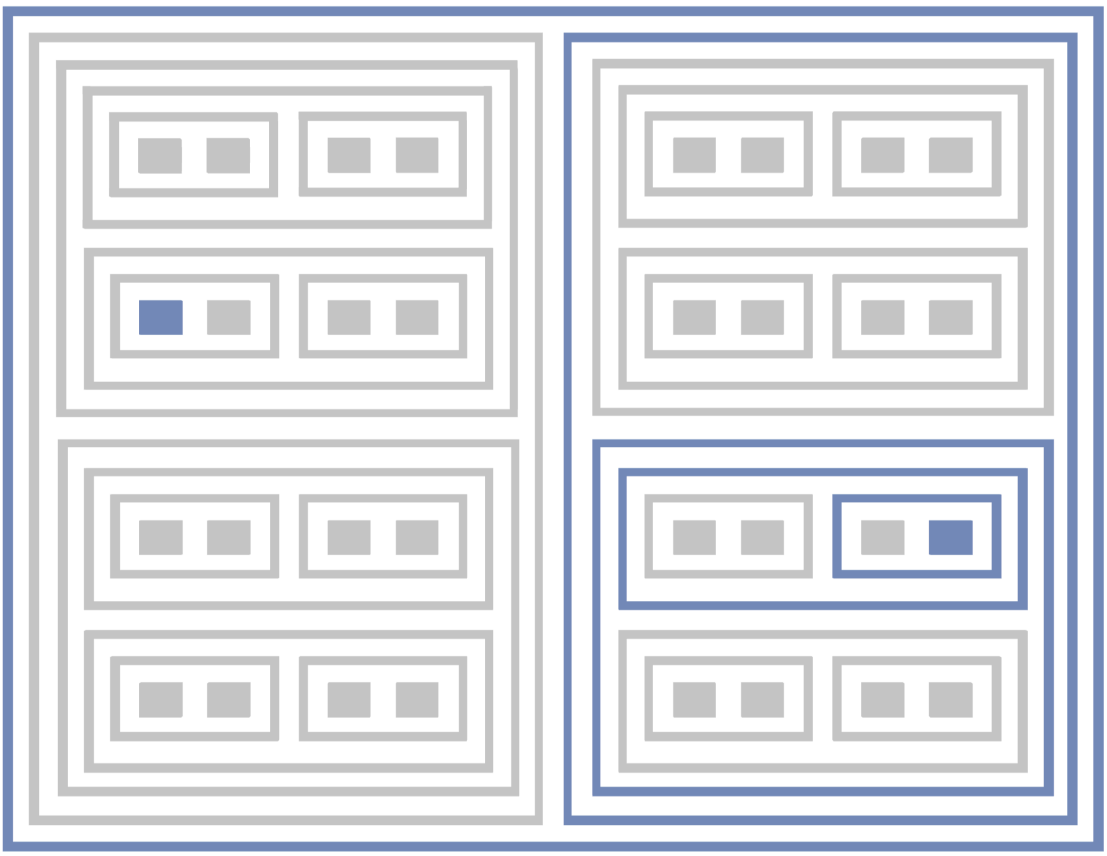
Inventions No. 2
sci-sci.org/virtual-hensel
DOI: 10.5281/zenodo.17602727

SciSci Research
サイサイ・リサーチ

Future Computing
コン 未来



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.17602727

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 load-store is performed in the processor and how 2-adic arithmetic is 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 load-store. (2AALU operations are covered in the 2AALU report). 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. The cluster, possessing a nested structure described in the original report, facilitates arithmetic and load-store 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 $\mathrm{FC}(\frac{4}{5})$ can be obtained by merging the $\chi^{\mathfrak{p}}$ for $\mathrm{FC}\left(-\frac{1}{5}\right)$ and $\mathrm{FC}(1)$: the encoding is $\mathrm{FC}(\frac{4}{5})=(\bot,(0,0,1,1),\bot,(1),\bot,())$. It is said that $\mathrm{FC}\left(\frac{4}{5}\right)$ 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 preliminarily defined as follows:

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

where $\chi^{\mathfrak{d}}_{(*,-,-)}=\mathrm{FC}(\eta_{\mathsf{A}})$, $\chi^{\mathfrak{d}}_{(-,*,-)}=\mathrm{FC}(\eta_{\mathsf{B}})$, $\chi^{\mathfrak{d}}_{(-,-,*)}=\mathrm{FC}(\eta_{\mathsf{C}})$, and $\eta_{\mathsf{A}}+\eta_{\mathsf{B}}+\eta_{\mathsf{C}}=\mathsf{q}$, the η being 2-adic expansions. (This definition, which is not quite complete, is given in full in section 2.3.1.) 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 $\mathrm{FC}(-1)$ and $\mathrm{FC}(1)$, the resulting $\chi^{\mathfrak{c}}$ merely encodes q, because, additively speaking, the -1 and 1 cancel each other out; one could have just used $\left((),(),\chi_{(-,-,\mathsf{n})}(\mathrm{FC}(\mathsf{q}))\right)$, rather than $\left(\chi_{(2,-,-)}(\mathrm{FC}(-1))\right)$, $\left(\chi_{(-,1,-)}(\mathrm{FC}(1))\right)$, and $\left(\chi_{(-,-,\mathsf{n})}(\mathrm{FC}(\mathsf{q}))\right)$ 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 $\chi^{\mathfrak{c}}_{(\min(\mathsf{n}),\min(\mathsf{m}),\min(\mathsf{l}))}$, where, when permissible, $\min(-)=0$.

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.

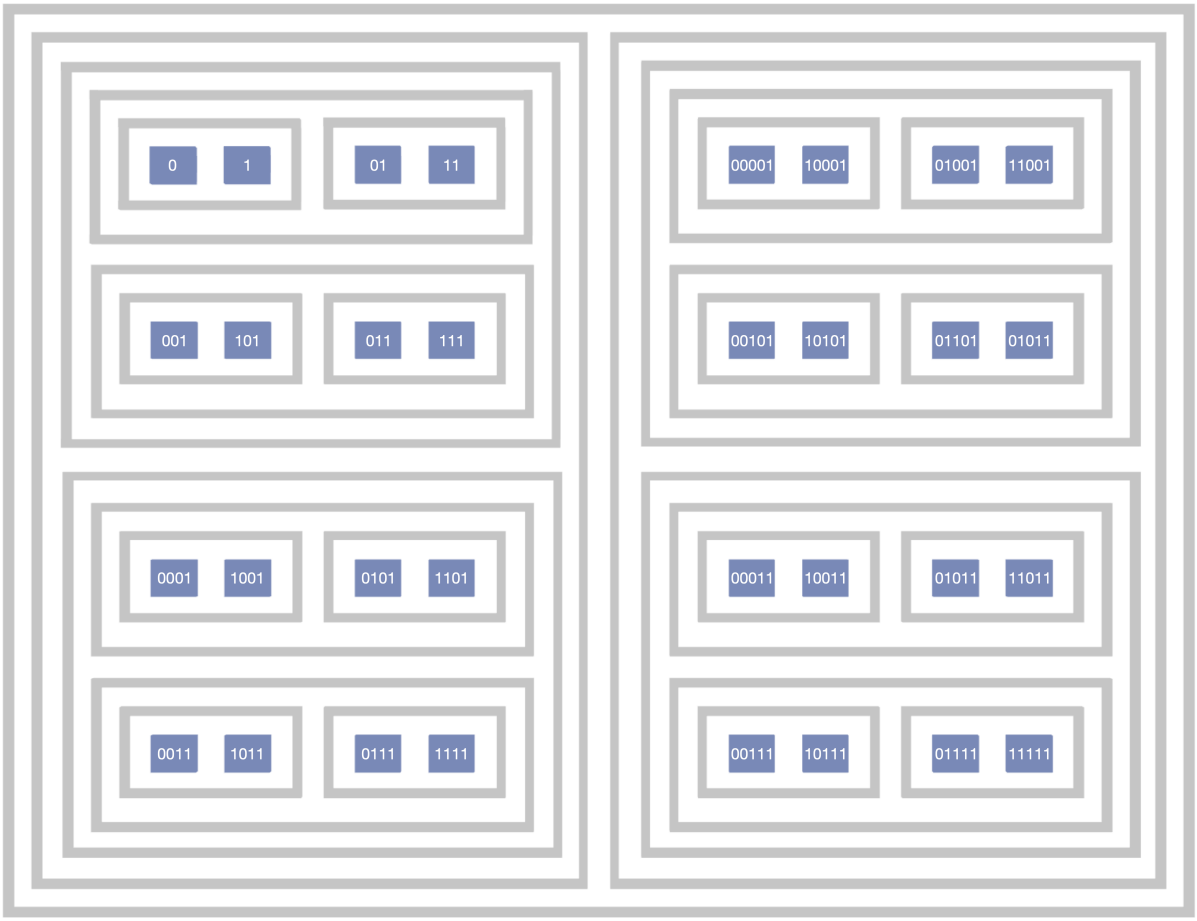


Figure 1: A labeled illustration of the 32 addresses $\mathfrak{A}(\mathcal{C}_{\mathrm{PR}})$ addresses for the cluster processor registers in the Virtual Hensel. The blue entities visualize register processors carriers, and the grey enclosures visualize the nested carrier packaging regime according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

One will quickly notice two coding-theoretic issues that arise when trying to facilitate load-store via $\chi^{\mathfrak{d}}$-$\mathfrak{A}$ matching.

First, $\mathsf{T_{FC}}$ block entries read from left to right, and the $\mathfrak{A}$ values for registers read from left to right, whereas $\mathsf{R_{FC}}$ and $\mathsf{L_{FC}}$ block entries read from right to left. This is rectified via the following trick. One applies a reverse operator $\mathfrak{r}$ to the $\mathsf{L_{FC}}$ $\chi^{\mathfrak{d}}$-ID and matches $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}\right)$ with the appropriate $\mathfrak{A}$. One also applies $\mathfrak{r}$ to $\chi^{\mathfrak{d}}_{(\mathsf{n},-,-)}$-IDs (i.e., for $\mathsf{R_{FC}}$).

Second, padding $\mathsf{R_{FC}}$ entries leaves the length of the repeated sequence unclear. For instance, whereas, in the case of $\mathsf{T_{FC}}$, 101 is no different from the right-padded 10100, in the case of $\mathsf{R_{FC}}$, 01 and 01000 are different repeating sequences. To rectify this issue, the FC-3-2025 standard also includes a new block, $\mathsf{N_{FC}}$, which gives the length of $\mathsf{R_{FC}}$. Because $\mathsf{N_{FC}}$ is integer-valued, it also has a $\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}$-ID, and is subject to the $\mathfrak{r}$ operator. By specifying the length of $\mathsf{R_{FC}}$ repeating sequences, one can encode all FC-3-2025 operands with a standard length, i.e., as the list

$$(\mathsf{N_{FC}}, \mathsf{R_{FC}}, \mathsf{L_{FC}}, \mathsf{T_{FC}}) \tag{2}$$

with each block encoded by the same number of bits, depending on the CPU. (For instance, in the case of the Virtual Hensel I, each block is of length 5. This will be important for the instruction set architecture.) We'll refer to any encoding involving solely the $\mathsf{R_{FC}}, \mathsf{L_{FC}}, \mathsf{T_{FC}}$ blocks, without the coding tricks, as the pre-FC-3-2025, or PFC-3-2025, encoding.

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:

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

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

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

and

$$\lambda^{\mathsf{D}}(\mathsf{FC}(\mathsf{q})): \left(\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}(\mathsf{FC}(\mathsf{q}))\right), (*,-,-,-)\right) \rightarrow \mathcal{C}^{\mathsf{D}}_{\mathsf{PR}}\left(\mathfrak{A}^{\chi^{\mathfrak{d}}_{(-,\mathsf{n},-)}}_{(*,-,-,-)}\right) \tag{6}$$

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, $\chi^{\mathfrak{d}}_{(-,2,-)} = (1,0)$ is acted on by the reverse operator in order to yield $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,2,-)}\right) = (0,1)$. (Note that $\chi^{\mathfrak{d}}$, regardless of being reversed or not, are right-padded to length five for $\mathfrak{A}$-matching, such that $(0,1)$ is treated as $(0,1,0,0,0)$.) As for $\left(\chi^{\mathfrak{d}}_{(2,-,-)}\right) = (0,1)$, it is reversed to $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,2,-)}\right) = (1,0)$. Thus, the loads are $\lambda^{\mathsf{A}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) = \mathcal{C}^{\mathsf{A}}_{\mathsf{PR}}\left(\mathfrak{A}^{(0,1)}_{(-,*,-,-)}\right)$, $\lambda^{\mathsf{B}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) = \mathcal{C}^{\mathsf{B}}_{\mathsf{PR}}\left(\mathfrak{A}^{(0,1)}_{(-,-,*,-)}\right)$, as well as $\lambda^{\mathsf{C}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) = \mathcal{C}^{\mathsf{C}}_{\mathsf{PR}}\left(\mathfrak{A}^{(1)}_{(-,-,-,*)}\right)$. Furthermore, the final and full FC-3-2025 form will include a $\mathsf{N_{FC}}$ term, $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,2,-)}\right) = (0,1)$, loaded as $\lambda^{\mathsf{D}}\left(\mathsf{FC}\left(\frac{13}{6}\right)\right) = \mathcal{C}^{\mathsf{C}}_{\mathsf{PR}}\left(\mathfrak{A}^{(0,1)}_{(*,-,-,-)}\right)$

Consider, on the other hand, the operand

$\frac{511}{62}$. Here,

$$\chi^{\mathfrak{c}}\left(\mathrm{FC}\left(\frac{511}{62}\right)\right) = (\bot, (0,1,0,0,0), \bot, (1,0,0,0), \bot, (1)) \quad (7)$$

The T_{FC} $\chi^{\mathfrak{d}}$-ID is $\chi^{\mathfrak{d}}_{(-,-,1)} = (1,0,0,0,0)$. Next, $\chi^{\mathfrak{d}}_{(-,4,-)} = (0,1,0,0,0)$ is reversed to $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,4,-)}\right) = (0,0,0,1,0)$. $\chi^{\mathfrak{d}}_{(5,-,-)} = (0,1,0,0,0)$ is reversed to $(0,0,0,1,0)$. Finally, the full FC-3-2025 encoding also includes the N_{FC} term, $(0,1,0,0,0)$, reversed to $(0,0,0,1,0)$.

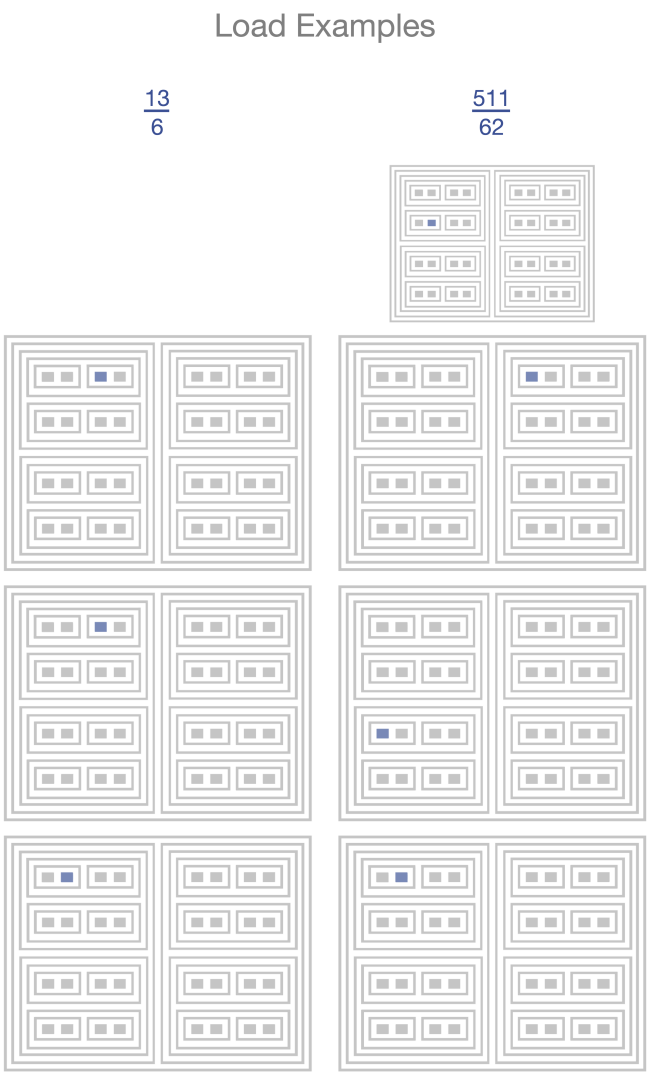


Figure 2: Illustration of the $\mathfrak{A}$ addresses for loads $\lambda^{\mathrm{A}}(\mathrm{FC}(\frac{13}{6}))$, $\lambda^{\mathrm{B}}(\mathrm{FC}(\frac{13}{6}))$, and $\lambda^{\mathrm{C}}(\mathrm{FC}(\frac{13}{6}))$, as well as $\lambda^{\mathrm{A}}(\mathrm{FC}(\frac{511}{62}))$, $\lambda^{\mathrm{B}}(\mathrm{FC}(\frac{511}{62}))$, $\lambda^{\mathrm{C}}(\mathrm{FC}(\frac{511}{62}))$, and $\lambda^{\mathrm{D}}(\mathrm{FC}(\frac{511}{62}))$. 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} blocks (i.e., $\chi^{\mathfrak{d}}$-IDs), subject to come coding tricks, – and also includes an N_{FC} block – which are loaded in distributed fashion to various processor registers in the processor cluster with matching addresses. A given operand can be decompounded 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 N_{FC}-block-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 $\mathcal{M}_{LSU}$, which composes them into an FC-3-2025-encoded operand for LSU-facilitated storage. The master LSU performs the following compoundment:

$$\kappa : \left(\mathfrak{r}\left(\chi_{\mathcal{C}_{PR}^{D'}}\right), \mathfrak{r}\left(\chi_{\mathcal{C}_{PR}^{A'}}\right), \mathfrak{r}\left(\chi_{\mathcal{C}_{PR}^{B'}}\right), \chi_{\mathcal{C}_{PR}^{C'}}\right) \rightarrow \left(\mathfrak{r}^{-1}\left(\chi_{\mathcal{C}_{PR}^{D'}}\right), \mathfrak{r}^{-1}\left(\chi_{\mathcal{C}_{PR}^{A'}}\right), \mathfrak{r}^{-1}\left(\chi_{\mathcal{C}_{PR}^{B'}}\right), \chi_{\mathcal{C}_{PR}^{C'}}\right) \quad (8)$$

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

2.4 Re-Loading

LSU loaders in the Hensel processor perform 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, the original operand can be re-loaded to new registers whose addresses match these $\chi^{\mathfrak{d}}$-IDs. The Hensel LSU performs re-loads via FC-2-2025 instructions, which modify $\mathfrak{A}$-values in accordance with new $\chi^{\mathfrak{d}}$-IDs, and then passing the π-sequences to the registers with these $\mathfrak{A}$-values.

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

With there being five processor register cluster levels in Ξ_5^{virt} and 2^5 processor registers, and the $\chi^{\mathfrak{d}}$-IDs being of maximum block length 5, a loader at each level performs a modification on a different entry in the $\chi^{\mathfrak{d}}$-ID, in parallel, with the right-most 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 $\mathfrak{A}$-value at the entry position corresponding with the level, with each entry modification in turn changing the $\mathfrak{A}$-value, and thus the $\mathcal{C}_{PR}$ to which the $\chi^{\mathfrak{d}}$-ID is to be loaded. With processor registers packaged in nested carriers, 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 loader modifying a single entry in turn affects the location in which the address-matching processor register will be located; a change of one "hop", as will

be the turn of phrase used here. Moreover, because, in the nested structure, each processor register has only one other same-level-ℓ neighbor packaged within the same level-$(\ell+1)$ carrier, a modification of an address by a given level-ℓ loader yields an output whose $\mathcal{C}_{\mathsf{PR}}$ location is within the nested carrier packaging of the neighbor of the processor register whose address was modified. 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 cluster level differ in address by only one entry (the leftmost entry), and will neighbor one another on the circuit board. If they differ in entries further to the right, then their nested carrier packaging will differ at higher levels, 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 $\mathfrak{A}$-values of the 32 $\mathcal{C}_{\mathsf{PR}}$.)

Herein, we will describe loader 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 load-store report.

Loads and reloads are performed on FC-3-2025-encoded operands according to FC-2-2025 instructions, which guide the loaders by specifying the entries in an operand encoding to be modified, and the cluster level at which the modification is to take place. "Hop operations" are a shorthand for loader modifications. 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 cluster 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 $\mathcal{M}_{\mathsf{LSU}}$. The middle five are for loaders. A hop $\mathsf{h}^{\sigma}_{\ell=2}$ is then equivalent to a reload that modifies an operand by $(1,0,0,0,0)$; $\mathsf{h}^{\sigma}_{\ell=3}$, a modification by $(0,1,0,0,0)$; $\mathsf{h}^{\sigma}_{\ell=4}$, a modification by $(0,0,1,0,0)$; $\mathsf{h}^{\sigma}_{\ell=5}$, a modification by $(0,0,0,1,0)$; and $\mathsf{h}^{\sigma}_{\ell=6}$, a modification by $(0,0,0,0,1)$. Likewise, a hop $\mathsf{h}^{\overline{\sigma}}_{\ell=2}$ is equivalent to a reload that modifies an operand by $(-1,0,0,0,0)$ to a $\chi^{\mathfrak{d}}$; $\mathsf{h}^{\overline{\sigma}}_{\ell=3}$, a modification by $(0,-1,0,0,0)$; $\mathsf{h}^{\overline{\sigma}}_{\ell=4}$, a modification by $(0,0,-1,0,0)$; $\mathsf{h}^{\overline{\sigma}}_{\ell=5}$, a modification by $(0,0,0,-1,0)$; and $\mathsf{h}^{\overline{\sigma}}_{\ell=6}$, a modification by $(0,0,0,0,-1)$. If a $\chi^{\mathfrak{d}}$-ID, 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 $(1,1,0,0,0)$.

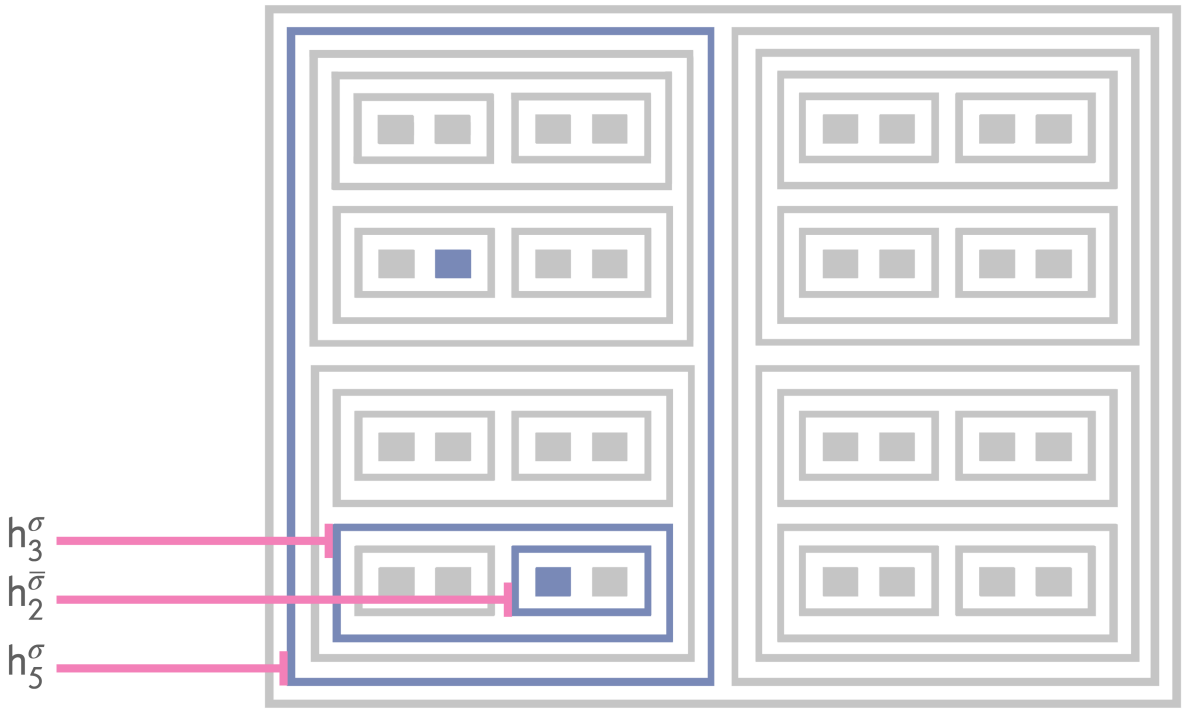


Figure 3: Illustration of hop visualization in the Virtual Hensel demo. A forward hop h_ℓ^σ gives an output located upward or rightward of the input, and a backward hop h_ℓ^σ gives an output upward or leftward of the input.

2.5 A Worked Example

Let's take the example of FC $\left(\frac{59}{12}\right)$ + FC $\left(\frac{49}{24}\right)$, which of course yields 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

$$\frac{59}{12} = 2^{-2} + 2 + 2^3 + 2^4 + 2^6 + 2^8 + 2^{10} + 2^{12} \ldots \quad (9)$$

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, $\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right) = (0,1)$ and $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right)\right) = (1,0)$. Next, because $\chi^{\mathfrak{p}}(5) = (1,0,1)$, it is the case that $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{59}{12}\right)\right) = (1,0,1)$. Finally, we see that because $\chi^{\mathfrak{p}}\left(-\frac{1}{3}\right) = (0,1)$, it is the case that $\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right) = (0,1)$ and $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right)\right) = (1,0)$.

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

$$\frac{49}{24} = 2^{-3} + 2^{-2} + 1 + 2 + 2^2 + 2^4 + 2^6 + 2^8 \ldots \quad (10)$$

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 $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{49}{24}\right)\right) = (1,0)$.

Lastly, $\frac{167}{24} = -\frac{2}{3} + 7 + \frac{5}{8}$, so one gets

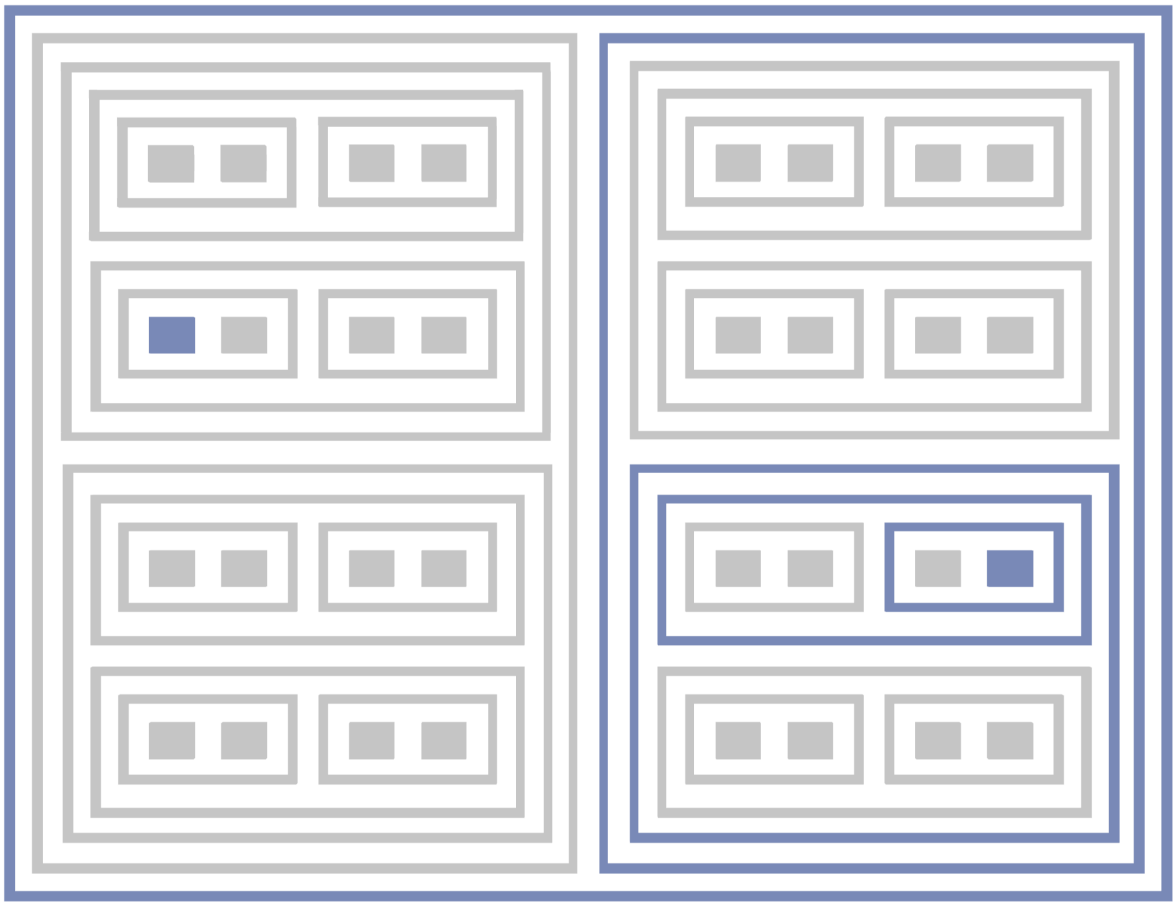


Figure 4: Arithmetic example: $3 + 7 = 10$. The input $\mathrm{FC}(3)$ is loaded to the $\mathcal{C}_{\mathrm{PR}}$ with address $\mathfrak{A}^{(1,0,1)}_{(-,3,-)}$ and the output $\mathrm{FC}(10)$ is loaded to the $\mathcal{C}_{\mathrm{PR}}$ with address $\mathfrak{A}^{(0,1,0,1)}_{(-,4,-)}$ following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

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, $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{167}{24}\right)\right) = (0,1)$.

Let's now look at the operand re-loads in terms of hop calculus. The arithmetic modification on the $\mathfrak{A}$-value for $\chi^{\mathfrak{d}}_{(-,-,2)}\left(\frac{59}{12}\right)$ (with input $(0,1)$ and output $(1,0,1)$) involves three hops: $\mathrm{h}^{\sigma}_{\ell=4}, \mathrm{h}^{\overline{\sigma}}_{\ell=3}, \mathrm{h}^{\sigma}_{\ell=2}$, which adds $(0,0,1,0,0)$, subtracts $(0,1,0,0,0)$, and adds $(1,0,0,0,0)$, respectively. The modification on the $\mathfrak{A}$-value for $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(-,3,-)}\left(\frac{59}{12}\right)\right)$ (with input $(1,0,1)$ and output $(1,1,1)$) involves one hop: $\mathrm{h}^{\sigma}_{\ell=3}$. Finally, the modification of the $\mathfrak{A}$-value for $\mathfrak{r}\left(\chi^{\mathfrak{d}}_{(2,-,-)}\left(\frac{59}{12}\right)\right)$ involves two hops, $\mathrm{h}^{\overline{\sigma}}_{\ell=3}$ and $\mathrm{h}^{\sigma}_{\ell=2}$.

Figure 5: Visual illustration of addition: $\mathsf{FC}\left(\frac{59}{12}\right) + \mathsf{FC}\left(\frac{49}{24}\right)$. The uppermost illustration displays the $\mathsf{N_{FC}}$ block, which remains unchanged. The second uppermost illustration shows $\mathsf{R_{FC}}$ block arithmetic, which takes $\mathfrak{r}\left(\chi_{(2,-,-)}(\frac{59}{12})\right) = (0,1)$ as input and yields $\mathfrak{r}\left(\chi_{(2,-,-)}(\frac{167}{24})\right) = (1,0)$ as output, whose register is separated by two hops. The middle illustration displays the $\mathsf{L_{FC}}$ block 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, whose register is separated by one hop. The lowermost illustration displays the $\mathsf{T_{FC}}$ 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, whose register is separated by three hops.

3 Virtual Hensel Demo

SciSci recently released a demo video for the Virtual Hensel I. The video includes animations simulating the combinational logic of 2AALUs and operand loading to the register cluster, as shown in Figure 6. Examples of exact arithmetic are shown, with inputs given in ARIXA assembly and converted to MRIXA machine code, with MRIXA outputs giving exact numerical results. A comparison with 20-bit floating point is also given. (See Figure 7 for an example.)

The Virtual Hensel video can be viewed as presenting a brief visual and computational synthesis of recent SciSci reports. The computations shown are executed using the same combinational logic that physical 2AALUs would use for length-5 $\chi^{\mathfrak{d}}$-IDs, i.e., the same Boolean functions as its gates. The $\chi^{\mathfrak{d}}$-IDs are subject to $\chi^{\mathfrak{d}}$-$\mathfrak{A}$ matching, and loaded to virtual registers the same way that they would be loaded to physical registers in the processor cluster. The MRIXA output, written in bits, is the same as the digital output that would be obtained from the physical CPU circuit. In summary, the Virtual Hensel computations shown in the demo are real computations, and the same as those which a 20-bit Hensel would perform (only virtual).

The key difference between the Virtual Hensel and the actual Hensel CPU is that the latter will have a larger bit-width; it will have more registers and, producing operands longer than 20 bits in length, have more circuitry in its 2AALUs. Nonetheless, if the actual CPU only had 32 registers and gave 20-bit outputs, it would be the same. The other point to note is that, whereas the demo, itself rather brief, only gives examples of addition and subtraction, ARIXA/MRIXA already supports multiplication/division, as discussed in the RIXA report.

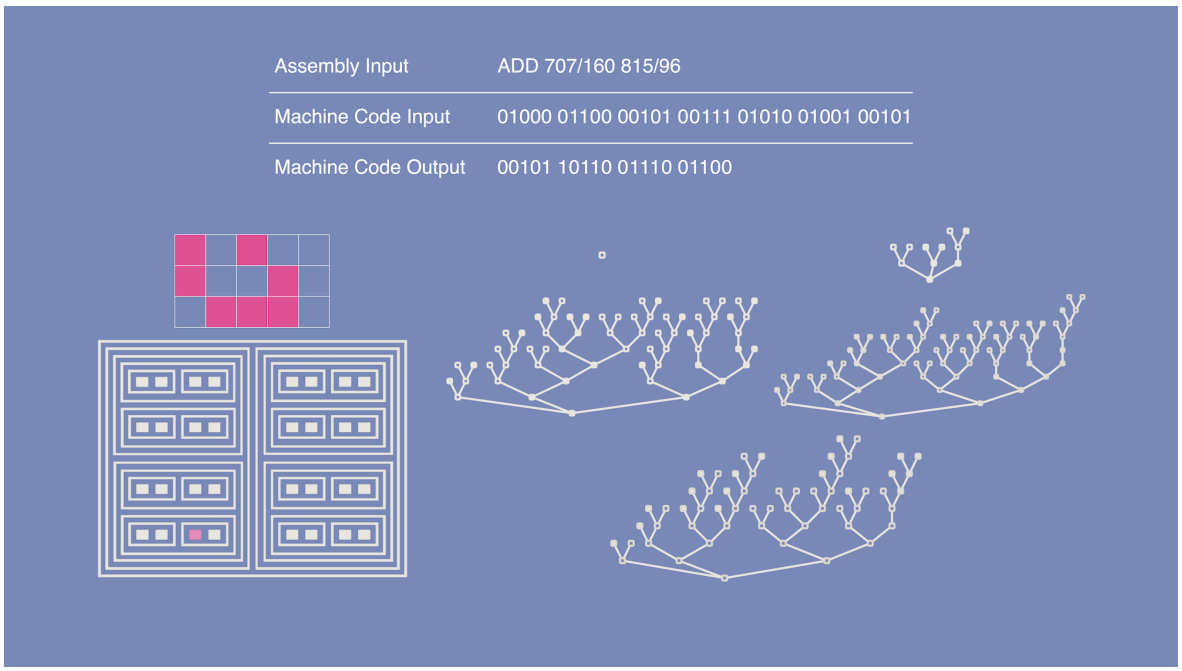


Figure 6: Virtual Hensel demo video image: visualization of 2AALU combinational logic and register cluster operand loading, with computations instructed by ARIXA input and returning MRIXA output.

```
VirtualHensel["ADD 707/160 815/96"]
```

Assembly Input	ADD 707/160 815/96
Machine Code Input	01000 01100 00101 00111 01010 01001 00101
Machine Code Output	00101 10110 01110 01100

```
ShowCompiler[{"00101 10110 01110 01100"}]
```

Machine Code Output	{00101 10110 01110 01100}
Numerical Output	$\frac{1549}{120}$

```
ShowAdvantage["ADD 707/160 815/96", {"00101 10110 01110 01100"}]
```

Error Prevented	0.00833333

Figure 7: Virtual Hensel demo video image: An ARIXA assembly input, ADD 707/160 815/96, is given, converted to MRIXA input, and computed by the Virtual Hensel to return 00101 10110 01110 01100 as MRIXA output. This is then converted to the exact result, $\frac{1549}{120}$, by the compiler. A comparison with 20-bit floating point is also given: the Virtual Hensel prevents a rounding error of $0.008\overline{3}$.

4 Discussion

4.1 Operand Capacity Scaling

The Virtual Hensel I processor is of nest depth $5+2$ (i.e., Ξ_5^{virt} has five levels of loaders, with the cluster PRs and master LSU/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 $\mathrm{n} = \mathcal{L}-2$. One then accounts for all possible $\chi^{\mathfrak{d}}$-IDs up to length 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 n. The formula is as follows:

$$\mathrm{n}^{-1} \sum_{\mathrm{i}=0}^{|\mathcal{S}|-1} \sum_{\mathrm{j}=0}^{|\mathcal{S}|-1} \sum_{\mathrm{k}=0}^{|\mathcal{S}|-1} 2^{\mathrm{i}} 2^{\mathrm{j}} 2^{\mathrm{k}} \tag{11}$$

Figure 8 plots the estimated scaling trajectory for Hensel processor operand capacity by nest depth.

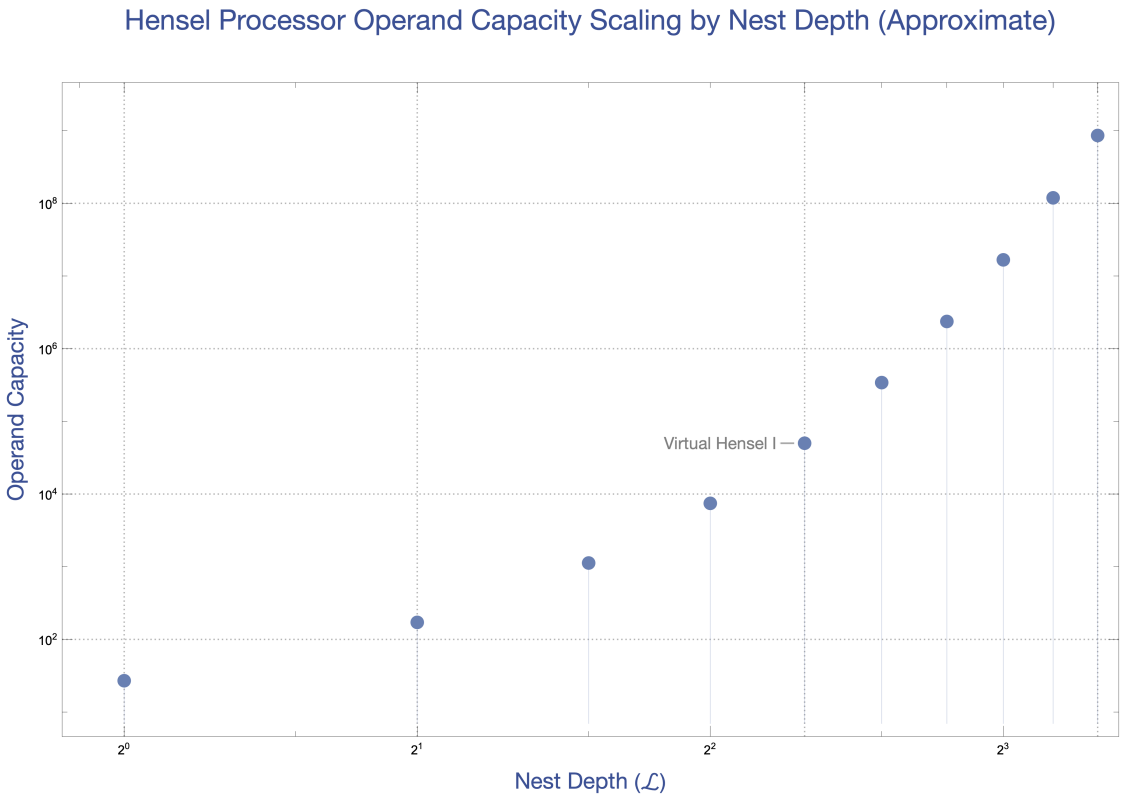


Figure 8: Estimated scaling behavior for CPU operand capacity by nest depth $\mathcal{L}$.

4.2 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:

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

$$\{w \in \mathfrak{O}_{\Xi_5^{\text{virt}}} \mid w = x \times y \wedge x, y \in \mathfrak{O}_{\Xi_5^{\text{virt}}}\} \tag{13}$$

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.

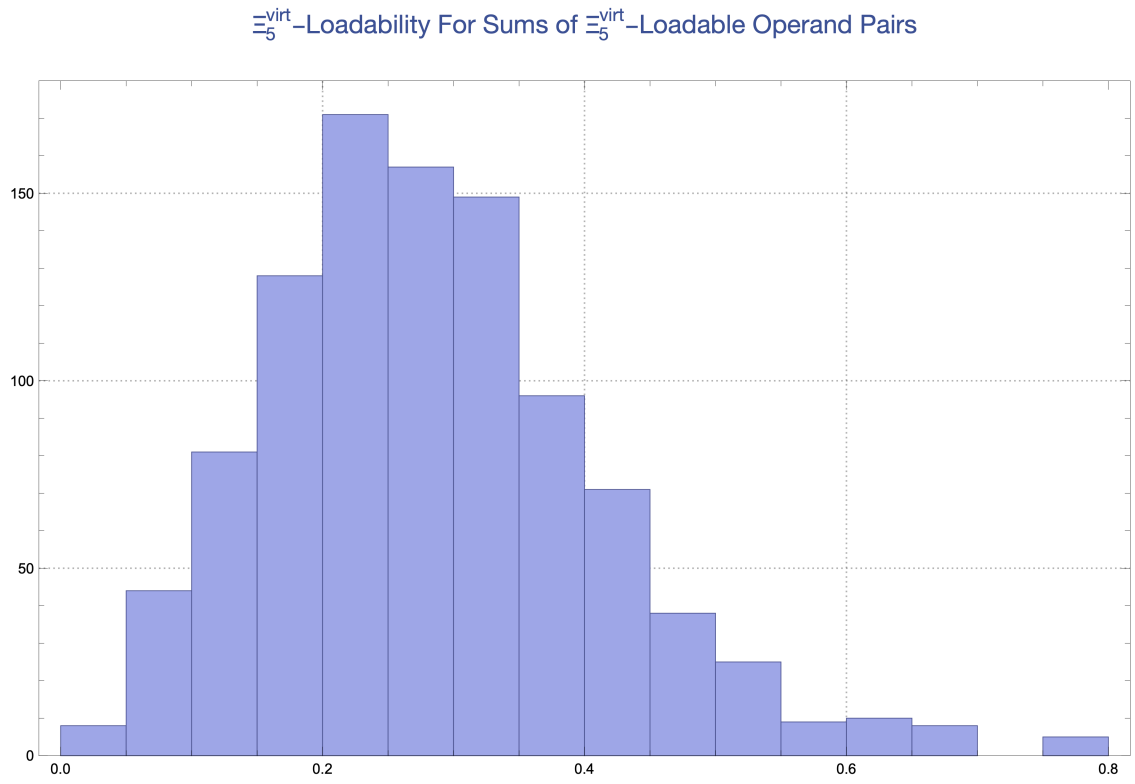


Figure 9: Distribution of the portion of sampled pairs $\{x, y\} \in \mathfrak{O}_{\Xi_5^{\text{virt}}} \oplus \mathfrak{O}_{\Xi_5^{\text{virt}}}$ that give sums $z \in \mathfrak{O}_{\Xi_5^{\text{virt}}}$

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 10 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.

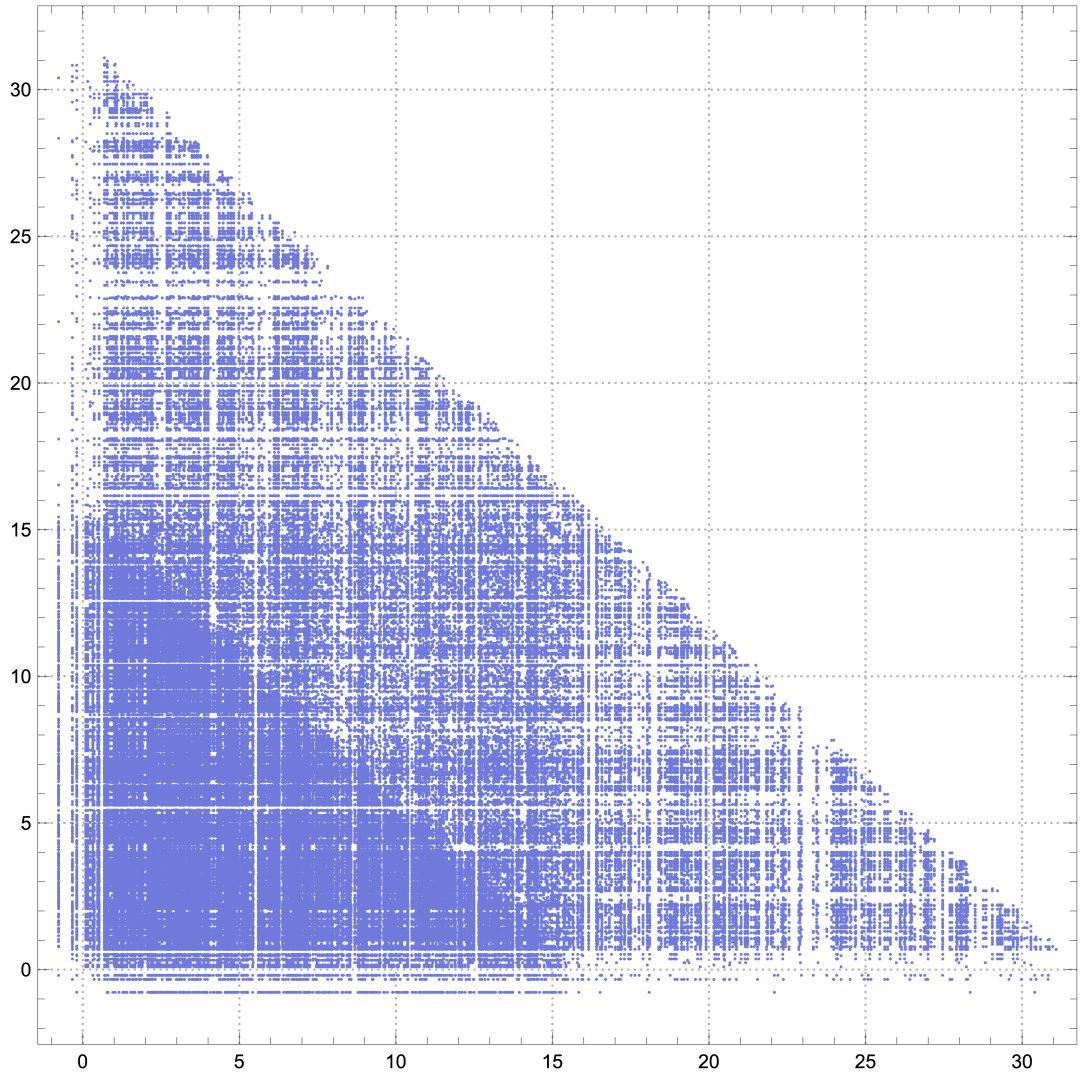


Figure 10: A plot of pairs drawn from $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$, where $|\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}| = 1000$, whose sum or product also belongs to $\overline{\mathfrak{O}}_{\Xi_5^{\text{virt}}}$

4.3 The Cost-Savings of Exact Computing

SciSci Research and Future Computing are working to realize exact computing to address a critical juncture in the high-performance computing industry. Computing performance is now scaling to a point where floating-point errors dangerously accumulate. The IEEE "double-precision" floating-point standard gives approximations up to 64 bits, or 16 decimal places. Petascale computing is between 10^{15}-10^{17} 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. As shown in Figure 11, 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 computing with the Hensel CPU architecture, beginning with Virtual Hensel I.

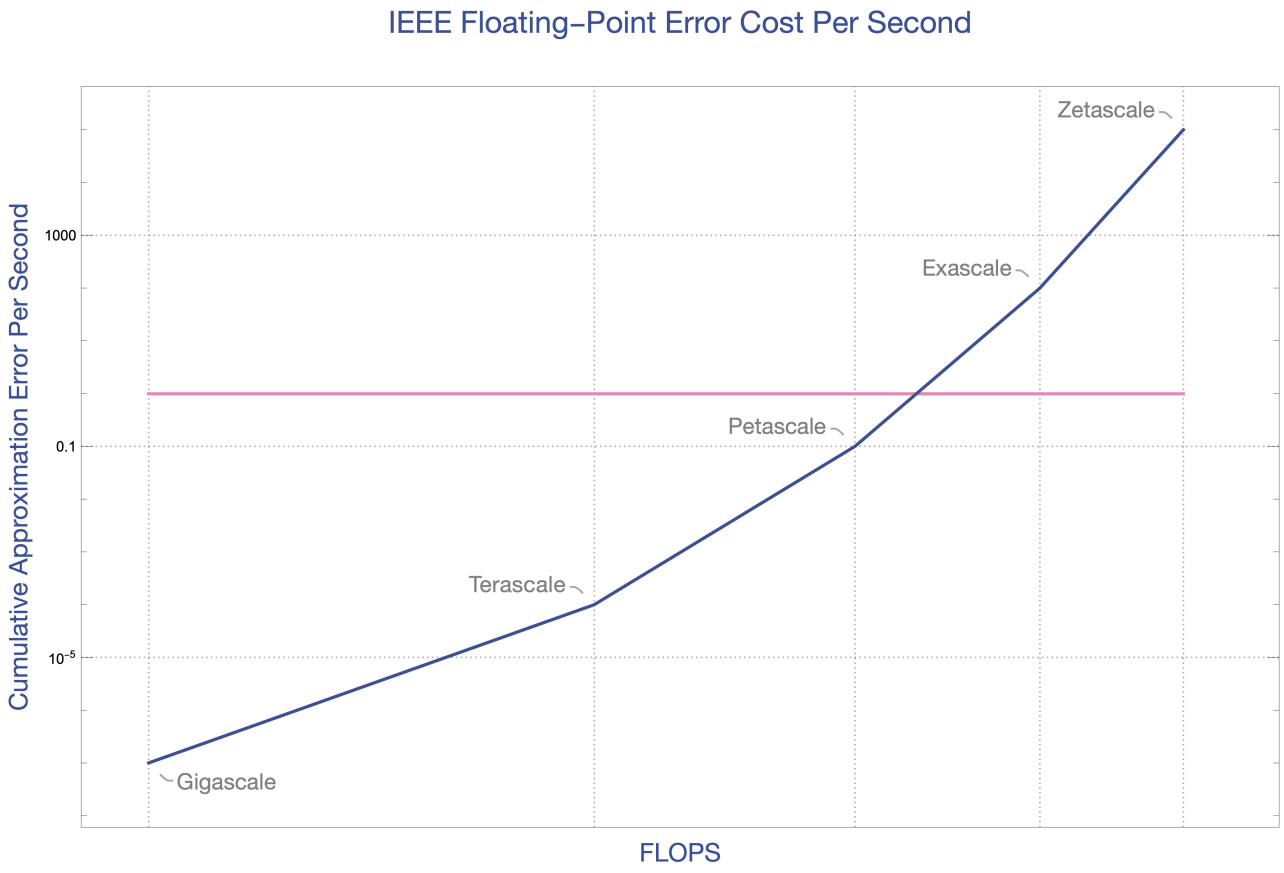


Figure 11: Cumulative approximation error per second for floating-point computing at varying FLOPS milestones, from gigascale computing to zetascale.

5 Looking Ahead

5.1 Roots (i.e., Beyond $\mathbb{Q}$)

A discerning reader may have noted that all examples of operands given in this report are FC-3-2025 encodings of 2-adic expansions of rationals (i.e., elements of $\mathbb{Q} \hookrightarrow \mathbb{Q}_2$). One might, from the perspective of algebraic number theory, express concerns regarding potential limitations of a $\mathbb{Q}_2$-based architecture for handling operands beyond $\mathbb{Q}$, e.g., operands that one might find in algebraic extensions of $\mathbb{Q}$. For instance, one might express concerns due to the fact that the existence of square roots in p-adic arithmetic depends on issues such as the existence of squares $\bmod p$, with $\mathbb{Q}_2$ being exceptionally thorny among $\mathbb{Q}_p$ in this respect. However, with computer engineering priorities in mind, such complications can be safely avoided with the help of some design tricks.

The FC-3-2025 standard allows for general, finite, unique, 2-adic encodings of operands in $\mathbb{Q} \hookrightarrow \mathbb{Q}_2$. From there, one can manipulate encodings to handle roots. For instance, both 2 and $\frac{1}{2}$ have simple FC-3-2025 encodings. $\sqrt{2}$ is just $2^{\frac{1}{2}}$; thus, one need only encode an exponentiation operation along with these operands in order to encode $\sqrt{2}$. One should be able to compute with operands in $\mathbb{Q}[\sqrt{2}]$ this way. Likewise, the FC-3-2025 encoding for -1 is simple; one can just treat i as $-1^{\frac{1}{2}}$. One should be able to encode operands in $\mathbb{Q}[i]$ this way. Because all operands are decompounded into $\chi^{\mathfrak{d}}$-IDs, one could simply augment FC-3-2025 (i.e., to "FC-4-2025") to include an encoding block denoting exponentiation between T_{FC}, L_{FC}, and R_{FC} blocks. With an encoding standard as such, one could exponentiate any FC-3-2025-encoded rational by any other FC-3-2025-encoded rational. The exponent blocks could also be subject to arithmetic according to the rules of exponentiation. (For instance, the CPU could compute $3^{\frac{1}{2}} \times 3^2 = 3^{\frac{5}{2}}$ by simply adding the exponent blocks.)

So long as exponentiation is given code-theoretically (i.e., encoded into the operand as a block), rather than computed arithmetically, the Hensel CPU's computations are still technically performed in $\mathbb{Q}_2$, since the individual $\chi^{\mathfrak{d}}$-IDs (including for the exponent blocks) are all given 2-adic encodings. Utilizing this trick, Future Computing can avail itself of the benefits of computing in $\mathbb{Q}_2$ all the while eschewing the straits in which one might find oneself when contending with 2-adic algebraic number theory.

5.2 The Compiler and ISA

The question of whether exponentiation should be treated as an operation or part of the operand offers foresight as Future Computing thinks ahead to matters such as machine code, assembly language, and the compiler. Indeed, if it is treated as part of the operand (as suggested above), then it will be encoded and later abstracted rather than be computed. However, this is in keeping with Future Computing's plan for Hensel CPU operands to be treated "symbolically" when subject to abstraction in higher-level programming. For instance, because FC $\left(\frac{1}{3}\right)$ is handled exactly, there is no need to ever write out decimals for $\frac{1}{3}$; one can simply work with the abstraction "$\frac{1}{3}$". Thus, likewise, a computation might yield an output abstracted as $\left(\frac{5}{26}\right)^{\left(\frac{17}{4}\right)}$, with the exponent unresolved. The user, if seeking to obtain a numerical answer (up to their desired decision), can then elect to do so (as one can do in software such as Mathematica). Nonetheless, numeric resolution will not be performed by the CPU as part of arithmetic operations.

(Update, the above strategy was indeed used for the latest Virtual Hensel demo.)

SciSci Research

ｷｬﾄｷｬﾄ・ﾆｷｰﾙ

Published by SciSci Press