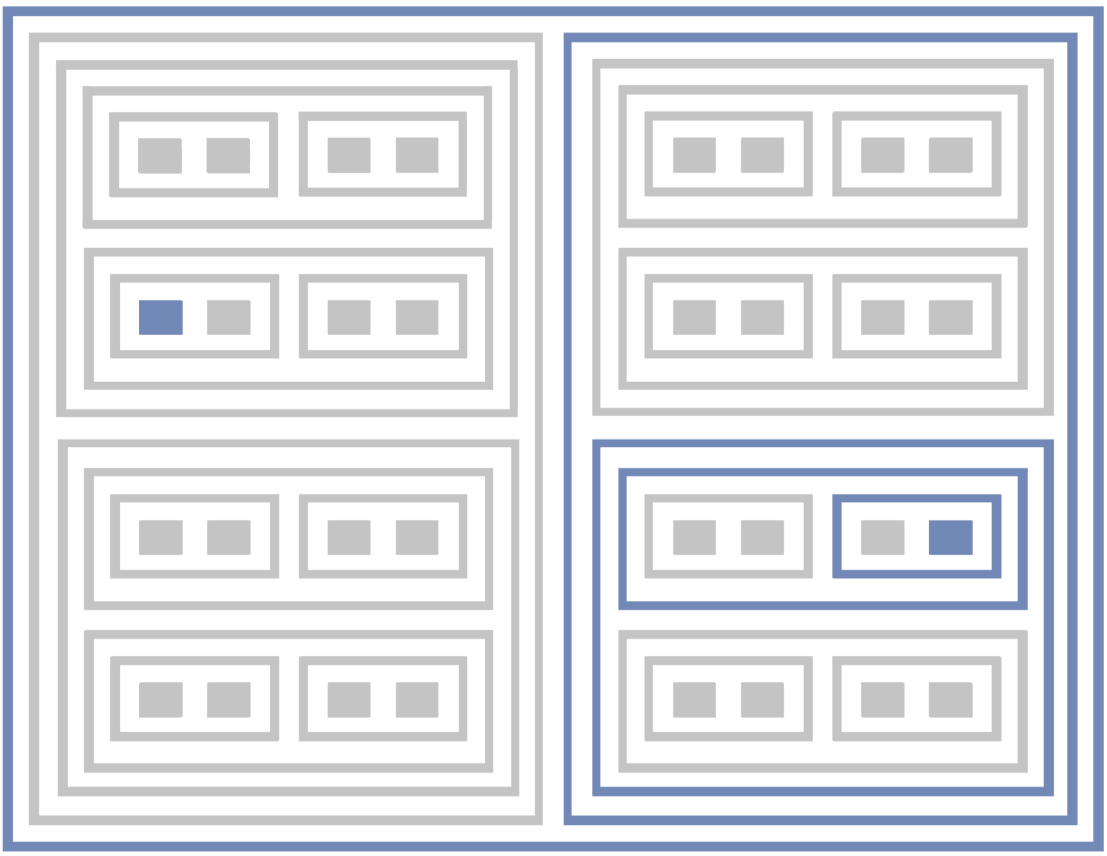
Inventions No. 2
sci-sci.org/virtual-hensel
DOI: 10.5281/zenodo.17026958

SciSci Research
サイサイ・リサーチ

Future Computing
コンフュー



Virtual Hensel (バーチャル・ヘンゼル)

A Demonstration of Exact Computing with 2-adic Arithmetic

James Douglas Boyd

SciSci Research, Inc. + Future Computing

SciSci Research, Inc.
Boulder, Colorado, United States
www.sci–sci.org

Citation Format:

Boyd, J.D. (2025). Virtual Hensel (バーチャル・ヘンゼル): A Demonstration of Exact Computing with 2-adic Arithmetic. SciSci Inventions, 1(2). DOI: 10.5281/zenodo.17026958

Contents

1 The Virtual Hensel

This report is a resource to accompany a forthcoming public demonstration of Virtual Hensel I, which is to be the first demo of a virtual machine based on the Hensel CPU architecture. The Hensel CPU architecture, designed to perform exact arithmetic, presents an alternative to floating-point computing, which, by comparison, is but approximate. Exact arithmetic is performed in the field $\mathbb{Q}_2$, i.e., the field of 2-adic numbers, rather than $\mathbb{R}$. The original Hensel CPU report gives an introductory – albeit notation-heavy – description of coding standards for 2-adic operands and instructions, and a sketch of an architecture for efficient 2-adic computations. Functioning as a proof-of-principle via *in silico* emulation of this architecture, Virtual Hensel I gives a first demonstration of the realizability of $\mathbb{Q}_2$-based exact computing.

Being of modest demonstrative capability, the Virtual Hensel I can perform exact arithmetic on just over $50,000$ operands. That is to say, all $50,000$ have FC-3-2025 encodings (with FC-3-2025 being a new standard for χ-IDs introduced in this report), and can be loaded to the 32 processor registers in the Virtual Hensel I processor cluster. The forthcoming Virtual Hensel I demo illustrates explicitly how operands are loaded to processor registers and how instructions are executed by 2-adic arithmetic logic units (2AALUs).

1.1 Report Scope

Necessarily building upon the architectural description provided in the original report, this report is written in a manner avoidant of undue repetition and redundancy, though at times recapitulating (in sparser detail) crucial descriptions given in the original in order to provide background for introducing new developments. For instance, an extensive review of FC-1-2025 operand encodings (including the R_{FC}, L_{FC}, and T_{FC} blocks of the encoding) will not be subject to elaboration here. Nonetheless, a light recapitulation of FC-1-2025 is in order so as to provide a premise for introducing FC-3-2025. So far as descriptive content is concerned, priority is given to that which was left wanting in the original report, particularly concerning the χ-ID system and load-store architecture, each of which, despite being subject to nontrivial discussion in the original report, has enjoyed improvement as the task of realizing a proof-of-principle for general exact computing with the Hensel architecture was undertaken in building Virtual Hensel I.

Following the updates on χ-IDs and load-store, the report will proceed with a description of Virtual Hensel I itself, with particular attention paid to the processor, covering processor register addresses and locations in the cluster, as well as 2AALU execution of arithmetic instructions. These descriptions will be complemented by both worked examples and Virtual Hensel I visualizations. The report concludes with wider analysis of the implications of the Virtual Hensel I proof-of-principle for the more general prospect of exact computing with the Hensel architecture. With Virtual Hensel I being of modest performance, this section focuses particularly on prospective scaling properties, including the scalability of processor operand capacity, the arithmetic reach of processor operations, and the cost-savings of exact computation relative to floating-point.

Those desiring an overall, gentle exposition on the Virtual Hensel and Hensel CPU architecture can refer to the forthcoming demo exposition video, to be released by SciSci Research and Future Computing in due course. This report includes stylized illustrations from the Virtual Hensel I demo, and thus, "snapshots" of the kind of content to be presented in the video. The demo video itself will be largely expository in nature, intended to provide an accessible introduction to exact computing. Thus, following the release of the demo video, this present report may be of interest as a deeper resource for those seeking a technical reference on Virtual Hensel I.

2 How the Virtual Hensel Works

2.1 FC-3-2025 Encodings

The Virtual Hensel's load-store architecture and arithmetic operations are designed for operands encoded according to FC-3-2025, an encoding standard for χ-IDs introduced herein. Compared to the FC-1-2025 standard from the original report, the distinguishing features of FC-3-2025 are but slight. FC-1-2025 encoded the coefficients of 2-adic expansions until reaching a repetitive subsequence. For instance, a 2-adic number $\overline{01}1101.11_2$ is encoded by FC-1-2025 as $(\perp, 0, 1, \perp, 1, 1, 0, 1, \perp, 1, 1)$, where $(0, 1)$ belongs to the repetition block R_{FC}, $(1, 1, 0, 1)$ belongs to the decimal-left block L_{FC}, and $(1, 1)$ belongs to the decimal-right block T_{FC}. The advent of FC-3-2025 was the insight that these blocks are additive: the 2-adic expansion giving $\overline{01}1101.11_2$ is just the sum of the individual separate expansions giving $\overline{01}$, $1101._2$, and $.11_2$, or, in terms of FC-1-2025, a merging of the blocks $(\perp, 0, 1, \perp, \perp)$, $(\perp, \perp, 1, 1, 0, 1, \perp)$, and $(\perp, \perp, \perp, 1, 1)$. Thus, the FC-3-2025 standard generates encodings from individual blocks, which are treated as primitives.

A key consequence of this construction is that FC-3-2025 does not necessarily include all coefficient terms in the 2-adic expansion before the repetitive subsequence, resulting in encodings that differ from FC-1-2025. Consider, for instance, $\frac{4}{3}$, which is but $2 - \frac{2}{3}$. The FC-1-2025 encoding is $(\perp, 0, 1, \perp, 1, 0, 0, \perp)$, because the 2-adic expansion is $2^2 + 2^3 + 2^5 + 2^7 \ldots$, giving $\overline{01}100._2$. However, the FC-1-2025 encoding for $-\frac{2}{3}$ is but the single block $(\perp, 1, 0, \perp, \perp)$, and the FC-1-2025 encoding for 2 is but the single block $(\perp, \perp, 1, 0, \perp)$. Thus, the FC-3-2025 merges these two FC-1-2025 blocks, as primitives, just as one would add 2 and $-\frac{2}{3}$. So, the FC-3-2025 encoding is $(\perp, 1, 0, \perp, 1, 0, \perp)$. A more rigorous definition of FC-3-2025 can be given with reference to the χ-ID system, the topic of the next subsection.

2.2 The χ-ID System

The Hensel CPU architecture endows the processor with a cluster of register processors and 2AALUs. The cluster, possessing a nested structure described in the original report, facilitates operand loading and arithmetic operations in a manner tailored to efficient 2-adic computation. Register processors are assigned specific addresses, such as $(1, 1, 0, 1)$ or $(1, 1)$ which match FC-3-2025 block primitives. Thus, an FC-3-2025 encoding for an operand built from block primitives is loaded to the processor by activating the processor registers with addresses matching these blocks. Operands are thereby loaded in distributed fashion, with the individual blocks that give their FC-3-2025 encodings each loaded to a distinct processor register with the appropriate address. Matching is facilitated by treating these block primitives as IDs to be matched with processor register addresses. These are the so-called χ-IDs; they are the coefficient sequences encoded in FC-3-2025 primitives, which, matched against processor register addresses, permit distributed loading of operands to the Hensel processor.

Thus, the Hensel CPU loads operands according to the χ-ID system, as described herein. For a given operand q, the ID $\chi(q)$ may consist of several components. A primitive ID χ^p encodes a 2-adic number given by but a single R_{FC}, L_{FC}, or T_{FC} block. For instance, $FC(1)$ is encoded as $L_{FC(1)} = (1)$. $FC\left(\frac{1}{2}\right)$ is encoded as $T_{FC(\frac{1}{2})} = (1)$. $FC\left(-\frac{1}{3}\right)$ is encoded as $R_{FC(-\frac{1}{3})} = (0, 1)$. The thing to emphasize is that encoding each of these numbers requires only a single R_{FC}, L_{FC}, or T_{FC} block; none requires multiple blocks (e.g., both L_{FC} and R_{FC} blocks). Isolating these R_{FC}, L_{FC}, or T_{FC} blocks, one obtains χ^p primitives: (1) is a primitive, and so too is $(0, 1)$. Hensel load-store matches χ^p against processor register addresses in the processor cluster.

χ^p primitives can then be merged – and their encoded 2-adic expansions thereby added – to obtain encodings for other operands;

this is how FC-3-2025 encodings for $> 50,000$ operands for Virtual Hensel I were algorithmically generated. For instance, as $-\frac{1}{5} + 1 = \frac{4}{5}$, the encoding for $\mathsf{FC}(\frac{4}{5})$ can be obtained by merging the $\chi^{\mathfrak{p}}$ for $\mathsf{FC}\left(-\frac{1}{5}\right)$ and $\mathsf{FC}(1)$: the encoding is $\mathsf{FC}(\frac{4}{5}) = (\perp, (0,0,1,1), \perp, (1), \perp, ())$. It is said that $\mathsf{FC}\left(\frac{4}{5}\right)$ has a compound ID, $\chi^{\mathfrak{c}}$, built from primitive IDs $\chi^{\mathfrak{p}}$.

A given $\chi^{\mathfrak{c}}$ can, in turn, be decompounded into constituent R_{FC}, L_{FC}, or T_{FC} blocks. These are written as $\chi^{\mathfrak{d}}_{(*,-,-)}$ (i.e., the R_{FC} block), $\chi^{\mathfrak{d}}_{(-,*,-)}$ (i.e., the L_{FC} block), or $\chi^{\mathfrak{d}}_{(-,-,*)}$ (i.e., the T_{FC} block), where, for specific encodings, we replace $*$ with the length of the block. Thus, an FC-3-2025 encoding for an operand q is defined as follows:

$$\mathsf{FC}(\mathsf{q}) := \left(\perp, \chi^{\mathfrak{d}}_{(*,-,-)}, \perp, \chi^{\mathfrak{d}}_{(-,*,-)}, \perp, \chi^{\mathfrak{d}}_{(-,-,*)}\right) \quad (1)$$

where $\chi^{\mathfrak{d}}_{(*,-,-)} = \mathsf{FC}(\eta_{\mathsf{A}})$, $\chi^{\mathfrak{d}}_{(-,*,-)} = \mathsf{FC}(\eta_{\mathsf{B}})$, $\chi^{\mathfrak{d}}_{(-,-,*)} = \mathsf{FC}(\eta_{\mathsf{C}})$, and $\eta_{\mathsf{A}} + \eta_{\mathsf{B}} + \eta_{\mathsf{C}} = \mathsf{q}$, the η being 2-adic expansions. The Virtual Hensel I demo supports computations on operands with compound IDs up to a $\chi^{\mathfrak{c}}_{(5,5,5)}$ encoding ceiling: there are over $50,000$ such operands.

Generating operand IDs algorithmically according to compound construction is efficient and amenable to scaling. Furthermore, it respects the arithmeticity of operands by design; that is to say, one obtains a large number of operand pairs whose sum or product (or additive or multiplicative inverse) is also an operand encoded within the $\chi^{\mathfrak{c}}_{(5,5,5)}$ ceiling. That $\chi^{\mathfrak{c}}_{(5,5,5)}$ is selected as the ceiling owes to the nest depth of the Virtual Hensel processor. Its cluster, Ξ^{virt}_5, is of nest depth 5 (+2). Thus, it contains $2^5 = 32$ processor registers, which can thus match with $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$, $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$, or $\chi^{\mathfrak{d}}_{(-,-,\mathsf{n})}$, with $\mathsf{n} \leq 5$.

One might note that this constructive approach to generating IDs will necessarily yield multiple IDs for the same operand. For instance, if one constructs a $\chi^{\mathfrak{c}}$ with the $\chi^{\mathfrak{d}}$ for some operand q as well as the $\chi^{\mathfrak{d}}$ for both $\mathsf{FC}(-1)$ and $\mathsf{FC}(1)$, the resulting $\chi^{\mathfrak{c}}$ merely encodes q, because, additively speaking, the -1 and 1 cancel each other out; one could have just used $\left((), (), \chi_{(-,-,\mathsf{n})}(\mathsf{FC}(\mathsf{q}))\right)$, rather than $\left(\chi_{(2,-,-)}(\mathsf{FC}(-1))\right)$, $\left(\chi_{(-,1,-)}(\mathsf{FC}(1))\right)$, and $\left(\chi_{(-,-,\mathsf{n})}(\mathsf{FC}(\mathsf{q}))\right)$ together. However, such superfluity can be discarded by filtering out generated IDs for minimality. χ-IDs are always built from sums of $\chi^{\mathfrak{p}}$ (rather than $\chi^{\mathfrak{c}}$), and minimize both block length and the number of primitives included. That is to say, one selects the FC-3-2025 encoding $\chi^{\mathfrak{c}}_{(\mathsf{min}(\mathsf{n}),\mathsf{min}(\mathsf{m}),\mathsf{min}(\mathsf{l}))}$, where, when permissible, $\mathsf{min}(-) = 0$.

2.3 Load-Store

2.3.1 Loading

The load-store unit (LSU) loads operands to processor registers (PRs) in the processor cluster. Each PR has a unique address, and is loaded with χ-IDs that match. For instance, suppose that the address $\mathfrak{A}$ of a clustered processor register $\mathcal{C}_{\mathsf{PR}}$ is $\mathfrak{A}(\mathcal{C}_{\mathsf{PR}}) = (1,1)$. The PR can be loaded, for instance, with an operand with ID $\chi^{\mathfrak{d}}_{(-,-,2)} = (1,1)$ (i.e., $\frac{3}{4}$) because there is a χ-to-$\mathfrak{A}$ match. The processor register is loaded by being activated with a π-sequence $\pi_{(0,0,1)}$ (where $(0,0,1)$ corresponds to $(-,-,*)$, indicating that the χ-ID matched is a $\chi^{\mathfrak{d}}_{(-,-,*)}$-ID). Thus, we write the loaded PR by address and activation sequence; in this case, it is $\mathfrak{A}^{(1,1)}_{(0,0,1)}$. The information encoded in π-sequences is crucial for recompounding the individual $\chi^{\mathfrak{d}}$-IDs back into a compound ID $\chi^{\mathfrak{c}}$ following a computation for storage. Thus, an operand is loaded in distributed fashion across processor registers in the processor cluster by activating PRs whose addresses match the $\chi^{\mathfrak{d}}$-IDs that compose the FC-3-2025 encoding of the operand.

Ξ^{virt}_5 contains 32 different $\mathcal{C}_{\mathsf{PR}}$, each with a unique address $\mathfrak{A}(\mathcal{C}_{\mathsf{PR}})$. Two addresses have one nontrivial entry: $(0,0,0,0,0)$ and $(1,0,0,0,0,0)$. Two have two nontrivial entries: $(0,1,0,0,0)$ and $(1,1,0,0,0)$. Four have three nontrivial entries: $(0,0,1,0,0)$, $(1,0,1,0,0)$, $(0,1,1,0,0)$, $(1,1,1,0,0)$. Eight ad-

dresses have four: $(0,0,0,1,0)$, $(1,0,0,1,0)$, $(0,1,0,1,0)$, $(1,1,0,1,0)$, $(0,0,1,1,0)$, $(1,0,1,1,0)$, $(0,1,1,1,0)$, and $(1,1,1,1,0)$. Sixteen of the addresses have five nontrivial entries: $(0,0,0,0,1)$, $(1,0,0,0,1)$, $(0,1,0,0,1)$, $(1,1,0,0,1)$, $(0,0,1,0,1)$, $(1,0,1,0,1)$, $(0,1,1,0,1)$, $(1,1,1,0,1)$, $(0,0,0,1,1)$, $(1,0,0,1,1)$, $(0,1,0,1,1)$, $(1,1,0,1,1)$, $(0,0,1,1,1)$, $(1,0,1,1,1)$, $(0,1,1,1,1)$, and $(1,1,1,1,1)$. These alone, thanks to a number of efficiency tricks ascertained and deployed in the development of the Virtual Hensel, can handle over $50,000$ distinct operands.

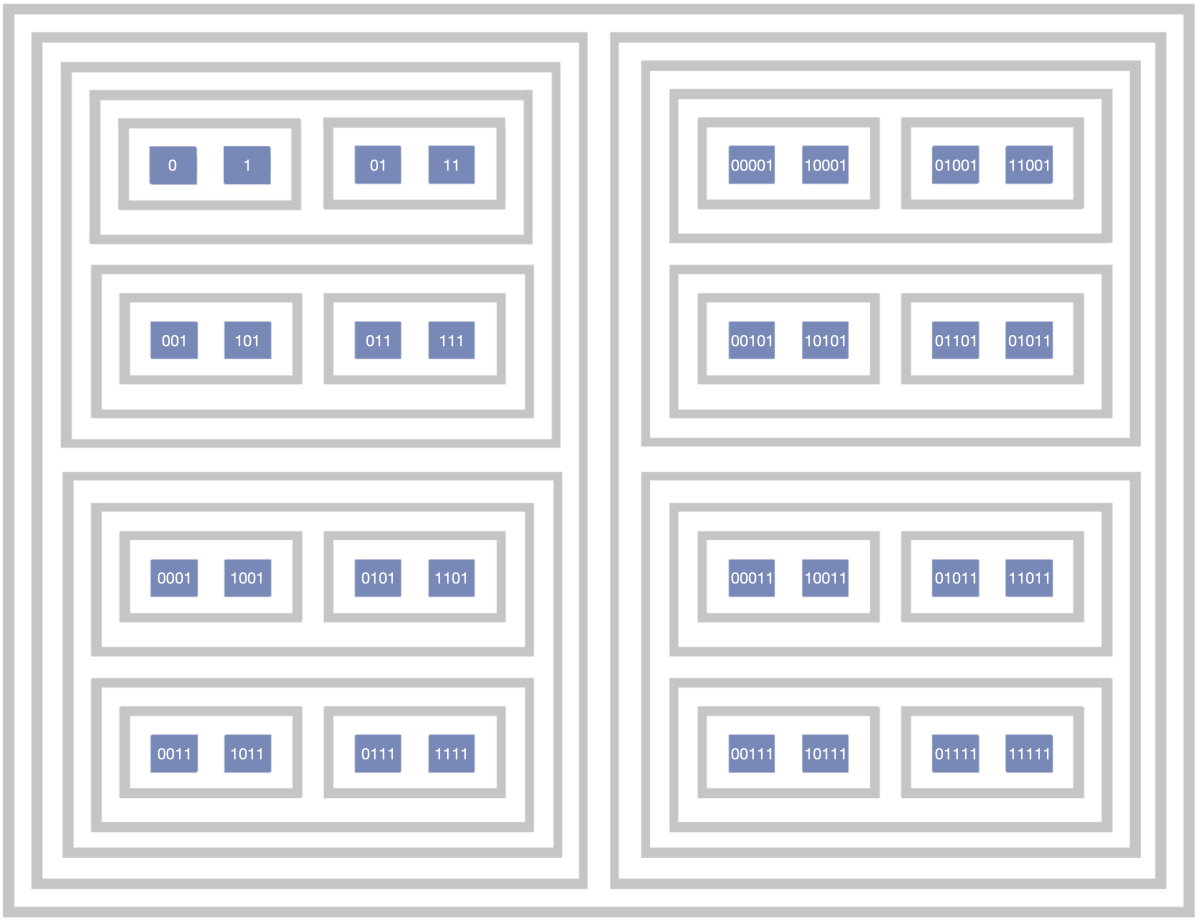


Figure 1: A labeled illustration of the 32 addresses $\mathfrak{A}(\mathcal{C}_{\mathrm{PR}})$ addresses for the cluster processor registers in the Virtual Hensel. The blue entities visualize register processors carriers, and the grey enclosures visualize 2AALU carriers and packaging. (Trivial entries in addresses are omitted; for instance, $(1,0,0,0,0)$ is labeled merely as (1).)

As one will quickly notice when generating FC-3-2025 encodings for operands, there are indeed $\mathsf{FC}(\mathsf{q})$ operands whose $\chi^{\mathfrak{d}}$ do not match any of the 32 $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$ given above. Nonetheless, with a few tricks, all operands up to $\chi^{\mathfrak{c}}{}_{(5,5,5)}$ can nonetheless be made compatible with Ξ_5^{virt} load-store; they are Ξ_5^{virt}-loadable.

The first trick is as follows. Whereas a $\chi^{\mathfrak{d}}_{(-,-,\mathsf{n})}$-ID (for $1 \leq \mathsf{n} \leq 5$) always matches one of the 32 $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$, one finds that $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$-IDs must be reversed, as they read from right to left. Thus, one applies a reverse operator $\mathfrak{r}$ to these and matches $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}\right)$ with the appropriate $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$. Of course, $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-IDs also read from right to left, but a different trick is deployed for them.

This brings us to the second trick. The key difference between $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$ and $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-IDs is that the latter are more numerous than the former. For instance, whereas one will not find a $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$-ID with block entries $(0,1,1)$ (or, when reversed, $(1,1,0)$) – the 0 entry being superfluous – one does find a $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-ID with block entries $(0,1,1)$. After all, in the case of the $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-ID, encoding the repetitive block R_{FC} in the FC-1-2025 encoding, 0 entries are not superfluous: all entries in a repetitive subsequence are needed. Thus, merely reversing $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-IDs does not rectify the address-matching issue. The issue is that $\chi^{\mathfrak{d}}_{(\mathsf{n},--)}$-IDs (for $2 \leq \mathsf{n} \leq 5$) beginning on the right with 0 cannot be loaded (for there are no $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$ to match). Nonetheless, the IDs can be loaded if they are shifted so that they begin on the right with a 1 term. This is permissible, given that these IDs encode repeated sequences, so long as one keeps track of the shift term ς, which amounts to a term subtracted from the repeated sequence. This term will be encoded with its own χ-ID. Because it is integer-valued, its χ-ID is always a $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$ block.

A load task for an operand q can be decomposed into up to four distinct loads λ, with up to four distinct $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$ destinations, which we'll write as $\mathfrak{A}\left(\mathcal{C}^{\mathsf{A}}_{\mathsf{PR}}\right)$, $\mathfrak{A}\left(\mathcal{C}^{\mathsf{B}}_{\mathsf{PR}}\right)$, $\mathfrak{A}\left(\mathcal{C}^{\mathsf{C}}_{\mathsf{PR}}\right)$, and $\mathfrak{A}\left(\mathcal{C}^{\mathsf{D}}_{\mathsf{PR}}\right)$. The loads are written as follows:

$$\lambda^{\mathsf{A}}(\mathsf{FC}(\mathsf{q})) : \left(\varsigma\left(\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}(\mathsf{FC}(\mathsf{q}))\right),(*,-,-)\right) \rightarrow \mathcal{C}^{\mathsf{A}}_{\mathsf{PR}}\left(\mathfrak{A}^{\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}}_{(*,-,-)}\right) \quad (2)$$

$$\lambda^{\mathsf{B}}(\mathsf{FC}(\mathsf{q})) : \left(\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}(\mathsf{FC}(\mathsf{q}))\right),(-,*,-)\right) \rightarrow \mathcal{C}^{\mathsf{B}}_{\mathsf{PR}}\left(\mathfrak{A}^{\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}}_{(-,*,-)}\right) \quad (3)$$

$$\lambda^{\mathsf{C}}(\mathsf{FC}(\mathsf{q})) : \left(\left(\chi^{\mathfrak{d}}_{(-,-,\mathsf{n})}(\mathsf{FC}(\mathsf{q}))\right),(-,-,*)\right) \rightarrow \mathcal{C}^{\mathsf{C}}_{\mathsf{PR}}\left(\mathfrak{A}^{\chi^{\mathfrak{d}}_{(-,-,\mathsf{n})}}_{(-,-,*)}\right) \quad (4)$$

and

$$\lambda^{\mathsf{D}}(\mathsf{FC}(\mathsf{q})) : \left(\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}(\mathsf{FC}(\mathsf{q}))\right),(-,*,-)\right) \rightarrow \mathcal{C}^{\mathsf{D}}_{\mathsf{PR}}\left(\mathfrak{A}^{\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}}_{(-,-,-)}\right) \quad (5)$$

Consider the example of $\frac{13}{6}$. It's the sum of $-\frac{1}{3}$, $\frac{1}{2}$, and 2, all of which have $\chi^{\mathfrak{p}}$ FC-3-2025 encodings; thus, its compound ID, $\chi^{\mathfrak{c}}(\mathsf{FC}(\frac{13}{6})) = (\perp,(0,1),\perp,(1,0),\perp,(1))$, is decompounded into $\chi^{\mathfrak{d}}_{(2,-,-)} = (0,1)$, $\chi^{\mathfrak{d}}_{(-,2,-)} = (1,0)$, and $\chi^{\mathfrak{d}}_{(-,-,1)} = (1)$. In this case, $\chi^{\mathfrak{d}}_{(-,2,-)} = (1,0)$ doesn't match any of the 32 $\mathfrak{A}\left(\mathcal{C}_{\mathsf{PR}}\right)$, but $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,2,-)}\right) = (0,1)$ does. (Note that $\chi^{\mathfrak{d}}$ are right-padded to length five for $\mathfrak{A}$-matching, such that $(0,1)$ is treated as $(0,1,0,0,0)$.) As for $\left(\chi^{\mathfrak{d}}_{(2,-,-)}\right) = (0,1)$, there is no need for a ς-shift, as it begins on the right with a 1 term. Thus, the loads are $\lambda^{\mathsf{A}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) : ((0,1),(*,-,-)) \rightarrow \mathcal{C}^{\mathsf{A}}_{\mathsf{PR}}\left(\mathfrak{A}^{(0,1)}_{(*,-,-)}\right)$, $\lambda^{\mathsf{B}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) : ((0,1),(-,*,-)) \rightarrow \mathcal{C}^{\mathsf{B}}_{\mathsf{PR}}\left(\mathfrak{A}^{(0,1)}_{(-,*,-)}\right)$, $\lambda^{\mathsf{C}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) : ((1),(-,-,*)) \rightarrow \mathcal{C}^{\mathsf{C}}_{\mathsf{PR}}\left(\mathfrak{A}^{(1)}_{(-,-,*)}\right)$.

Consider, on the other hand, the operand $\frac{511}{62}$. Here,

$$\chi^{\mathfrak{c}}\left(\mathrm{FC}\left(\frac{511}{62}\right)\right) = (\bot, (0,1,0,0,0), \bot, (1,0,0,0), \bot, (1)) \quad (6)$$

$\chi^{\mathfrak{d}}_{(-,-,1)} = (1)$ matches a PR address, as expected. $\chi^{\mathfrak{d}}_{(-,4,-)} = (1,0,0,0)$ need be reversed to $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,4,-)}\right) = (0,0,0,1)$ to match. $\chi^{\mathfrak{d}}_{(5,-,-)} = (0,1,0,0,0)$ need be shifted, for there is no processor register whose address matches $(0,1,0,0,0)$. There is, however, a processor register whose address matches $(0,0,0,0,1)$; thus, one need only shift $(0,1,0,0,0)$ to the right by three positions (i.e., such that $\chi^{\mathfrak{d}}_{(5,-,-)}(\mathrm{FC}(\frac{511}{62} - \varsigma)) = (0,0,0,0,1)$, where $\chi^{\mathfrak{d}}_{(-,2,-)}(\varsigma) = (1,1)$).

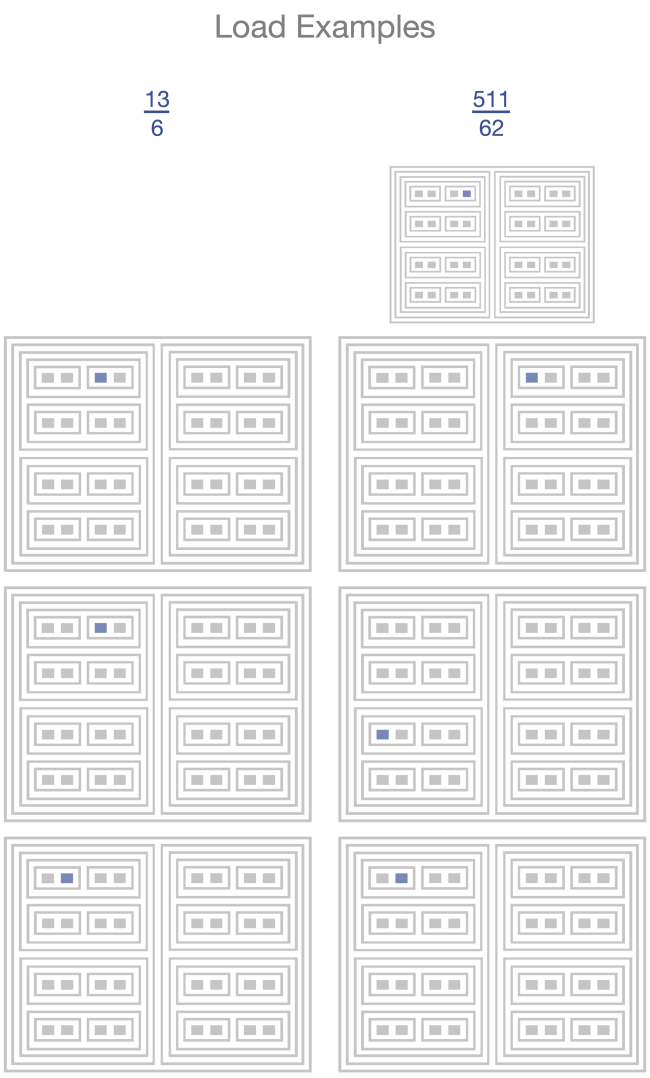


Figure 2: Illustration of the $\mathfrak{A}$ addresses for loads $\lambda^{\mathsf{A}}(\mathrm{FC}(\frac{13}{6}))$, $\lambda^{\mathsf{B}}(\mathrm{FC}(\frac{13}{6}))$, and $\lambda^{\mathsf{C}}(\mathrm{FC}(\frac{13}{6}))$, as well as $\lambda^{\mathsf{A}}(\mathrm{FC}(\frac{511}{62}))$, $\lambda^{\mathsf{B}}(\mathrm{FC}(\frac{511}{62}))$, $\lambda^{\mathsf{C}}(\mathrm{FC}(\frac{511}{62}))$, and $\lambda^{\mathsf{D}}(\mathrm{FC}(\frac{511}{62}))$. The top panel illustrates λ^{D} loads; the second highest, λ^{A}; the second lowest, λ^{B}; and the lowest, λ^{C}.

Let us review. Each operand has an FC-3-2025 encoding which is decomposed into R_{FC}, L_{FC}, and T_{FC} portions (i.e., $\chi^{\mathfrak{d}}$ IDs), which are loaded in distributed fashion to various processor registers in the processor cluster with matching addresses. A given operand can be decomposed and loaded to up to four distinct processor registers, distributed throughout the processor cluster. The processor registers are loaded by being activated with a π-sequence which contains information about the kind of $\chi^{\mathfrak{d}}$-ID being loaded, which is used for re-compounding $\chi^{\mathfrak{d}}$-IDs back to $\chi^{\mathfrak{c}}$-IDs, the latter necessary for storage.

2.3.2 Storing

Following an arithmetic computation (discussed in the next section), the processor returns an output. The output too is loaded to the processor registers, which we'll write as $\mathcal{C}_{PR}^{A'}$, $\mathcal{C}_{PR}^{B'}$, and $\mathcal{C}_{PR}^{C'}$, as well as a possible shift-term-loaded PR, $\mathcal{C}_{PR}^{D'}$. Matched against the addresses for $\mathcal{C}_{PR}^{A'}$, $\mathcal{C}_{PR}^{B'}$, and $\mathcal{C}_{PR}^{C'}$ (and possibly $\mathcal{C}_{PR}^{D'}$) are the $\chi^{\mathfrak{d}}$-IDs for this output. In order for the output to be stored, it must be recompounded back to a $\chi^{\mathfrak{c}}$-ID.

Recompoundment begins when the master PR $\mathcal{M}_{PR}$ receives the three or four $\chi^{\mathfrak{d}}$ from the LSU, which it retrieves from $\mathcal{C}_{PR}^{A'}$, $\mathcal{C}_{PR}^{B'}$, and $\mathcal{C}_{PR}^{C'}$ (and possibly $\mathcal{C}_{PR}^{D'}$). $\mathcal{M}_{PR}$ then sends these to $\mathcal{M}_{2AALU}$, which composes them into an FC-3-2025-encoded operand for LSU-facilitated storage. The master 2AALU performs the following compoundment:

$$\kappa : \left(\varsigma\left(\chi_{\mathcal{C}_{PR}^{A'}}\right), \mathfrak{r}\left(\chi_{\mathcal{C}_{PR}^{B'}}\right), \chi_{\mathcal{C}_{PR}^{C'}}, \chi_{\mathcal{C}_{PR}^{D'}}\right) \rightarrow \left(\chi_{\mathcal{C}_{PR}^{D'}} + \chi_{\mathcal{C}_{PR}^{A'}}, \mathfrak{r}^{-1}\left(\chi_{\mathcal{C}_{PR}^{B'}}\right), \chi_{\mathcal{C}_{PR}^{C'}}\right) \quad (7)$$

where $\mathfrak{r}^{-1}$ is a reverse-reverse operation.

2.4 Computation

2AALUs in the Hensel processor perform arithmetic operations on each $\chi^{\mathfrak{d}}$-ID of an operand encoding, yielding an output with a new collection of $\chi^{\mathfrak{d}}$-IDs. Under the Hensel load-store architecture, these $\chi^{\mathfrak{d}}$-IDs, if different from those for the input operand, must be loadable to distinct processor registers with matching addresses. Thus, the way the Hensel processor performs arithmetic is by modifying (i.e., performing arithmetic on) individual $\chi^{\mathfrak{d}}$-IDs, and then passing the π-sequences from the processor registers with the input-$\chi^{\mathfrak{d}}$-ID-matching-addresses to new ones that match the modified $\chi^{\mathfrak{d}}$-IDs.

The relationship between 2AALUs, arithmetic modifications, processor register positions, and the nested structure of the processor follows straightforwardly from the circuit-level combinational logic according to which 2AALU operations are executed and FC-2-2025 instructions are implemented. This is the topic of the 2AALU report. For now, it suffices to give a rather qualitative description of this relationship, an explanation of which is to be found in the 2AALU report.

With there being five 2AALU levels in Ξ_5^{virt} and 2^5 processor registers, and the $\chi^{\mathfrak{d}}$-IDs being of maximum block length 5, a 2AALU at each level performs a modification on a different entry in the $\chi^{\mathfrak{d}}$-ID, in parallel, with the rightmost entry modified at level $\ell = 6$ and the leftmost entry modified at level $\ell = 2$. Then, at each level where a modification is performed, there is a corresponding entry modification in the $\chi^{\mathfrak{d}}$-ID at the entry position corresponding with the level, with each entry modification in turn changing the output $\chi^{\mathfrak{d}}$-ID, and thus the $\mathcal{C}_{PR}$ to which the $\chi^{\mathfrak{d}}$-ID is to be loaded. With 2AALUs packaged in nested carriers (the processor registers packaged at the innermost level), and with processor register addresses determined by location in the nested processor structure, there is a direct correspondence between $\mathfrak{A}$ values and circuit board locations of PRs. Thus, a 2AALU modifying a single entry in a $\chi^{\mathfrak{d}}$-ID block in turn affects the location in which the address-matching processor register will be located. Moreover, because, in the nested structure, each 2AALU has only one other same-level-ℓ 2AALU neighbor packaged within the same level-$(\ell + 1)$ car-

rier, a modification by a given 2AALU yields an output whose $\mathcal{C}_{\mathsf{PR}}$ location is within the nested carrier packaging of its neighbor. It is for this reason that a modification is at times referred to by the shorthand term "hop"; it yields an output stored in a $\mathcal{C}_{\mathsf{PR}}$ located in the neighboring nested carrier package, such that a modification amounts to "hopping" from one nested carrier package to another. At outermost levels in the nested structure, these hops span over the circuit board, whereas the hops are less distal at innermost levels, as one would expect given the nested nature of carrier packaging.

The $\mathcal{C}_{\mathsf{PR}}$ are surface-mounted to the printed circuit board according to address, such that, for instance, two processor registers which are packaged in the same nested carrier packaging save the innermost 2AALU level differ in address by only one entry (the leftmost entry), and will neighbor one another on the circuit board. If they differ only in a nontrivial entry further to the right, then their nested carrier packaging will differ at a higher level, such that the processor registers will be more distally positioned on the board. Moreover, for any two process registers $\mathcal{C}_{\mathsf{PR}}$ and $\mathcal{C}'_{\mathsf{PR}}$, the number of entries by which the addresses $\mathfrak{A}(\mathcal{C}_{\mathsf{PR}})$ and $\mathfrak{A}(\mathcal{C}'_{\mathsf{PR}})$ differ is the same as the number of levels at which their nested carrier packaging differs. (This is all made clearer by Figure 1, which shows the $\mathfrak{A}$-labels of the 32 $\mathcal{C}_{\mathsf{PR}}$.)

Herein, we will describe 2AALU operations using the high-level shorthand of "hop calculus", eliding underlying combinational logic, and "π-sequence passing" as a higher-level shorthand for the load-store procedure for outputs of 2AALU arithmetic. A more detailed treatment of combinational logic and circuit design can be found in the 2AALU report.

Arithmetic is performed on FC-3-2025-encoded operands according to FC-2-2025 instructions, which guide the 2AALUs by specifying the entries in an operand encoding to be modified, and the 2AALU level at which the modification is to take place. "Hop operations" are a shorthand for 2AALU operations. If the modification at level ℓ is made from 0 to 1, then a forward hop $\mathsf{h}^{\sigma}_{\ell}$ is taken. If the modification is made from 1 to 0, then a backward hop $\mathsf{h}^{\overline{\sigma}}_{\ell}$ is taken.

Hops can occur at any of the five 2AALU levels in Ξ^{virt}_5. According to the processor design given in the original report, the overall processor will have $5\ (+2)$ levels, with the lowest level (i.e., $\ell = 1$) for the $\mathcal{C}_{\mathsf{PR}}$ and the greatest level (i.e., $\ell = 7$) for the $\mathcal{M}_{\mathsf{PR}}$ and $\mathcal{M}_{\mathsf{AALU}}$. The middle five are for 2AALUs. A hop $\mathsf{h}^{\sigma}_{\ell=2}$ is then equivalent to adding $(1,0,0,0,0)$ to a $\chi^{\mathfrak{d}}$; $\mathsf{h}^{\sigma}_{\ell=3}$, adding $(0,1,0,0,0)$; $\mathsf{h}^{\sigma}_{\ell=4}$, adding $(0,0,1,0,0)$; $\mathsf{h}^{\sigma}_{\ell=5}$, adding $(0,0,0,1,0)$; and $\mathsf{h}^{\sigma}_{\ell=6}$, adding $(0,0,0,0,1)$. Likewise, a hop $\mathsf{h}^{\overline{\sigma}}_{\ell=2}$ is equivalent to adding $(-1,0,0,0,0)$ to a $\chi^{\mathfrak{d}}$; $\mathsf{h}^{\overline{\sigma}}_{\ell=3}$, adding $(0,-1,0,0,0)$; $\mathsf{h}^{\overline{\sigma}}_{\ell=4}$, adding $(0,0,-1,0,0)$; $\mathsf{h}^{\overline{\sigma}}_{\ell=5}$, adding $(0,0,0,-1,0)$; and $\mathsf{h}^{\overline{\sigma}}_{\ell=6}$, adding $(0,0,0,0,-1)$. If a $\chi^{\mathfrak{d}}$, such as (1), is of length > 5, it is right-padded in 2AALU arithmetic, such that $(0,1)+(1,0,0,0,0) = (0,1,0,0,0)+(1,0,0,0,0)$, summing to $(1,1)$.

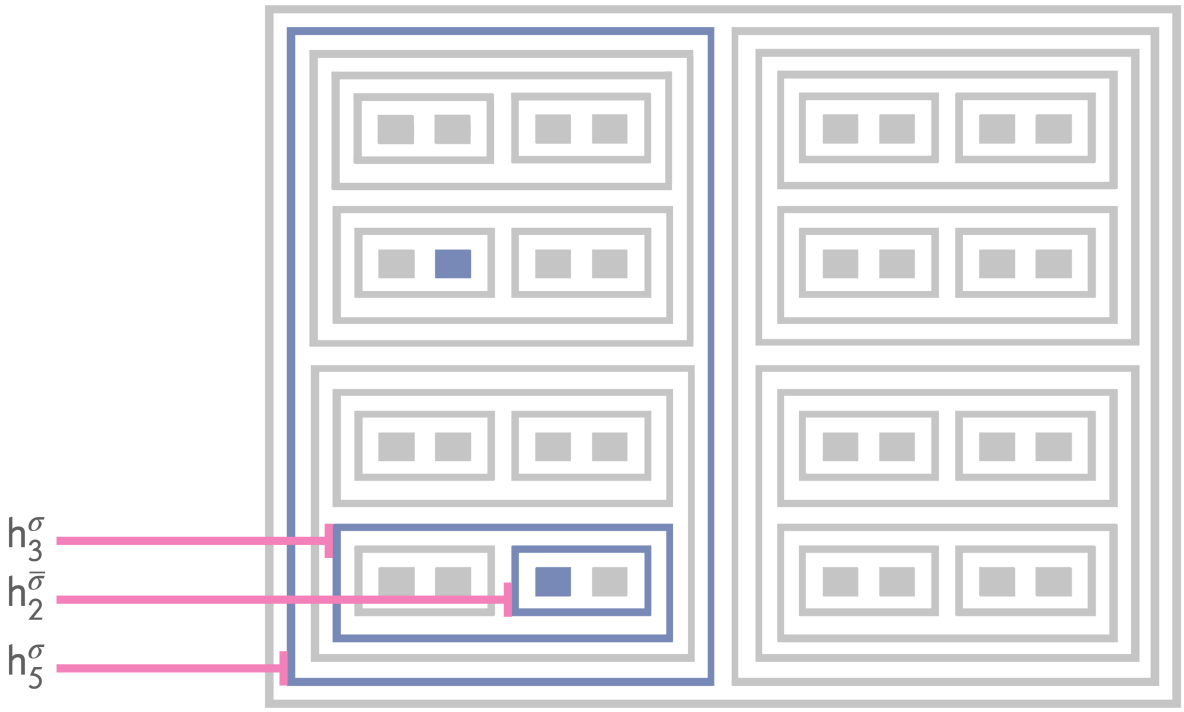


Figure 3: Illustration of hop visualization in the Virtual Hensel demo. A forward hop h_ℓ^σ gives an output located upward or rightward of the input, and a backward hop h_ℓ^σ gives an output upward or leftward of the input.

2.5 A Worked Example

Let's take the example of $\mathrm{FC}\left(\frac{59}{12}\right) + \mathrm{FC}\left(\frac{49}{24}\right)$, which of course yields $\mathrm{FC}\left(\frac{167}{24}\right)$. We can begin all the way at the beginning, with 2-adic expansions. The expansion for $\frac{59}{12}$ is

$$\frac{59}{12} = 2^{-2} + 2 + 2^3 + 2^4 + 2^6 + 2^8 + 2^{10} + 2^{12} \ldots \quad (8)$$

This, in turn, can be thought of as a sum of the expansion for $\frac{1}{4}$ (which is simply 2^{-2}), the expansion for 5 (which is simply $1 + 2^2$), and the expansion for $-\frac{1}{3}$ (which is simply $1 + 2^2 + 2^4 + 2^6 \ldots$). Sure enough, $5 + \frac{1}{4} - \frac{1}{3} = \frac{59}{12}$. Let's next turn to the $\chi^{\mathfrak{p}}$ primitives for each. $\chi^{\mathfrak{p}}\left(\frac{1}{4}\right) = (0,1)$; thus, $\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right) = (0,1)$. Next, $\chi^{\mathfrak{p}}(5) = (1,0,1)$; thus, it is the case that $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{59}{12}\right)\right) = (1,0,1)$. Finally, because $\chi^{\mathfrak{p}}\left(-\frac{1}{3}\right) = (0,1)$, it is the case that $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right) = (0,1)$.

Let's repeat for $\frac{49}{24}$. The expansion is

$$\frac{49}{24} = 2^{-3} + 2^{-2} + 1 + 2 + 2^2 + 2^4 + 2^6 + 2^8 \ldots \quad (9)$$

This is a sum of $\frac{3}{8} = 2^{-3} + 2^{-2}$, $2 = 2^1$, and $-\frac{1}{3} = 1 + 2^2 + 2^4 + 2^6 + 2^8 \ldots$. $\chi^{\mathfrak{p}}\left(\frac{3}{8}\right) = (0,1,1)$; thus, $\chi^{\mathfrak{d}}_{(-,-,3)}\left(\frac{49}{24}\right) = (0,1,1)$. Next, because $\mathfrak{r}(\chi^{\mathfrak{p}}(2)) = (0,1)$, it is the case that $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,2,-)}\left(\frac{49}{24}\right)\right) = (0,1)$. Finally, for R_{FC}, $\chi^{\mathfrak{p}}\left(-\frac{1}{3}\right) = (0,1)$; thus, it is the case that $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{49}{24}\right) = (0,1)$.

Lastly, $\frac{167}{24} = -\frac{2}{3} + 7 + \frac{5}{8}$, so one gets the following: $\chi^{\mathfrak{d}}_{(-,-,3)}\left(\frac{167}{24}\right) = (1,0,1)$, $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{167}{24}\right)\right) = (1,1,1)$, and, finally, $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{167}{24}\right) = (1,0)$, ς-shifted to $(0,1)$.

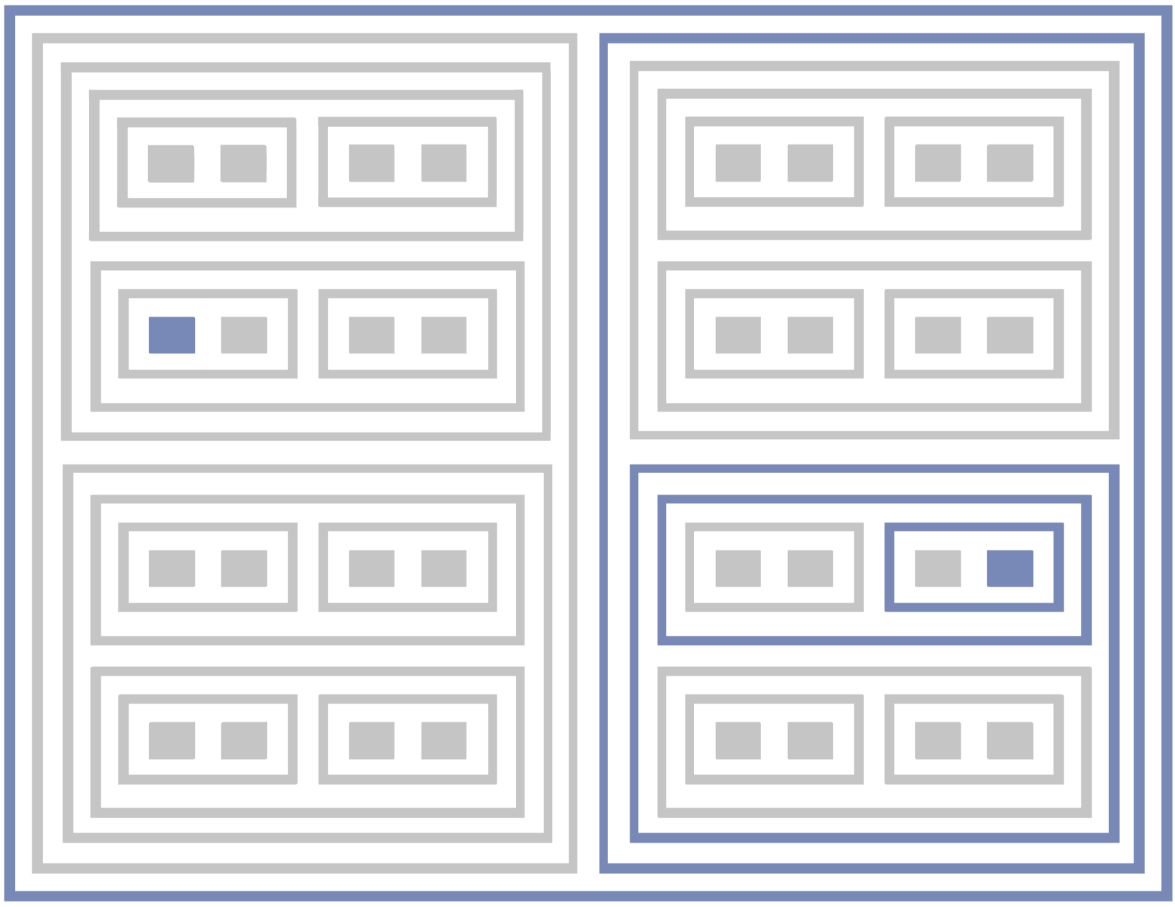


Figure 4: Arithmetic example: $3 + 7 = 10$. The input $\mathrm{FC}(3)$ is loaded to the $\mathcal{C}_{\mathsf{PR}}$ with address $\mathfrak{A}^{(1,0,1)}_{(-,3,-)}$ and the output $\mathrm{FC}(10)$ is loaded to the $\mathcal{C}_{\mathsf{PR}}$ with address $\mathfrak{A}^{(0,1,0,1)}_{(-,4,-)}$ following the computation $\left(\mathsf{h}^{\overline{\sigma}}_{6}, \mathsf{h}^{\sigma}_{5}, \mathsf{h}^{\overline{\sigma}}_{4}\right)$

The computation in Ξ^{virt}_{5} takes $\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right)$, $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{59}{12}\right)\right)$, and $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right)$ as inputs and gives $\chi^{\mathfrak{d}}_{(-,-,3)}\left(\frac{167}{24}\right)$, $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{167}{24}\right)\right)$, and $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{167}{24}\right)$ as outputs.

The arithmetic modification on $\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right)$ (with input $(0,1)$ and output $(1,0,1)$) involves three hops: $\mathsf{h}^{\sigma}_{\ell=4}, \mathsf{h}^{\overline{\sigma}}_{\ell=3}, \mathsf{h}^{\sigma}_{\ell=2}$, which adds $(0,0,1,0,0)$, subtracts $(0,1,0,0,0)$, and adds $(1,0,0,0,0)$, respectively. The modification of $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{59}{12}\right)\right)$ (with input $(1,0,1)$ and output $(1,1,1)$) involves one hop: $\mathsf{h}^{\sigma}_{\ell=3}$. Finally, $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right)$ requires no modification.

One might note the tediousness with which this computation must be written in notion, as compared to the simplicity with which it is captured visually by the Virtual Hensel illustrations.

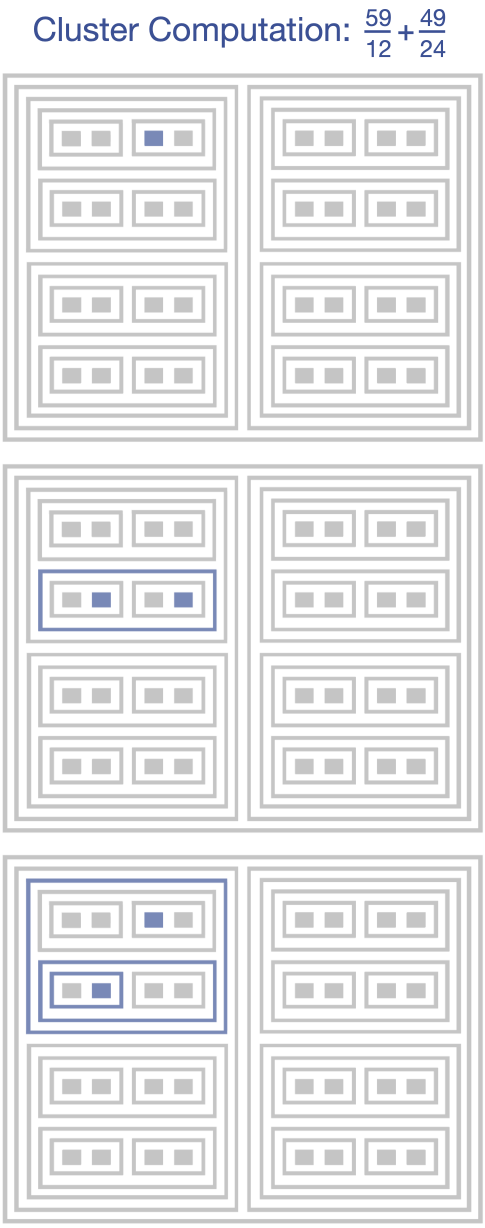


Figure 5: Visual illustration of addition: $\mathrm{FC}\left(\frac{59}{12}\right) + \mathrm{FC}\left(\frac{49}{24}\right)$. The uppermost illustration displays R_{FC} portion arithmetic, which takes $\mathfrak{r}\left(\chi_{(2,-,-)}(\frac{59}{12})\right) = (0,1)$ as input and yields $\mathfrak{r}\left(\chi_{(2,-,-)}(\frac{167}{24})\right) = (0,1)$ as output; thus, the processor register address $\mathfrak{A}^{(0,1)}_{*,-,-}\left(\mathcal{C}_{\mathsf{PR}}\right)$ for the input and output is the same. The middle illustration displays the L_{FC} portion arithmetic, which takes $\mathfrak{r}\left(\chi_{(-,3,-)}(\frac{59}{12})\right) = (1,0,1)$ as input and yields $\mathfrak{r}\left(\chi_{(-,4,-)}(\frac{167}{24})\right) = (1,1,1)$ as output after one hop. The lowermost illustration displays the T_{FC} portion arithmetic, which takes $\chi_{(-,-,2)}(\frac{59}{12}) = (0,1)$ as input and yields $\chi_{(-,-,3)}(\frac{167}{24}) = (1,0,1)$ as output following three hops.

3 Discussion

3.1 Operand Capacity Scaling

The Virtual Hensel I processor is of nest depth $5 + 2$ (i.e., Ξ_5^{virt} has five levels of 2AALUs, with the cluster PRs and master AALU/PR in turn occupying their own levels) and can handle over $50,000$ operands. For a processor of nest depth $\mathcal{L}$, the number of allowable operands can be roughly estimated as follows. Let ${}_{\mathcal{S}}\mathrm{P}_3$ be set of 3-tuples drawn from the set $\mathcal{S} = \{0, \ldots \mathcal{L} - 2\}$. Then, $\mathcal{S}$ gives the set of lists of block lengths for $\chi^{\mathfrak{d}}$-IDs up to length $\mathfrak{n} = \mathcal{L} - 2$. One then accounts for all possible $\chi^{\mathfrak{d}}$-IDs up to length $\mathfrak{n}$, and all possible compoundments of possible $\chi^{\mathfrak{d}}$-IDs into $\chi^{\mathfrak{c}}$-IDs. However, inasmuch as not all possible sequences of 0 or 1 terms actually appear in FC-3-2025 encodings (e.g., there are no $\chi^{\mathfrak{d}}_{(-,*,-)}$-IDs that terminate with superfluous 0 terms on the left, or $\chi^{\mathfrak{d}}_{(-,-,*)}$-IDs that end with superfluous terms on the right), we, as a rough procedure, divide the total by $\mathfrak{n}$. The formula is as follows:

$$\mathfrak{n}^{-1} \sum_{\mathfrak{i}=0}^{|\mathcal{S}|-1} \sum_{\mathfrak{j}=0}^{|\mathcal{S}|-1} \sum_{\mathfrak{k}=0}^{|\mathcal{S}|-1} 2^{\mathfrak{i}} 2^{\mathfrak{j}} 2^{\mathfrak{k}} \tag{10}$$

Figure 6 plots the estimated scaling trajectory for Hensel processor operand capacity by nest depth.

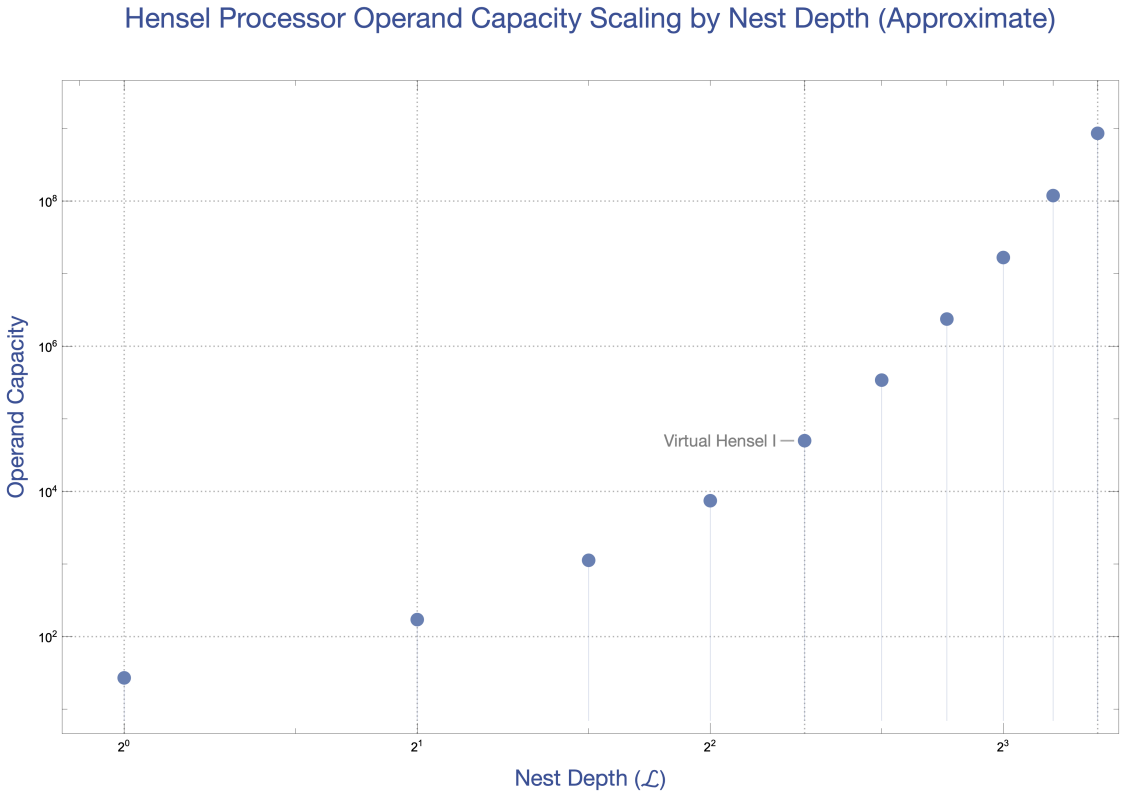


Figure 6: Estimated scaling behavior for CPU operand capacity by nest depth $\mathcal{L}$.

3.2 Arithmetic Reach

Beyond operand capacity, one might inquire into the number of arithmetic operations that can be permissibly executed on such operands. Not all operands loadable to the processor can be added or multiplied; namely, pairs whose sum or product exceeds the $\chi^{\mathfrak{c}}_{(5,5,5)}$ FC-3-2025 encoding ceiling cannot be added or multiplied. Thus, one would like to inquire into the set of Ξ_5^{virt}-loadable operands that are 2-ary sums or products of Ξ_5^{virt}-loadable operands. Let $\mathfrak{O}_{\Xi_5^{\text{virt}}}$ be the set of all Ξ_5^{virt}-loadable operands. We wish to inquire into the following two sets:

$$\{z \in \mathfrak{O}_{\Xi_5^{\text{virt}}} \mid z = x + y \wedge x, y \in \mathfrak{O}_{\Xi_5^{\text{virt}}}\} \tag{11}$$

$$\{w \in \mathfrak{O}_{\Xi_5^{\text{virt}}} \mid w = x \times y \wedge x, y \in \mathfrak{O}_{\Xi_5^{\text{virt}}}\} \tag{12}$$

Due to the sheer combinatorics of possible pairs of loadable operands, we can but conduct empirical studies of samples. We will take 2000 random samples of $\mathfrak{O}_{\Xi_5^{\text{virt}}}$, each of size 15 and, of the 225 admissible operand pairs obtainable from each sample, check to see which give sums or products that also belong to $\mathfrak{O}_{\Xi_5^{\text{virt}}}$.

This sample-checking approach gives the following empirical result. $26.7 \pm 9.7\%$ of the 225 pairs sum to a $\mathfrak{O}_{\Xi_5^{\text{virt}}}$ value. $0.67 \pm 0.79\%$ pairs multiply to a $\mathfrak{O}_{\Xi_5^{\text{virt}}}$ value. Thus, one infers that the $\approx 52,000$ operands loadable to the Virtual Hensel I may be subject to $\approx 7.2 \times 10^8 \pm 2.6 \times 10^8$ distinct 2-ary addition operations, and $1.8 \times 10^7 \pm 2.1 \times 10^7$ distinct 2-ary multiplication operations. While thhe overdispersion of this last statistic is something of an eyesore, one can nonetheless glean from estimates given here how arithmetic reach scales with operand capacity.

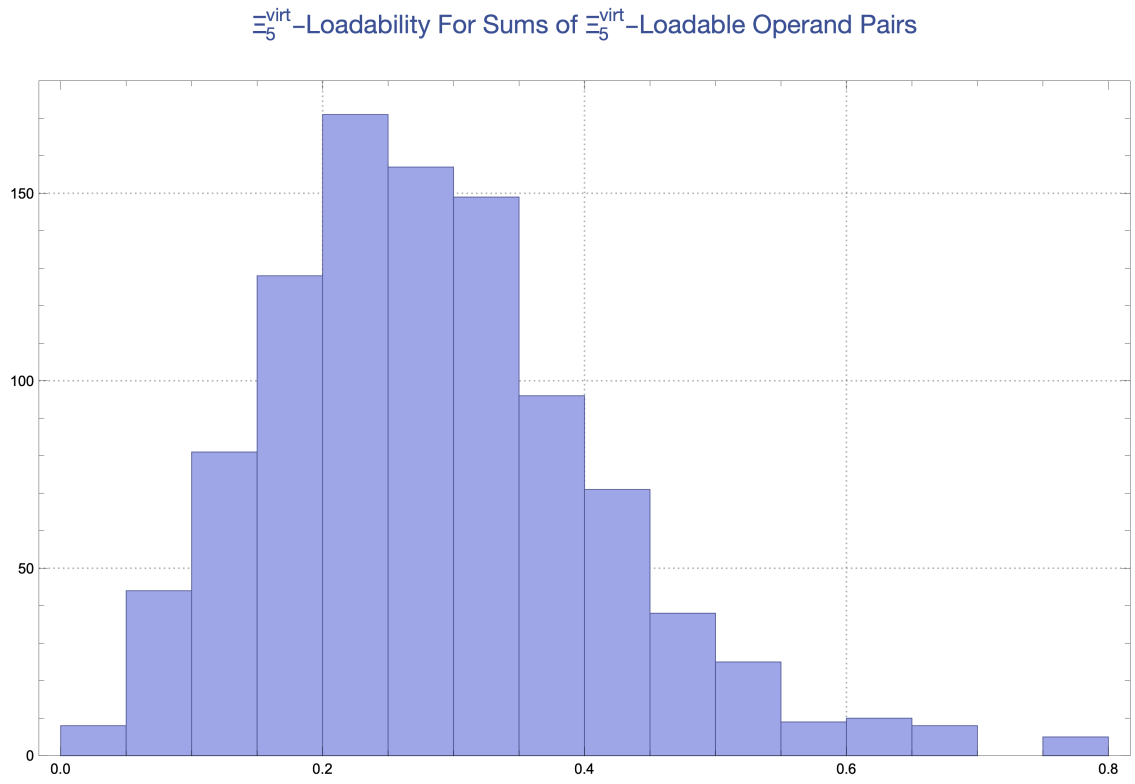


Figure 7: Distribution of the portion of sampled pairs $\{x, y\} \in \mathfrak{O}_{\Xi_5^{\text{virt}}} \oplus \mathfrak{O}_{\Xi_5^{\text{virt}}}$ that give sums $z \in \mathfrak{O}_{\Xi_5^{\text{virt}}}$

One might then ask about the distribution of sum and product values that are members of $\mathfrak{O}_{\Xi_5^{\text{virt}}}$. Preferring for this distribution to be as uniform as possible, one might wish to confirm that their distribution doesn't display discernible patches. Figure 8 plots $\{w \in \overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}} \mid (w = x \times y \vee w = x + y) \wedge x, y \in \overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}\}$ where $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$ is a random sample of 1000 entries from $\mathfrak{O}_{\Xi_5^{\text{virt}}}$. That is to say, it plots all pairs from $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$ whose sum or product also belongs to $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$. As one can see, the values extend outward roughly to 32 along each axis. Density is heterogeneous, but one doesn't find worrisome patches that cannot be attributed to sample size.

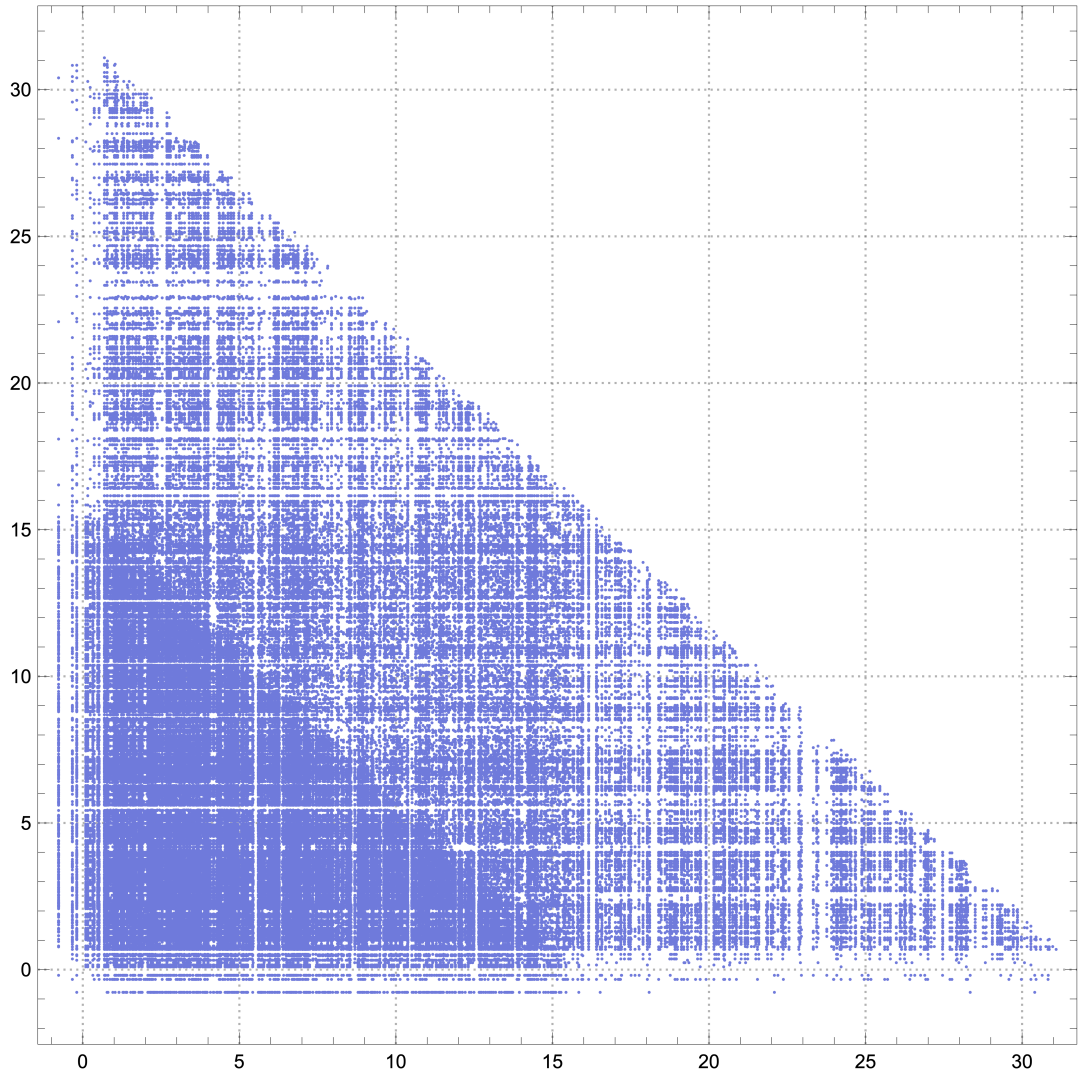


Figure 8: A plot of pairs drawn from $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$, where $|\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}| = 1000$, whose sum or product also belongs to $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$

3.3 The Cost-Savings of Exact Computing

SciSci Research and Future Computing are working to realize exact accelerated computing to address a critical juncture in the computing industry. Computing performance is now scaling to a point where floating-point errors dangerously accumulate. The IEEE floating-point standard gives approximations up to 64 bits, or 15 decimal places. Petascale computing achieves 10^{15} floating-point operations per second (FLOPS). Thus, petascale is literally the turning point where the cumulative error of floating point, per second, is no longer less than one. Petascale computing is not simply the purview of national laboratories or specialized supercomputing projects; the Nvidia Grace Blackwell is petascale. As shown in Figure 9, the cumulative costs per second of floating-point errors explode beyond petascale. Thus, the cost-savings of exact computing scale profoundly as operations-per-second performance climbs beyond petascale. It is for this reason that SciSci Research and Future Computing are endeavoring to begin a revolution in exact accelerated computing with the Hensel CPU architecture, beginning with Virtual Hensel I.

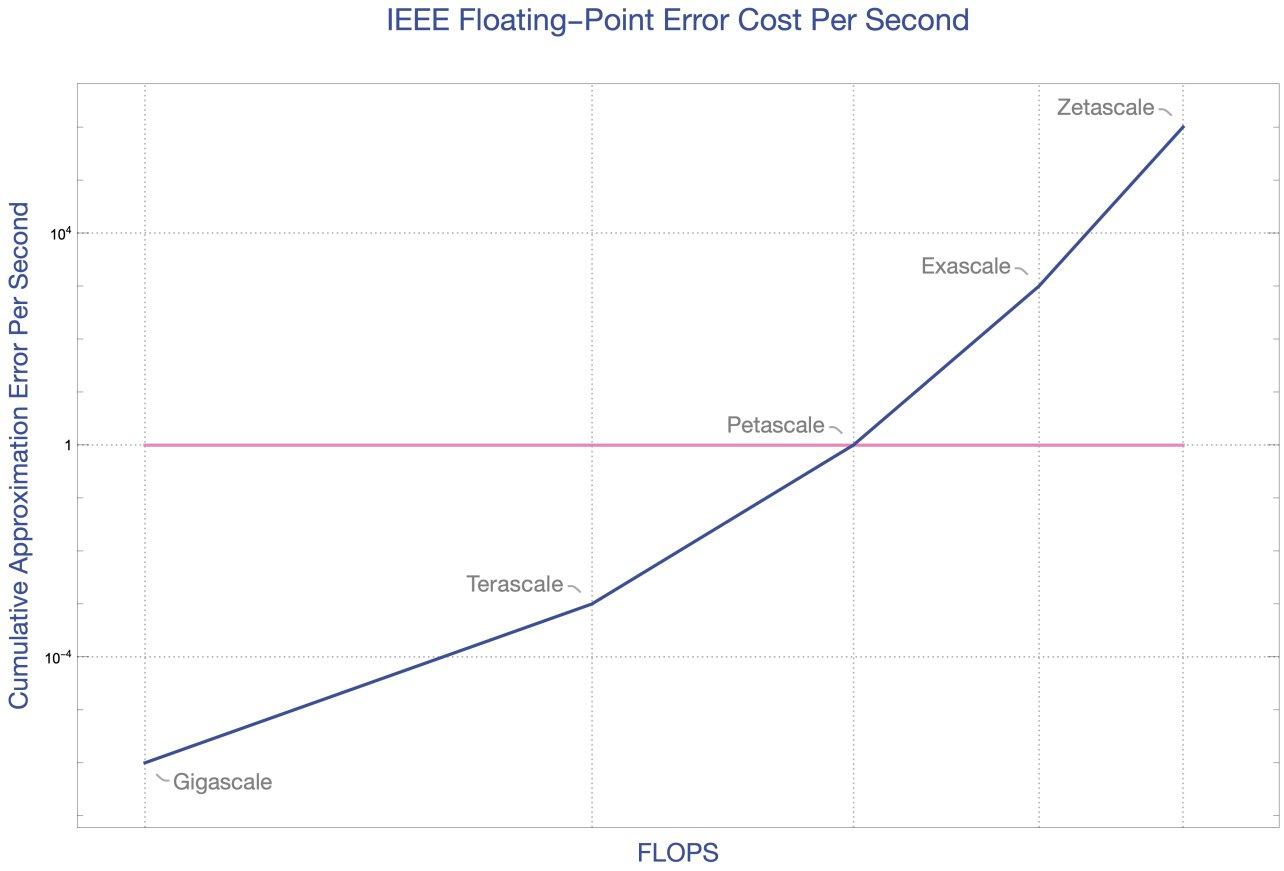
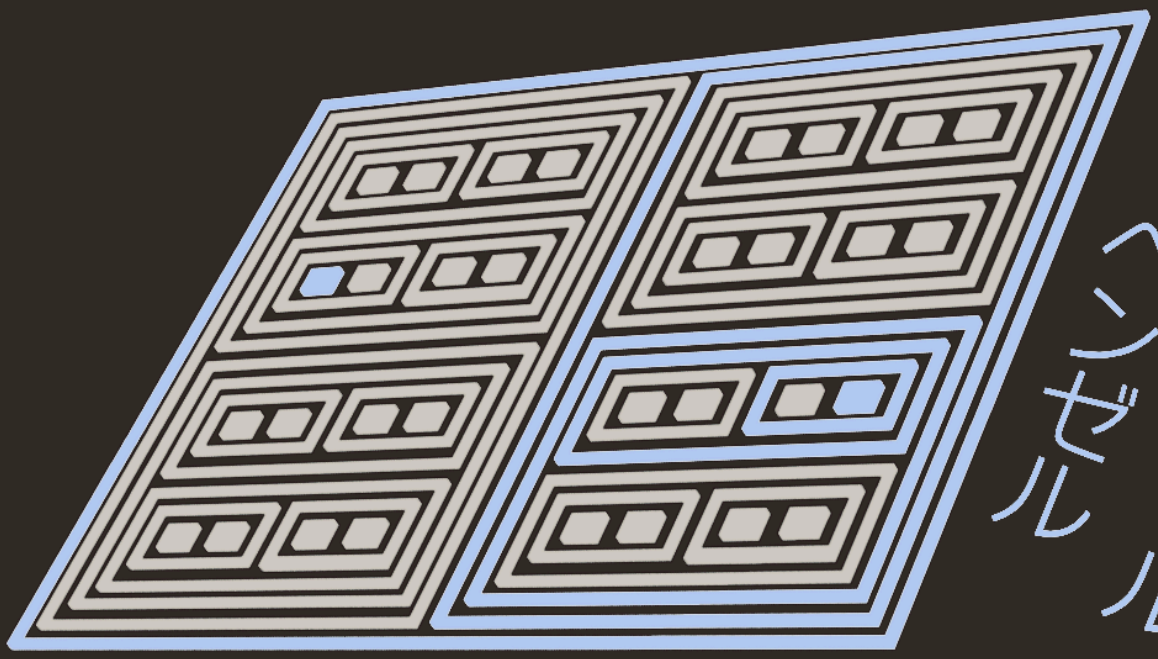


Figure 9: Cumulative approximation error per second for floating-point computing at varying FLOPS milestones, from gigascale computing to zetascale.



バーチャルヘンゼル

SciSci Research

ｷｬﾄｷｬﾄ・ﾆｷｰﾙ

Published by SciSci Press