $i \times i$ possible operand lengths up to the maximum dataset size of 20 million, we call this a dataset of size i in the main text. For evaluation we sample 100 samples for each pair of operand lengths evaluated.

Bitwise OR: The input for this problem is two binary vectors, the longer input vector is all zeros and the shorter input contains a one. The output should be the length of the longer vector with the one in the same position as in the shorter vector. If the inputs are the same length, the one can be in either vector. E.g. $001 \oplus 00000 = 00100$. For training, we exhaustively sample the space of all vectors of sizes less than or equal to the predefined maximum input vector size.

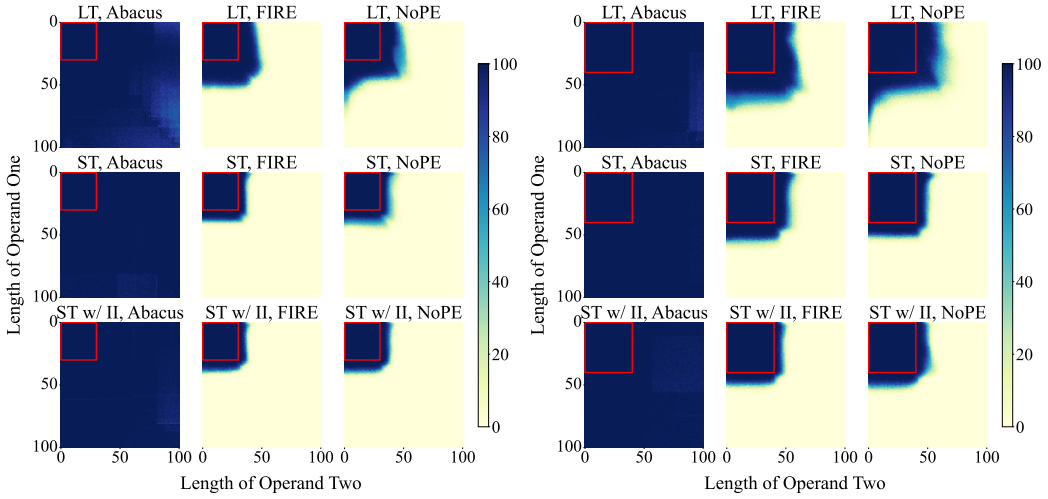

Figure 11: Full 100×100 exact match accuracy plots, taking the mean over three models. **Left:** Size 30 training data, corresponding to Bottom Left Figure 8; **Right:** Size 40 training data, corresponding to Bottom Right Figure 8 and Figure 7.

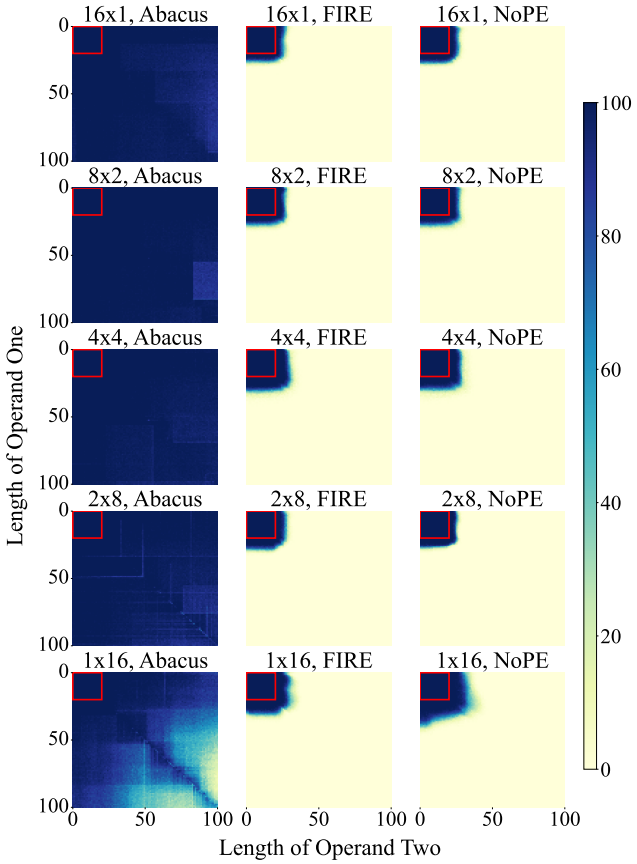

Figure 12: Full 100x100 exact match accuracy plots, taking the mean over three models, relating to Figure 4.

Sorting: Given a list of reversed integers indexed by characters, output the characters in ascending order. E.g. $a : 64957, b : 99963, c : 10218, d : 7141, e : 05781 = d, e, b, a, c$. We implement the sampling process for sorting in a grid like manor. We query each "square" of an $[1, n] \times [1, n]$ grid

until the maximum size has been reached for the dataset. When querying "square" (i, j) we randomly sample i integers of size less than or equal to j digits. We randomly sample consecutive indices for the natural numbers in our list at both train and test time.

Multiplication: We implement the multiplication datasets for both training and testing the exact same manor as for addition, only changing the operation used to calculate the answer.

A.7 Addition Ablations

A.7.1 Analyzing the Intermediate Properties of Recurrence

Thanks to the looped transformer architecture, we can extract intermediate solutions from the models, allowing us to plot the models outputs over iterations of the recurrent block. We present an example in Figure 13 and suggest that this level of interpretability could be leveraged in future work. The model presented is a 1×16 model, one decoder layer and sixteen recurrences. We do not show the full 16 iterations in this plot for readability but these models do maintain a fixed point to 16 iterations and beyond.

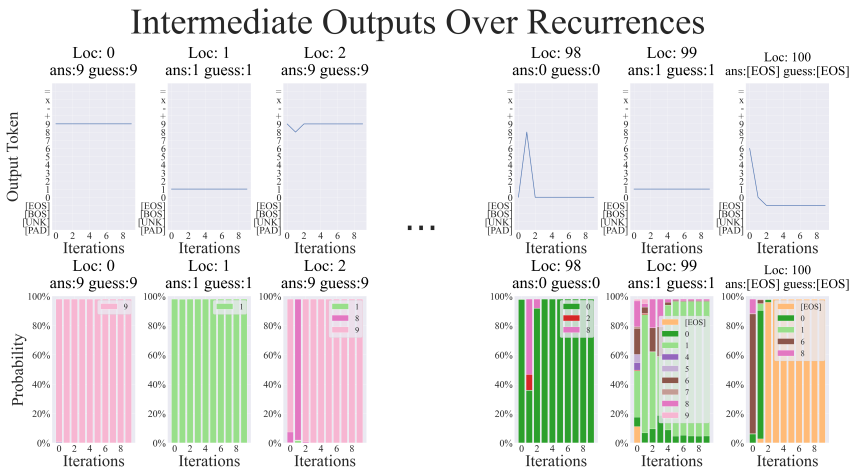

Figure 13: Plot showing the improvement of the prediction over "thinking" iterations on a 100 digit addition problem.
Input Prompt:
587928785434679080355608971949871667189221012941443697496891519051264419888571617
009625529523370283 6+435811039155283076968397848018750172176490052521809790380875 0
786159803668915002036143168815597779644=
Answer:
919576073626374550845911684630020084191658772891994105418527595750262943203928417
5860647426258495700 1[EOS]
(Note that the plot is truncated.)

A.7.2 Removing Masking Before Equals

We mask all tokens before the equals sign in all of our experiments, we hypothesize that with more training time this constraint may be able to be removed. In Figure 14, we show the effect of training with the same amount of flops as the other addition experiments without masking before the equals sign.

A.7.3 Varying Effective Depth

In Figure 15, we present models with effective depths 8 and more than 16, respectively. In Figure 15 (left), we see that the effective depth 8 models under perform the models with 8 layers in the recurrent block and two recurrences shown in Figure 4, demonstrating the benefit of recurrence in this case. We see very high accuracy from all models in Figure 15 (right). Again, the depth 32 recurrent

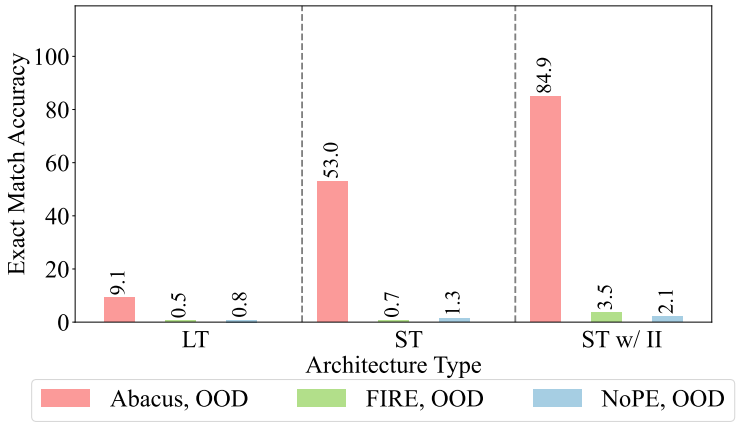

Figure 14: Effect of removing the masking of the loss before the "=" sign in the addition task. All models perform worse when trained for 24 hours on a single Nvidia RTXA4000 if we do not mask the input question in the loss function.

models outperform the standard models with input injection, even though it only has approximately a quarter of the parameters and achieves the highest OOD mean accuracy of all models presented. These ablations show that with Abacus Embeddings the addition task can be learned across many effective depths to varying degrees of accuracy.

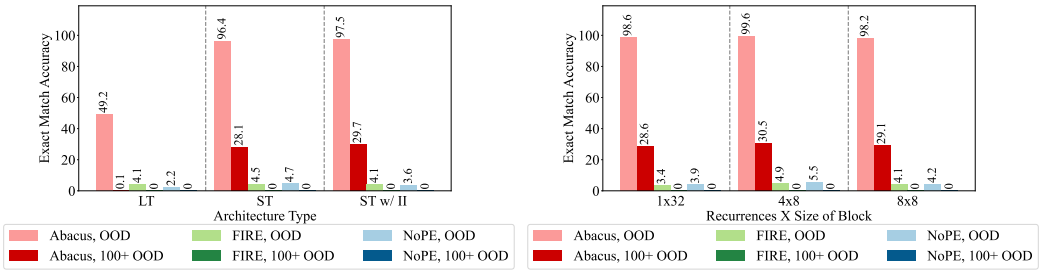

Figure 15: **Left:** Effective depth 8 models, trained on size 20 data. These models under perform the models with eight layers in the recurrent block and two recurrences shown in Figure 4, showing the benefit of recurrence for addition. **Right:** Effective depth >16 models, trained on size 20 data. The models contain many more parameters than all other models we present, showing more that an effective depth of more than 16 does not necessarily improve accuracy in this setting.

In Figure 16 (left), we remove the input injection to the intermediate layers in the recurrent block, only keeping input injection to the first layer of the recurrent block. In Figure 16 (right) we divide the gradients in the recurrent block by the number of recurrences for the looped transformer models during training. We see very minor performance changes for all models shown in Figure 16, with the 2×8 model improving its performance slightly in left plot and the 4×4 model improving slightly in the right plot. We ablate this design choices as we have to remove the input injection inside of the recurrent and divide the gradients in the recurrent block by the number of recurrences for the multiplication models show in Figure 5. Hence, we can conclude there would only be very minor performance changes in this case for addition.

A.7.4 Adding randomized Padding

Abacus Embeddings give strong priors for numerical tasks but without them, looped transformers perform better than the standard transformer architectures we present. The result shown in Figure 17 aligns well with the hypothesis that with fewer priors the looped transformer models are able to generalize better. In this case the priors are reduced as the training data is noised with random pad symbols, a method which was shown to improve length generalization in prior work (Shen et al., 2023).

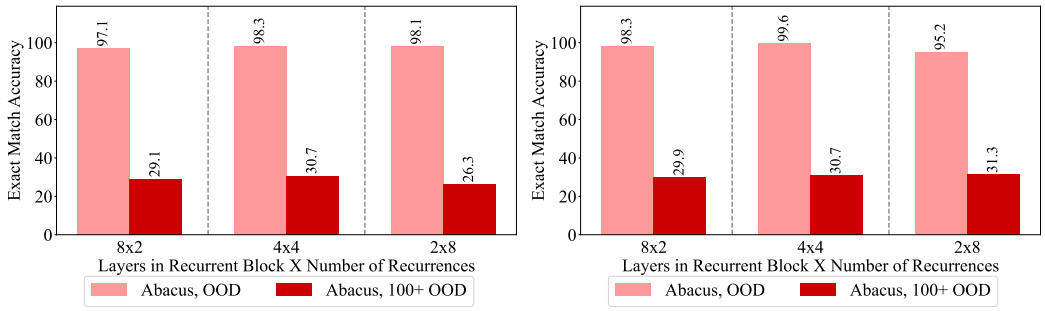

Figure 16: Replicas of the looped transformer models shown in Figure 4, to check the modifications we use to train addition models do not adversarially impact addition training, taking the mean of three models in each case. **Left:** without the input injection to the layers inside of the recurrent block, only to the first layer of the recurrent block. **Right:** dividing the gradients in the recurrent block by the number of recurrences.

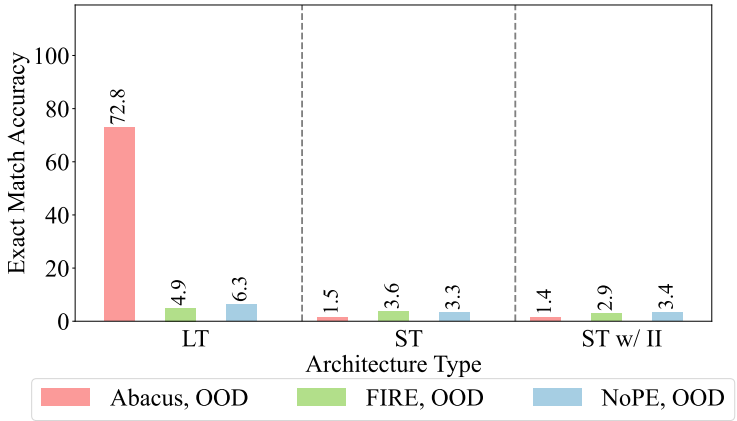

Figure 17: Effect of adding randomized padding into training data only for the addition task. Looped transformer models are able to maintain high accuracy when random padding is added into the data.

A.7.5 Index Hints

Zhou et al. (2023) "randomly sample consecutive index hints from a pre-defined ordered set of hints with 102 symbols," for example $a6b7c5 + a1b6c3 = a7b3c9$. We implement this method two ways. Firstly, cyclic, here we treat the list as cyclic when sampling. Secondly, non-cyclic, this reduces the number of samples which receive the embeddings later in the ordering as we only sample from the list in order. We see similar results for models trained on up to twenty digits as Zhou et al. (2023). We do note that our format of taking the mean exact match accuracy does highlight robustness as if one of the three models tested were to not generalize well, this would impact reported accuracy heavily. We only show a comparison to size 20 training data due to the increased cost of evaluating these index hint models, as the inputs and outputs are approximately double the length of regular questions the inference time is heavily increased. Due to the robustness issues highlighted by Zhou et al. (2024) with their methods, we try to the best of our abilities to faithfully reproduce their work within our experimental set up, noting that perhaps a better random seed or initialization may be able to produce better results for these models.

A.8 Additional Experimental Information

In this work, we consider three different model types, the classical standard transformer, standard transformer with input injection, and looped transformers. We visually describe these in Figure 19. Due to the looped transformer architecture the number of recurrences at train time can be different to the number of recurrences at test time, although we do not make use of this in this work.

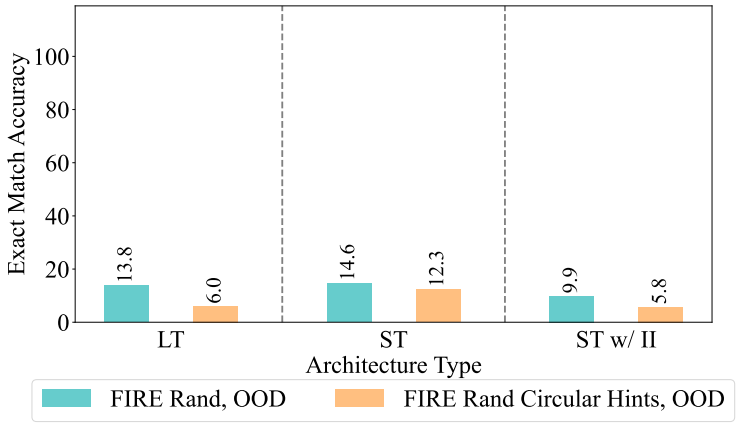

Figure 18: Using index hints and randomized FIRE embeddings, presented by Zhou et al. (2024), training on size 20 data with our methodology, such as masking before the equals sign. This would be comparable to "1 to 20" in Figure 13 presented by Zhou et al. (2024) and Figure 2 of our work.

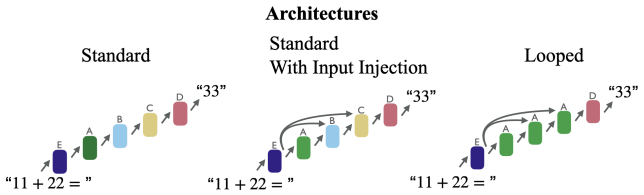

Figure 19: Visualization of the three architectures we study.

As Abacus Embeddings are a variant of absolute embeddings, reused only for numbers, they could be combined with relative embeddings being deployed in current models. If all digits input to the model are tokenized individually, we can perform a linear time operation to find and assign relative embeddings to all numbers in an input, which is lower than the quadratic cost incurred by attention. Training a small number of Abacus Embeddings may be enough to handle all numerical inputs for addition as they are reused. To fully implement our methodology all numbers also have to be reversed, this can be implemented with simple regular expressions on all inputs and outputs.

To facilitate training of many models from scratch, we use a language model cramming setup (Geiping & Goldstein, 2023) and limit each training run to 8 exaFLOP of compute (a single Nvidia RTXA4000 GPU for 24 hours); for multiplication results we allow 64 exaFLOP (eight Nvidia RTXA4000 GPUs for 24 hours). During training, we mask the input question and only compute loss on the answer digits. We use a character level tokenizer for all experiments and greedy decoding in all testing. We train all models with a local batch size which is the maximum batch size that is a power of two that will fit into the sixteen gigabytes of GPU memory. For multiplication models we first take the mean loss across samples before taking the mean across all samples in a batch, instead of taking the mean loss across all token in a batch; we find this leads to slightly more stable training. We note that training models to solve multiplication requires more hyperparameter tuning than addition, perhaps implying it is a trickier task to learn. Also, FIRE models require a much greater compute budget for hyperparameter search as compared to Abacus models for multiplication. In Table 3, we present the approximate parameter counts for models trained with input injection and Abacus Embeddings.

Compute Usage. We detail the default use of GPUs for each experiment in Table 4. For some experiments, such as extreme length generalization (Figure 9) and index hints (Figure 18) more GPU hours are required for testing, these are included in the total number of GPU hours used. Our testing pipeline for addition and Bitise OR uses Nvidia V100 GPUs. Due to a technical problem, 'torch.compile' cannot be used on the V100 GPUs we use, therefore others may be able to reduce this compute time in future studies. All compute was provided by internal resources. During the exploratory phase of this project, we used more GPU hours to test and design the experiments shown,

Table 3: Number of parameters, to the nearest million, in a model with Abacus Embeddings and input injection.

Layers in Recurrent Block	Recurrences	Parameters (Millions)
16	1	122
8	2	64
4	4	34
2	8	19
1	16	12

Table 4: Default number of Nvidia GPU hours used to train a model.

Dataset	Number of GPU Hours (training)	Number of GPU Hours (testing)
Addition	24 - RTXA4000	65.8 - V100
Bitwise OR	1 - RTXA4000	45 - V100
Sorting	24 - RTXA4000	64 - RTXA4000
Multiplication	192 - RTXA4000	0.83 - RTXA4000

using approximately 1.5 terabytes of storage of the entire project. An estimate of the total compute required for all of the results presented in the main paper is $10,039$ GPU hours. The appendix results require a further $18,278$ GPU hours.

A.8.1 Hyperparameters

We detail what we believe to be an important subset of the default hyperparameter values in Table 5. A full list of all hyperparameters and model configurations is contained in the code release. For multiplication models with FIRE embeddings, the learning rate is 0.00006, due to large instabilities in higher learning rates which were not experienced for the Abacus Embeddings.

A.8.2 Code Release

We will release all code and datasets on GitHub with an MIT License.

Table 5: Default hyperparameter values.

Hyperparameter	Default Value
Hidden Size	1024
Intermediate Size	2048
Embedding Size	1024
Number of Attention Heads	16
Progressive Loss Alpha (Bansal et al., 2022)	1.0
Data Type	float16/float32
Optimizer	AdamW (Loshchilov & Hutter, 2017)
Global Batch Size	8192
Batch Size Ramp	0.6
Learning Rate	0.0001
Learning Rate Scheduler	Trapezoid (Zhai et al., 2022)
Activation Function	GELUglu (Shazeer, 2020)
Normalization Layer	LayerNorm (Ba et al., 2016)
Normalization Type	Post
Offset Randomization Hyperparameter (k)	100
Initialization	Deepnorm (Wang et al., 2022)