Elegant ALUs from Surface Mount SRAMs

A proposal submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

by

Marc W. Abel
B.S., California Institute of Technology, 1991

2020
Wright State University

ABSTRACT

Marc W. Abel, Department of Computer Science and Engineering,
Wright State University, 2020. Elegant ALUs from Surface Mount SRAMs.

We consider the logic family comprised of separately packaged asynchronous static RAMs for building computing systems requiring isolation from supply-chain backdoors. Static RAM is singular in its ability to operate as pure combinational logic notwithstanding its high transistor count, resulting in tremendous computational elegance on one hand, but no sequential enclaves to build a backdoor into on the other hand. Still further security benefit derives from SRAM's fungibility, low cost, assured absence of data when power is removed, and limited input word size.

Register files, control decoders, and memory management units are straightforward to produce using SRAM. But a knowledge gap exists for arithmetic logic units, or ALUs. It can be tempting to encode "textbook" ALUs into SRAM tables, but so doing throws away much computing power and speed. It turns out that even small ALUs can implement not just bitwise and additive logic, but also fast rotations and shifts with overflow detection, subword transpositions and copies, bit and prefix counts, floating point micro-operations, short multiplication, pointer realignments, minimum and maximum, mixed-signedness comparisons, mixed-signedness arithmetic with overflow detection, subword permutations, and robust mixing functions for hashing, pseudorandom numbers, and cryptography.

An interesting duality emerges between the computation required of ALUs and the structure of SRAM logic. This duality tends to pull SRAM ALU designs into dense, elegant circuits with many symmetries, surprising word size preferences, and unexpected speed. The proposed research will specify, test, characterize, and disseminate a 36-bit ALU design that arises from these structures, thereby combining the robustness of computation offered by SRAM ALUs with the ease of manufacture and peace of mind they afford.

CONTENTS

Plates

Listings

Tables

1. Problem and approach

1.1 Manifesto on the security of CPUs

Herein is the privilege of being a computer scientist: each of us has an opportunity to identify technology that could bolster a specific and preferably humanitarian agenda, and then seek out that technology and make it visible for others. A special interest I owe to my undergraduate faculty is demonstrable correctness of computing implementations, a property of paramount importance thirty years later in the face of increasingly adversarial conditions.

Much concern has been expressed recently about how the computing hardware we use is designed, and the immense attack surface that is now at our our adversaries' disposal as a consequence. I'm not going to write much about this attack surface, as it is being extensively documented by others [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Instead, here is some work I can do in cooperation with my dissertation committee to offer a safer home for some of the world's automation, computation, and communication.

The security problems with today's silicon derive from root causes external to the hardware itself. We can talk about speculative execution, pipelines, cache misses, malformed input, unstable memory schemes, corrupt stacks, arithmetic overflow, dopant-level trojans, closed-source microcode, open system buses, proprietary firmware, counterfeit accessories, and privilege escalation. These technical opportunities superpose with the human circumstances of globalization, divergent interests, addiction to power, situational ethics, and economic inequality. But I believe the *root problem* with our CPUs is they are designed, assembled, and distributed without input from the community that uses them. If we hope to find solutions for our security concerns, a thoughtful look in a mirror might be a promising start.

A parallel malady has long existed in the software supply chain: software companies have been writing software for the purpose of selling it, rather than out of their own need to have or use their wares. The symptoms are all too familiar: needless changes and revisions made to already deployed software, premature end of support for widely used systems, easily foreseen vulnerabilities, cloud-based licensing, and subscription-only availability. In many domains, the open-source software movement has afforded relief for some users. In other domains, much remains to be accomplished. I believe that on the whole, community-sourced software has brought unprecedented freedom, security, and stability to a world of users. I am also confident that we can can achieve similar progress with CPUs, offering stability, security, and freedom instead of obsolescence, malfeasance, and monopolism.

I can only speak on my own behalf, but I think a consensus would exist that CPUs for sensitive applications need to be demonstrably our own. They must answer to no one other than their owners, exhibit no bad behaviors, and have a feature set that we find agreeable. Here are some basic postulates of satisfactory CPUs:

1. CPUs must not run operating systems. Instead, operating systems must run CPUs.
2. A CPU must not facilitate any exploit that can bypass any control of the operating system that is running it.
3. A CPU must run any and all adversary-supplied code without possibility for privilege escalation,[1] never violating operating system permissions or resource limits.
4. Included and attached hardware, as well as buses, must not facilitate any exploit that can bypass any control of an operating system that is running a CPU.
5. Attaching a CPU to a computer network must be no less secure than keeping the CPU air gapped.
6. A CPU must unconditionally protect all instruction pointers, including branch targets and subroutine return addresses.

1 A regrettable exception is needed for social engineering attacks, which by Rice's theorem cannot dependably be distinguished from fully mechanized attacks.

7. A CPU’s security must not be fragile in the presence of malformed input.
8. A CPU must mitigate unanticipated modular arithmetic wrapping without bloat, inconvenience, slowdown, or incorrect results.
9. A CPU must never give incorrect results merely due to unexpected signedness or unsignedness of an operand.
10. A CPU must support preemptive multitasking and memory protection, except for uses so simple that the application running and the operating system are one and the same.
11. A CPU must provide hashing, pseudorandom number generation, and cryptography capabilities consistent with its intended use.
12. A CPU must not depend in any manner on microcode updates for its continued security or suitability for use.
13. A CPU must be repairable by its owner, particularly with regard to on-site replacement of components or stored data that the owner might foreseeably outlive.
14. A CPU must be replaceable by its owner in its as-used form.
15. A CPU must be delivered with objective, verifiable evidence of conformity with these postulates.

Simultaneously satisfying the above postulates is comfortably within human intellect to accomplish. I have two sorobans in my lab that generally meet the parameters of this list, even though they don’t use electricity to do arithmetic. Two TRS-80s here also run fine, and although lacking, conform closer to the above than the 64-bit machine that this proposal was written on.

Requirement 15 spawns two corollaries:

16. All components of a CPU must be available, affordable, fungible, and inspectable.
17. A CPU must not contain unknown, undocumented, unreadable, or unauditable circuitry, firmware, or microcode.

Fungible means if a component can't be obtained from one supplier, a different component that will work equivalently can be obtained from a choice of suppliers. *Inspectable* means the manner of operation of a component can be verified by the owner in as much detail as desired.

The requirements I've laid out aren't wistful parameters to secure this committee's go-ahead for investigative daydreaming. I have identified specific, constructible, elegant mechanisms for such needs as memory protection, arithmetic wrap-around, efficient hash functions, and alternatives for electrolytic capacitors. Not all are within the scope of this proposal, but I stand ready to back up all of this document's functional demands.

If our suppliers aren't ready to assent to the above postulates, we can build superb CPUs independently. But citizen-builders need to be prepared to trade away two things. First, our CPU designs will depart from historic norms. They won't run MacOS or Microsoft Windows. They won't play DRM-protected copies of Angry Birds. They might never run OpenBSD or Android, and I hope they won't become compilation targets for GCC or LLVM, as neither is small enough to plausibly audit against backdoors. But if users commit to this concession, suitable operating systems, toolchains, and tools will appear.

The second concession we need to build our own CPUs concerns their dimensional quantities. We won't get the speed, energy efficiency, memory density, economy, word size, or parallelism that a semiconductor fab delivers. Instead, we'll get security and ownership characteristics that we need more than these dimensional quantities. The dimensions are merely *numbers*, and as long as our CPUs are suitable and effective as used, it doesn't matter what their performance metrics come to. As a computer-literate undergraduate, I was content to read my email on a VAX. I didn't need to know that its clock speed was 3125 kilohertz.

1.2 Practicality and economy of building CPUs

Building a CPU requires an assembly process we can manage, and components we can live with. Our selections need to be available to most technical people with ordinary means. That rules out designing our own VLSI, but multilayer circuit board manufacturing is available in small runs at nominal cost, as is surface mount assembly. The change in capital scale is forceful: manufacturing technology costing less than $1 million displaces a semiconductor fabrication plant that costs over $1 billion. This vastly expands the options for assembly plant ownership, siting, and use.

The cost to use an assembly facility will not exclude many builders. Some new reflow ovens cost less than $300, and yet-smaller-scale soldering methods exist. Yet larger, automated assembly lines with excellent controls can be hired at even less cost than donated time using the cheapest tools. The price to have a "one-off" multilayer circuit board assembled, including manufacture of the board itself, can be less than $100 over the cost of the components used. The whole transaction can be a turnkey process where CAD files are uploaded, and the specified CPU arrives via mail.

An eighth-grader with a web browser and a debit card can build a CPU. This is a great shift from historic norms, when CPUs were assembled using wire wrap on boards plugged into backplanes in cabinets. Although CPU designs shrank into fewer and fewer discrete packages to reduce assembly costs, assembly costs themselves ultimately collapsed. The least expensive time in history to build CPUs "the hard way"—assembled from separate packages outside of a semiconductor plant—is today and tomorrow.

1.3 Logic family selection

The electronic component industry has few patrons who want parts to build CPUs. The inventory is meant for other purposes, so for right now we have to be creative and make do with what we can obtain. Like MacGyver on Monday nights, we need both purposeful intent and an open mind to use what's available.

Almost all CPUs made out of anything except silicon wafers lately have been somebody's avocation; however, the research of this proposal was not undertaken for hobbyists. Trustworthy CPUs are needed to control dams, fly planes, protect attorney-client privilege, mix chemicals, leak secrets, coordinate troops, transfer funds, and retract the five-million-pound roof of Lucas Oil Stadium at the option of the Colts. All components selected must be for sale at scale, not scrounged from scrap, and they must remain available for the expected duration of manufacturing and servicing.

Here is a list of logic families we might be able to procure and use. Neither practicality nor seriousness was a requirement to appear on this list, because every choice here has important drawbacks. It is better to start with an openly imaginative list than to overlook a meritorious possibility.

Electromagnetic relays have switching times between 0.1 and 20 ms, are large, costly, and have contacts that wear out. Relays generally have the wrong scale: a practical word processor will not run on relays. Relays offer certain benefits, such as resistance to electrostatic damage, and purring sounds when operated in large numbers [12].

Solid-state relays, including **optical couplers**, can compute, but more cost-effective solid-state logic families are readily available.

Vacuum tubes have faster switching times than relays, but are large, costly, and require much energy. Like relays, their scale is wrong in several dimensions. Commercial production in the volumes needed does not exist today. Power supply components might also be expensive at scale. Ordinary vacuum tubes wear out quickly, but special quality tubes have proven lifespans of at least decades of continuous use [13].

Nanoscale vacuum-channel transistors may someday work like vacuum tubes, but at present are only theoretical.

Transistors in individual packages are barely within scale. The VML0806 package size is the smallest available, measuring 0.8 x 0.6 x 0.36 mm. An advantage to using discrete transistors is that no component sees more than one bit position, so slipping a hardware backdoor into the CPU unnoticed would be particularly difficult.[2] Finding transistors with desirable characteristics for CPUs might not be possible now; the MOnSter 6502 is an 8-bit CPU, but it can only operate to 50 kHz due to component constraints [15].

7400 series and other **glue logic** has largely been discontinued. NAND gates and inverters aren't a problem to find, but the famed 74181 4-bit ALU is gone, the 74150 16:1 multiplexer is gone, etc. Most remaining chips have slow specifications, obsolete supply voltages, limited temperature ranges, through-hole packages, and/or single sources. 4-bit adders, for example, are still manufactured, but their specs are so uncompetitive as to be suggestive for use as replacement parts only. Counter and shift register selection is equally dilapidated. Even some leading manufacturers are distributing datasheets that appear to be scanned from disco-era catalogs.

Current-mode logic offers fast, fast stuff with differential inputs and premium prices. Around $10 for a configurable AND/NAND/OR/NOR/MUX, or $75 for one XOR/XNOR gate. Propagation delay can be under 0.2 ns. Power consumption is high. For ordinary use, parallel processing using slower logic families would be cheaper than using today's current-mode devices for sequential processing.

Mask ROM requires large runs to be affordable, and finished product must be reverse engineered to assure against backdoors. Propagation delay has typically been on the order of 100 ns, probably due to lack of market demand for faster products. If anyone still makes stand-alone mask ROM, they are keeping very quiet about it.

EPROM with a parallel interface apparently comes from only one company today. 45 ns access time is available, requiring a 5V supply. Data retention is 10 years in vendor advertisements, but omitted from datasheets.

EEPROM is available to 70 ns with a parallel interface. Data retention is typically 10 years, but one manufacturer claims 100 years for some pieces.

NOR flash with parallel interfaces is suitable for combinational logic, offering

2 One possible backdoor would be to install several RF retro-reflectors like NSA's RAGEMASTER [26] in parallel, or a single retro-reflector in combination with a software driver.

speeds to 55 ns. Storage density is not as extraordinary as NAND flash, but 128M × 8 configurations are well represented by two manufacturers as of early 2020. Although access time is much slower than static RAM, the density offered can make NOR flash faster than SRAM for uses like finding transcendental functions (logs, sines, etc.) of single-precision floating-point numbers. Data retention is typically 10 to 20 years, so these devices must be periodically refreshed by means of additional components or temporary removal from sockets. Few organizations schedule maintenance on this time scale effectively. Also, because no feedback maintains the data stored in these devices, NOR flash may be comparatively susceptible to soft errors.

One use for parallel NOR flash could be for tiny, low-performance microcontrollers that are free of backdoors. We will need exactly such a controller for loading microcode into SRAM-based CPUs. Here again, a servicing mechanism would need to exist at the point of use on account of NOR flash's limited retention time.

NAND flash is available with parallel interfaces, but data and address lines are shared. These devices aren't directly usable as combinational logic. Serial NAND flash with external logic could be used to feed microcode into SRAM-based ALUs. Periodic rewrites are required as with NOR flash.

Dynamic RAM, or **DRAM**, does not have an interface suitable for combinational logic. This is in part because included refresh circuitry must rewrite the entire RAM many times per second due to self-discharge. Although standardized, DRAM interfaces are very complex, and datasheets of several hundred pages are common. DRAM is susceptible to many security exploits from the RowHammer family [16], as well as to cosmic ray and package decay soft errors. The sole upside to DRAM is that an oversupply resulting from strong demand makes it disproportionately inexpensive compared to better memory.

Static RAM, or **SRAM**, has the parallel interface we want for combinational logic once it has been initialized, but is not usable for computing at power-up. It is necessary to connect temporarily into all of the data lines during system initialization to load these devices. Facilitating these connections permanently increases component count and capacitance.

As a logic family, static RAM’s main selling point is its speed. 10 ns access times are very common, with 8 and 7 ns obtainable at modest price increases. Price is roughly 600 times than of DRAM as of 2020, around $1.50/Mibit. As a sequential logic family using standalone components, SRAM offers the best combination of cost and computation speed available today. As main memory, SRAM’s decisive selling point is natural immunity from RowHammer and other shenanigans.

SRAM’s ability to provide program, data, and stack memory at the required scale, abundant registers, and for some designs cache memory, is a characteristic not afforded by the other logic families. This means that regardless of what components we select for computation, our design will include SRAM for storage. This storage will have to be trusted, especially in light of the global view of data that the SRAM would be given. If our trust of SRAM for storage is not misplaced, then also using SRAM for computation might not expand the attack surface as much as adding something else instead.

Programmable logic devices, or **PLDs**, and **field programmable gate arrays**, or **FPGAs**, can implement CPUs but are not inspectable, are auditable, not fungible, ship with undocumented microcode and potentially other state, have a central view of the entire CPU, and have a very small number of suppliers controlling the market. They are amazing, affordable products with myriad applications, but they might also be the ultimate delivery vehicle for supply chain backdoors. They are easily reconfigured without leaving visible evidence of tampering. I would not employ these when security is a dominant need.

1.4 SRAMs as electrical components

Static RAM is confusingly named, as here *static* means the RAM does not spontaneously change state, not that it uses a static charge to hold information. Dynamic RAM uses a static charge, and therefore spontaneously forgets its stored contents and requires frequent refreshes. SRAM is said to be expensive, but this is only compared to DRAM. SRAM generally uses four to ten transistors to store each bit, while DRAM uses one transistor and one capacitor. SRAM is about 600 times as expensive per bit, so factors beyond transistor count influence price. Even so, the price of SRAM has fallen more than 100-fold during the period I've been writing software.

Asynchronous SRAM is available in 2020 in sizes to about 32 Mibit, with access times around 10 ns. For more money, 7 ns is offered. An organization named JEDEC has standardized pinouts not only between manufacturers, but also across SRAM sizes. With careful planning, one can make boards for computers in advance, then wait for an order before deciding what size RAM to solder on. Current trends favor 3.3 volt power, operating temperatures from −40 to +85 C, and a certain but imperfect degree of component interchangeability. The JEDEC pinouts are somewhat undermined by manufacturers ensuring they offer non-overlapping—although standardized—package geometries.

At least one manufacturer offers error correction code, or ECC, as an optional feature for its SRAM.[3] These RAMs with ECC are plug-in-replacements for ordinary RAMs, so it's possible to add ECC to a system design even after circuit boards have been made and early models have been tested. There isn't really a price or speed penalty for specifying ECC, but it definitely limits supplier selection and appears to preclude dual sourcing. Also, as ECC RAM packages are only visually distinguishable by supplier markings and are not operationally distinguishable, it would be prudent to verify ECC's presence (by electron microscopy or comparing error rates in the presence of noise or radiation) in samples prior to claiming that computing hardware was built using ECC RAM.

If a path through a CPU is long enough to warrant pipelining, synchronous SRAM (that is, with clocked input and/or output) can be used for this purpose. The designs presented in this paper have short paths, no deeper than three gates except for long multiplication, and will run fastest if not clocked. Nonetheless, synchronous SRAM could be beneficial at circuit boundaries if it eliminates a need for external

3 This particular SRAM uses four ECC bits per byte stored.

latches.

Because much RAM-based ALU logic will produce outputs just a few bits wide, one might be tempted to drop the 64k × 16 size that is frequently seen in this proposal down to 64k × 8 bits. There are a few reasons this isn't practical, but a key one is no one bothers to make the narrower size. This omission may seem foreign to early microcomputer users who dreamed of someday upgrading their systems to 64 kilobytes. Although 16-wide RAM uses double the transistors than we strictly require, there are tangible benefits to having the extra width. Fan-out gets heavy at various points, and being able to double up on some outputs may spare the cost and propagation delay of additional buffers. There are also situations where information must be duplicated between two otherwise segregated buses. The redundant outputs already included in today's SRAM make this need for isolation easier to accommodate, particularly given that 64k × 16 RAMs have separate enable lines for their upper and lower data bytes.

The chief drawback of using RAM as a building block is that what we actually desire is ROM. The contents of function-computing memory only need altered if the architecture or features change. There are a couple of problems with ROM, to include EEPROM, write-once EPROM, and related products. Foremost, ROMs are extremely slow, so much that most models are actually manufactured with serial interfaces. They aren't for use when speed counts. The fastest parallel EPROM I've found offers a a 45 ns access time, contrasted with 10 ns for inexpensive SRAM.

The parallel interface we need to compute with SRAM is a problem at power-up, because no facility is included to load the data. The forward-feed nature of SRAM combinational logic makes it possible to progressively set many address lines to desired locations as initialization continues, but nothing connects within the layers to access the data lines—note these are separate from and work differently than the address lines—and write the memory. An alternative serial interface like JTAG could do this, but JTAG is only available on large, complex, synchronous RAMs. The smaller, simpler SRAMs we need aren't available with JTAG ports. What we have to do instead is connect extra components into every isolated node, and these nodes number in the hundreds. These startup components and their connections will consume power and board space, and worst of all, add capacitance to the CPU and slow it down nominally. But even with this bad news, SRAM is still the best logic family we know of to build CPUs on our own.

Analog switches have been identified which might be appropriate for feeding lookup tables to nested SRAMs during power-up. These include[4] the PI3B3253 from Diodes Inc. and IDT's QS3VH253, and are available with footprints down to 3 × 2 mm. The capacitance of either of these devices is of similar order to that of another RAM pin or even two, so there is a penalty. Each package has two copies of a 1-to-4 pass transistor mux-demux with an all-off option, so reaching the data lines of a 64k × 16 RAM would require roughly two switch ICs on an amortized basis, plus additional logic farther away from where computations are made. Cost varies widely by model but can approach $0.25 per switch chip for some. So initialization hardware costs as of now will add about 50% to our SRAM cost.

One benefit of installing these analog switches is data can flow in either direction when they're in the on state. We can not only write to individual RAMs within a CPU or ALU, but read back their contents for diagnostic and development purposes. Of course we can build a backdoor this way, but we're not interested. Such a backdoor would also be plainly visible. Concerns that this extra path to the RAMs may present a security concern may be mitigated by adding a non-resettable latch that locks out reads and writes once all tables are loaded and the system is ready to compute. This same latch might also enable the CPU clock: if the system can run at all, SRAM access via the initialization circuitry is locked out for as long as the power stays on. Including an LED indicator could promote further confidence.

Cheap tricks can reduce the extent of what SRAMs see when used as registers and main memory. For example, odd bit positions can be stored on one chip, and even positions on another. The security benefit to this might be more psychological than actual, but as a general principle every component's access to information

4 Analog switch models and manufacturers mentioned are to facilitate comprehension only. I have tested none, and I endorse neither.

should be kept at a minimum, especially when a precaution costs nothing.

As SRAM fabrication processes vary considerably, there exists at least a factor-of-four variance in power consumption among chips rated for the same speed and supply voltage. Fan-out, capacitances, package type, and other characteristics need careful consideration. Assuring continual availability will require being ready with multiple sources. As a manufacturer change often forces a package change, a variety of board layouts should be prepared in advance of supply chain surprises. SRAM for data memory can be hard to specify, because availabilities lapse as one tries to establish and maintain a range of size options from more than one manufacturer.

1.5 Proposal's focus on ALUs

Although the justification and consequence of our research applies to democratizing CPUs, the work proposed for my dissertation focuses on arithmetic logic units specifically. There are three reasons. First, there is a distinction between the technology an ALU comprises and the technology of the remainder of its CPU. These differences give rise to a clear milestone, stopping point, or "done state" once an ALU is mature. Second, significant differences separate the methods of operation and construction of mainstream ALUs and elegant SRAM ALUs. Surprisingly, the SRAM ALU family appears to be new to the world; at a minimum it's unfamiliar and obscure. These ALUs are due extra attention because of their novelty, complexity, and consequent rapid growth in new knowledge. Third, the characteristics and design specifics of SRAM ALUs leak outward into otherwise unexpected requirements for the CPU surrounding it.

To elaborate on the first reason, the principal architectural difference between an ALU and the balance of its CPU is that the ALU uses combinational logic, while the balance uses sequential logic. Yes, sequential ALUs with microsequences, bit-serial operations, pipelines, and other transient state exist in other domains, but SRAM is so capability-rich that their general-purpose ALUs benefit little if at all from sequential logic. Another reason we need to keep ALUs free of sequential logic is preemptive multitasking. We seek a CPU that can switch between programs very quickly while being easy to manufacture from scratch, so we want to preclude any need for extra circuits for saving and restoring intermediate ALU state. Another architectural distinction of ALUs is that its regular structure facilitates building the entire unit from static RAM, while the balance of the CPU will need a broader range of components. Focusing research on ALUs specifically helps keep the magnitude of our scope manageable.

As to our second reason for focusing on ALUs, atomic operations in the SRAM logic family operate on small subwords, instead of on individual bits as a NAND gate or full adder might. These atomic operations execute in the same amount of time, regardless of which operations are chosen, and irrespective of the time and logic resources that would needed to implement them using other logic families. So for comparable expenditures of intuition and components, we should expect more capabilities from ALUs built using SRAMs than ALUs built out of other families.

If this research is to be contributing, we need to avoid loading ALU designs for other logic families into SRAMs, and reporting: *See, we can do this with SRAMs! Fewer parts! Easy assembly! Brilliant!* Of course we can do that with SRAM. The lower part count is an unavoidable consequence of SRAM being VLSI. This approach throws away the extraordinary capability offered by SRAM, while instead abusing it to emulate 1960s / 1970s / 1980s memes. It’s like locking down Windows 10 to only run Atari 2600 cartridges. But if we aren’t to copy the past, then designs are needed that tap the power of SRAM for ALU implementations. My searches for such designs in the literature found nothing. Elegant SRAM ALUs might not exist in the literature at all, in which case we have to invent them. I don’t invent things quickly, so it’s prudent to limit the scope of this research to ALUs.

As I mentioned in reason three, it’s not a good idea to design a CPU with a metaphorical socket that its matching ALU will plug into, on the basis of prior understandings as to how ALUs should work. I am aware of neither literature nor historic systems that offer any hints as to what SRAM ALUs need from the surrounding system. It looks deceptively easy: bring inputs from the register file, bring the control word, out go the new flags, and *voilà!* everything is awesome. But what if it turns out an SRAM ALU shouldn't generate carry or overflow flags? These flags worked fine in the 1960s, but they are a giant problem today and factor significantly into this proposal. An insightful ALU would clean up these flag issues, so its surrounding CPU will need a new groove. And that register file, what's its word size? Billions of machines with a half century of history vouch for 32 bits. 16 bits will usually be inadequate, and 64 bits will usually be excessive. But here again are old assumptions. It turns out that SRAMs have affinities for unexpected word sizes, and once we tabulate some benefits and drawbacks, we might not select a multiple of eight bits, let alone a power of two. So here are two brief examples, flags and word size, that warn us not to get entangled in CPU design until the ALU design is very mature.

1.6 Questions for research

Here are several questions that I have framed to validate and preserve our work, as well as to establish a clear milestone as to the present state of the art for SRAM ALUs. Not only will these answers be immediately reproducible and directly usable by future implementers and researchers, but they will also help persons who might be engineers, users, journalists, or policymakers understand the benefits, applications, and limitations of this emerging information security management technology.

1. What are some essential logic blocks of SRAM ALUs?
2. How can these logic blocks be combined in an elegant design?
3. What evidence can be offered in support of a design?
4. Is there a yardstick for assessing the strength of further schemes?
5. Have we produced design tools that would accelerate future work?
6. What are other dividends of using the static RAM logic family?

Work has been done to estimate some of the answers. For Question 1, essential logic blocks are known to include simple lookup elements, arbitrary geometry adders, carry-skip adders, swizzlers, semi-swizzlers, logarithmic shifters, substitution-permutation networks, and multipliers. But except for simple lookup elements, and bolt-on, stateful substitution-permutation networks outside of a general purpose ALU's ordinary datapath, these logic blocks when implemented using SRAM are not known to—or are scantly known by—the literature today so far as I have been able to ascertain. These essential blocks are described more specifically later in this proposal in enough detail to make their construction straightforward.

Question 2 pursues a tangible, testable ALU design as opposed to a methodology. This is for the benefit of our readers. Someone who is new to airplanes can learn more about them by flying actual ones than by reading nineteenth-century flight literature. This relates to Question 4, which pursues an answer in the form of "here are some benchmarks for the ALU we are sharing with you." Generalized or intangible answers to Questions 2 and 4 could leave a reader either replicating discoveries already made, or not grasping key concepts.

Question 3 is to measure both correctness and suitability of the design that follows from Question 2. It's a practical application of this work, and we'll get to do some real testing. The correctness portion of the test involves writing test cases separate from and independently of the earlier design work, and using simulation to evaluate these test cases. I have seen enough preliminary design to know these tests are necessary. Designing an ALU with a particular use in mind is at least NP-hard.[5] There are many opportunities to run out of space in a table, not have enough address lines, or to confuse the semantics of signals. These are issues that would be far easier for me to correct once, than for later readers who unwittingly look for truth in my mistakes to stumble over repeatedly.

The suitability test for Question 3 will take our design once it has passed checks for correct computation, and evaluate whether it should nonetheless be annotated or improved prior to publication. Did we fail to incorporate valuable

5 Without an application in mind, ALU design is an undecidable problem.

features? Can the instruction set be more practical? Will a potential user regret that I hadn't spent more time trying to use the ALU myself? The suitability test I propose entails writing an estimated 10 000 lines of assembly code for a simulated CPU extrapolated from the ALU design of Question 2. The program I write will be a *self-hosted assembler*, meaning it's an assembler that when given its own source code as input, produces its own executable as output.[6] This coding task will provide our suitability test a good range of data structures and algorithmic tasks to implement. This method of assessing suitability will be subjective and not perfectly reproducible, and is likely to find in favor of suitability, but it will shake out bugs, identify wish list items, test for sufficient stability to allow a surrounding CPU to be designed, and vouch that the research is of high quality and advances the state of the art.

Question 5 asks what I can leave behind for others who might be interested in SRAM ALUs. First of all, the working models from Questions 2, 3, and 4 will be placed in a public open-source repository. People will be able to try, modify, and assemble code for this ALU. But there are a couple more tools people will discover that they want, and I need them for my own investigative work as things stand. All SRAM ALUs are hard to design optimally with chosen constraints on memory sizes, but carry-skip adders are particularly tedious. I think I can build a 55-bit adder in two layers using 64k × 16 RAMs, but at the time I tested a simulation, I only thought I could reach 45 bits. I had made a miscalculation, but at least the 45-bit test was successful. It would be useful to be able to have fast answers to what-if questions concerning adders that minimize the risk of such mistakes. I also think that still using 64k × 16 RAMs, we can build 256-bit adders in three layers, but probably not 512-bit adders, and I have no tools right now other than pen-and-paper binary search to determine where that size boundary is. Question 5 seeks a software tool that answer such inquiries expediently and confidently, verifying any answers given with confirming simulations, and providing netlists for potential fabrication. SRAM multipliers are even more complex: the reason Section 2.9 illustrates how 32-bit multipliers can be built is that when I crunched the figures, I had a 32-bit CPU i mind. Today I prefer a

6 All software that I've specified to be written is comfortably within my ability.

different word size, so the previous work needs repeated. I’m not certain yet whether the new word size will change the multiplier’s speed until I have the updated design. This time I know to automate the process! Also, multipliers for real CPUs will need to accommodate both signed and unsigned numbers, meaning some place values are going to have negative weights. Those of us who don’t reckon like Gauss will require software to keep our arithmetic and wiring straight.

Question 6 asks for us to look beyond the logic elements that come out of Question 1, to see if we can add more value to our ALU by tapping the extra computational heft that SRAM affords us. It’s an easy observation that an ALU can be dual-purposed to provide offset calculations during loads and stores in RISC architectures, although this is not unique to SRAM. But we can add yet more freebies to our instruction set without adding any more integrated circuits. I expect these to include counting operations such as Hamming weight, leading and trailing zeros and ones, permutations within subwords, permutations within transposed subwords, “short” multiplication of words by subwords, and an opcode I designed for manipulating pointers that replaces the rightmost portion of a word in one cycle. I also anticipate one-cycle minimum and maximum instructions. These last don’t sound very innovative, until we consider that x86 CPUs don’t offer minimum or maximum at all for scalars. Our instructions will not only compute minimum and maximum, but do so even for inputs of arbitrary signedness. I further expect we’ll be able to test for equality in one cycle correctly, even if one of the words is signed and the other is unsigned. This is not a common feature of instruction sets and might be original to this work. I also anticipate being able to do power-of-two multiplications by means of left arithmetic shift—this is perhaps universally implemented wrong, because most wrap-around cases are not tested for. But I believe our SRAM ALU will do left and right arithmetic shifts perfectly, always identifying overrange results without false negatives or false positives. The same will be true for addition and subtraction, even if operands are not of the same sign. Thus for Question 6, I plan to keep looking for these opportunities to set new standards for ALUs by leveraging what SRAM can offer.

2. Essential logic blocks of SRAM ALUs

2.1 Hierarchy of ALU capabilities

Very simple ALUs provide addition and a functionally complete set of bitwise boolean operations. The “complete set” may be as small as one function, meaning it is possible to build an ALU with, for example, NOR and addition as its only operations. For practicality reasons, more than one boolean operation is usually included, and subtraction is often also included. But in order to use time and program memory efficiently, more complex ALUs are preferred.

Rotate and shift operations offer the first progression of ALU upgrades. This progression starts with shifts of one bit position, then multiple positions using a counter, and ultimately any number of bit positions in constant time. Doublewords rotations and shifts appear early on as ALUs mature, especially when a particular register is designated as “the” accumulator. Although convenient, doubleword rotations and shifts don’t enhance performance much for most programs. It’s more that they are very easy to build when it’s always the same two registers that rotate.

The next step up for ALUs is hardware multiplication. Typically two words are accepted, and a doubleword product is produced. At this step, signed and unsigned genders of operation are offered, but combinations of signedness are not offered. The foremost need for multiplication is to locate array elements that aren’t a power of two size. Serious numeric computing also needs multiplication, but arrays will appear in a much broader range of applications. The distinction between multiplying for arrays and multiplying for numeric processing becomes important, because finding array element locations can usually be done using short multiplication, implying that many machines will do fine with only a short multiplier.

A lot of software doesn’t require hardware multiplication at all. CP/M, WordStar, and Pac-Man are historic examples that appeared between 1974 and 1980, when the 8-bit CPUs they targeted offered no multiplication opcodes. On the other hand, multiplication and division instructions had long been standard on computers with bigger footprints: the IBM 1130 and DEC PDP-10 are examples of mid-1960s systems with instructions to divide doublewords by words. The hold-up with early 8-bit microprocessors was they were critically short on die space. Zilog's Z80 only contained a 4-bit ALU, which was applied twice in succession to do 8-bit work. As dies came to support more gates, 16-bit processing, multiplication, and division simultaneously became practical. As a rule, these three capabilities came as a set.

The next stage of ALU robustness adds word rearrangement. The ability to whip subwords around outside the linear order of shifts and rotations is combinatorially difficult to implement, because the number of potential rearrangements is larger than the word size can represent. Even rearranging an 8-bit word requires a 24-bit argument if we are to have full selection including bit duplication. If only 8-bit permutations are supported and no bits are duplicated, a 16-bit argument is still needed, and fancy logic or a big table has to be added to decode it. Yet rearrangement within words comes up periodically, often with foreseeable gyrations, and ALUs that provide the common ones in constant time are desirable. In the x86 family, the 80386 introduced some sign extend instructions, the 80486 added a byte swap for fast endianness modification, and BMI2 recently brought parallel bit deposit and extract.[7] Word rearrangement is useful for serialization, floating-point routines on integer-only CPUs, swizzling for graphics, evaluating chess positions, hash functions, pseudorandom number generators, and cryptography. Where practical, ALUs should aid these operations rather than stand in their way.

Another family of ALU operations to consider is cryptography. Today's computers connect to networks, most of which are shared, and many of which are wiretapped. To be suitable for use, a computer must encrypt and decrypt fast enough to keep up with its own network traffic. Even if a computer will never be attached to a network, it's still likely to need unbiased pseudorandom number

7 Bit Manipulation Instruction Set 2 arrived with Haswell in 2013.

generation, effective hash functions, and protection (by encryption, information splitting using a random variable, or otherwise) from storage devices manufactured overseas.

With the above wish list for ALU features, here are some SRAM building blocks to deliver the goods. All exist in the literature using conventional logic families, but I have not found any literature to suggest implementation using RAM, ROM, or any kind of lookup table. Perhaps no one wants credit for giving away three orders of magnitude for execution speed, let alone an even larger spike in transistor count. But there are applications for which I would be willing to wait nanoseconds rather than picoseconds to assure a CPU’s conformance with the postulates of Section 1.1.

2.2 Simple lookup elements

Simple lookup elements are the fundamental building block of all SRAM logic. Remember that our decision to use RAM has nothing to do with its mutability: we actually want ROM, but RAM is much faster and does not require custom masks. The overall constraining parameter is the number of input bits, which is the base two logarithm of the number of rows. We need enough inputs bits to select among all operations the RAM supports, plus these operations’ inputs which are generally one or two subwords and sometimes a carry bit. Plate 1 shows an example with two logical and two arithmetic functions. Plate 2 show a another problem type that can be solved using simple lookups.

Two frequent design tradeoffs arise when using simple lookup elements. First, unary operations can offer input widths of twice what binary operations can. Second, the presence of a carry input tends to halve the number of operations a RAM can provide for a specific operand configuration.

2.3 Arbitrary geometry adders

Arbitrary geometry adders are a special case of simple lookup elements. Plate 2 shows two addition problems with more than two addends and irregular column spacing. In the right example, there's either a bit missing within an addend or a line with more than one addend, depending on interpretation. Both examples share a non-obvious feature of exactly 16 input bits, because each is a one-RAM subtask from the binary multiplication demonstration in Section 2.9. There is nothing tricky, just tedious, about how these two RAMs are programmed: some kind person computes all 2^{16} possibilities using the place values shown.

Simple RAM lookup elements are all over the computing literature and especially prevalent in PLDs and FPGAs. The case of arbitrary geometry adders using SRAM isn't in widespread use or discussion known to me, but relates to and conceptually derives from the full- and half-adders used in Wallace [17] and Dadda [18] multiplier circuits since the mid-1960s.

2.4 Carry-skip adders

Carry-skip adders combine subwords by augmenting not-always-certain carry outputs with a *propagate* output that is set whenever the correct carry output is indicated by the carry output from the immediate right subword. Plate 3 shows the carry outputs that will occur when two 3-bit subwords are added with a possible carry input. In practice we add wider subwords, but this size keeps the illustration legible. The key concept is that at times, the carry output cannot be computed until the carry input is known, and when this occurs the carry output is always exactly the carry input. So if two machine words are added by means of parallel subword additions, subword place value results must be determined later than subword carry results. The circuit responsible for this is called a *carry-skip adder*, and a 12-bit example combining 3-bit subword additions appears as Plate 4.

As Plate 4 shows, the number of stages to consider increases as we move leftward; this is like a spatial transposition of the time sequence of ripple-carry adders. Traditional carry-skip adders have a time-sequential propagation of carry information from right to left, but they move by subwords and therefore offer a speed benefit over ripple-carry adders. SRAM carry-skip adders go a step further:

they don't exhibit any right-to-left time behavior at all, but instead use their table lookup power to simultaneously make all carry decisions. This is unique to table-based carry-skip adders, about which I have found no reports in the literature.

SRAM carry-skip adders can be categorized into designs that use either two or three gate delays. A distinguishing characteristic of 2-layer adders as pictured in Plates 4–6 is that their carry deciding and final subword incrementing happen simultaneously. They don't determine the carry decision *and then* apply it, nor even compute the needed carry decision at all. They merely apply the correct carry decision as if it had been computed. Here, the magic of table-driven computation removes a stage.

Although SRAM carry-skip adders are always ripple-free and sometimes skip intermediate carry decisions as well, this doesn't make them universally faster than familiar carry-skip adders. Their zillion transistors of RAM are far slower than directly-constructed single-purpose logic like a word adder. So SRAM isn't going to revolutionize microprocessor[8] addition circuits. What SRAM *can* do is build reasonably fast adders for supremely secured and assured systems.

Numerous tradeoffs and alternatives appear in SRAM carry-skip adder designs. Plate 5 moves the old carry input to the first layer, saving much address space and a RAM in layer two, but crowding the least-significant layer one RAM more. This change also reduces potential to reuse this circuit for purposes other than addition—recall that these devices are merely RAMs, and they only become adders because of their contents. In fact, potential uses other than addition sometimes require that carry information propagate not from right to left as for addition, but from left to right.[9]

Plate 6 shows a "bidirectional" carry-skip adder that can propagate information in either direction. Note the choice to introduce the old carry at the second layer, and which SRAM the new carry is deemed to appear at depends on the direction of operation. Although bidirectional operation consumes more inputs at layer 2, their use flattens out among the RAMs and is in one respect a simplification. Although this

8 A *microprocessor* is a die that contains at least one complete CPU.
9 Left-to-right examples include NUDGE (see Section 4.7) and count leading ones.

circuit appears to be useful due to its directional flexibility, an ALU that supports the bit reversal permutation can mirror bits with a preliminary instruction, do the intended operation with ordinary right-to-left carries, and if need be mirror back the resulting bits. Considering the scarcity of occasions that a left-to-right carry is needed, using bit reversals instead—provided they are fast—might be a better design for some ALUs.

Plate 7 offers a remedy for the crowded propagate and carry wiring associated with 2-layer carry-skip adders. It isn't the wiring, of course, but the address space, memory, and transistor count for the high-place-value subwords which need conservation. Two ends of a candle are burning. First, the available subword size for computation soon reaches zero in the bottom layer, which the top layer must also mirror. This sets an upper limit on the total word size that can be added. Second, the adders in Plates 4–6 *only* add, due to the absence of input bits to select other operations. We need to find more address bits.

Plate 7 adds a middle layer to our carry-skip adder. Only one SRAM is needed for this layer, because it doesn't touch any of the tentative sums. It simply reckons the propagate and carry subword outputs into *carry decisions* that are passed to layer 3. The last layer of SRAM is much cleaner, with each RAM requiring only its own carry decision alongside its tentative sum. The conserved address bits can be used for other purposes, notably function selection, and perhaps another operand if a use for one emerges.

Switching to a 3-layer adder adds 50% to its propagation delay, a cost that requires justification. This extra delay isn't as harsh as it might sound, because additional delay exists in the datapath. The CPU's clock will lose at most 20% due to adding this third layer, subject to a few assumptions. As for justification, here are three items. First, a lot of address space opens up in the third layer, enabling other computation by these RAMs for non-additive instructions. Second, a lot of *time* opens up in the second layer, because Plate 7 has nothing happening to the tentative sums as the carries are being decided. The second layer can be filled in with more RAMs to do more computation with no further performance penalty. Third, the high fan-out in 2-layer adders for propagate and carry signals incurs a speed penalty.

One of the limitations of the adder circuits we've considered is their bandwidth limitation when computing across subwords. At best, the first layer can broadcast at most two bits per subword to the other subwords, and the three-layer adder can receive at most one bit from other subwords in the third layer. It would be helpful if we can route a lot of wires between subwords instead of within them, particularly if we can figure out a good system for using the extra connectivity.

2.5 Swizzlers

A **swizzler** is a layer of RAMs that operate on transposed subwords, meaning that each RAM gets one address bit from each subword, looks something up, and returns one bit to each subword. Plate 8 shows how this transposition is wired for a 16-bit word with 4-bit subwords. From right to left, each of the four RAMs operates solely on place value 1, 2, 4, or 8. If the four RAMs have the same contents and same function chosen, subwords will be treated as atomic entities. This is the case in the illustration: the operation here is copying the leftmost subword to the rightmost, and copying the inner left subword to the inner right. The four letters may be interpreted as either literal hexadecimal digits or 4-bit variables.

Some magic needs to happen in terms of advance planning, because the number of functions these RAMs can hold (and therefore execute) is a tiny sliver of the number of functions possible. Additionally, we need to consider that the RAMs might require unequal contents to achieve certain objectives. It's also possible that the function select bits applied might not be the same across all of the RAMs. It's even possible that there isn't an input bit that corresponds directly to each output bit: some outputs might be fixed ones, zeros, or some function of multiple inputs. It turns out that all three of these "mights" turn out to be very advantageous, although none are suggested by Plate 8.

2.6 Logarithmic shifters

A **logarithmic shifter** overcomes a key limitation of swizzlers, which are perfect for fast rotation of subwords, but not of bits. If we assign one bit each to letters a–p, we can swizzle `mnop abcd efgh ijkl` to become `abcd efgh ijkl mnop`. But when we try to rotate just one bit position from here instead of four, the result will be `ebcd ifgh mjkl anop`, because place values remain fixed. In fact, only the leftmost RAM would move anything, because the remaining transposed subwords are already correct. To finish our one-bit rotation, we have to clean up the subwords individually to yield `bcde fghi jklm nopa`, which is what we want. This requires a second layer of RAM that can operate within subwords rather than across them. Plate 9 shows this combination, which permits rotation of any number of bits in a single pass. Masking is easily added to the values in the RAMs to supply left and right shifts of any number of bits.

Three important properties pertain to logarithmic shifters. First, sign extension works out to support right arithmetic shifts: every RAM that needs a copy of the leftmost bit will receive it in time. Second, the RAMs within a layer need to all process the same number of bits. Third, the bits leaving the RAMs of layer one must be evenly distributed to the RAMs of layer two. Thus when the two layers do use the same number of RAMs, the subword size will be a multiple of the number of subwords. Equivalently, the word size will be a multiple of the square of the number of subwords.

2.7 Semi-swizzlers

A **semi-swizzler** or **half-shifter** is a contiguous half of a logarithmic shifter; that is, a transposition preceding or following a layer of swizzle RAMs, but no transposition going the other direction. An example appears at Plate 10, using the same input data and SRAM contents as Plate 8. The computed output looks like gibberish, because it is still transposed. A semi-swizzler can be applied twice by a program, using two consecutive instructions, to achieve ordinary shifts and rotations. Section 3 discusses the mechanism and tradeoffs of superposing a 2-layer adder with a semi-swizzler to build a compact ALU.

Because the RAMs of a semi-swizzler comprise both the first and second layer of a logarithmic shifter, the CPU word size must be a multiple of the square of the number of RAMs in the semi-swizzler.

2.8 Substitution-permutation networks

An **S-box** is an invertible substitution that can operate on subwords. Its purpose is to help progressively alter words in a key-dependent manner, until the alteration sequence is impractical to reverse without knowledge of the key that was used. Plate 11 shows a simple 4-bit S-box expressed in hex. Due to our requirement for invertibility, no value appears more than once in an S-box or its inverse.

A logarithmic shifter with its SRAM contents replaced by S-boxes is an instance of a **substitution-permutation network**, or **SPN**. Its intent is to scramble and unscramble bits by mixing data as it passes through layers of cross-connected S-boxes. SPNs are used for constructing hash functions, pseudorandom number generators, and ciphers. Desirable topologies for SPNs, as well as properties of cryptographically “strong” S-boxes, have been topics of secret research for half a century. Plate 12 shows a small SPN built using the S-box of Plate 11.

There is nothing novel about SRAM SPNs, but their mixing capability is very useful for ALUs to incorporate. They are best used under non-adversarial circumstances; e.g., to implement hash functions and PRNGs. Suitability for cryptography is considered in Section 4.9.

2.9 Fast multipliers

Fast SRAM **multipliers** are nearly as fast as adders in practice; however, the number of RAMs needed grows a little faster than the square of the word size. The process itself is primarily one of summing partial products quickly. As schematics of multipliers are tedious to draw and not particularly legible, here is a walkthrough of the process for 32-bit unsigned factors. The 64-bit product will be ready after only five gate delays.

Layer 1

The four bytes of each word are multiplied pairwise to form 16 results of 16 bits each. This requires 16 RAMs.

In the diagram here, bytes within a product are separated with colons to indicate they arrive from the same RAM. The colors used for each product matches the colors of the factors involved. The place values of each product are consistently aligned with the factors and each other. The diamond arrangement of these intermediate results might be unfamiliar: it accommodates the necessary width of the products and clearly indicates the distribution of bit positions that require summation.

```
                                      11100100 00100100 01101100 10100111
×                                     11111100 01001000 11111000 00011001
-------------------------------------------------------------------------
                             00010110:01000100
                    11011100:11100000 00000011:10000100
           01000000:00100000 00100010:11100000 00001010:10001100
  11100000:01110000 00001010:00100000 01101000:10100000 00010000:01001111
           00100011:01110000 00011110:01100000 10100001:11001000
                    01101010:01010000 00101110:11111000
                             10100100:01100100
```

Layer 2

256 bits of partial products are to be reduced to a 64-bit final product. This doesn’t take many iterations if the width considered by each RAM is kept as narrow as possible. Addition with carries is put off for as long as possible.

```
                             00010110 01000100
                    11011100 11100000 00000011 10000100
           01000000 00100000 00100010 11100000 00001010 10001100
  11100000 01110000 00001010 00100000 01101000 10100000 00010000 01001111
           00100011 01110000 00011110 01100000 10100001 11001000
                    01101010 01010000 00101110 11111000
+                            10100100 01100100
-------------------------------------------------------------------------
              00100 00   001 101 0010 11 01001     0100 010
  11100000            010000  001000  000110   0001010   0100100 01001111
        00 110100 0 11010  0 10011 01 1010 010 1011
```

In this picture, the bits above the line are an exact repetition of the previous layer’s output, but their colors now indicate which RAMs they are grouped into for addition. The rightmost place value of each color group can be observed to be the same before and after this operation. Each band can group as many as 16 input bits. The process used is the arbitrary geometry addition from Section 2.3, and is a simple table lookup. The black digits on the left and right are not being added, but are passed forward for later addition (left) or for direct use in the final product (right). The reduction in bits is considerably faster than one finds in Wallace [17] or Dadda [18] multipliers, because the SRAMs offer fuller capability than full- and half-adders. As this step starts, 8 bits are finished and 248 remain to add. Afterward, 13 bits are finished and only 99 remain to add. 15 RAMs are used.

Layer 3

The process of layer 2 repeats with 6 RAMs and further gains.

```
              00100 00   001 101 0010 11 01001     0100 010
  11100000            010000  001000  000110   0001010   0100100 01001111
+       00 110100 0 11010  0 10011 01 1010 010 1011
-------------------------------------------------------------------------
  1110           00 1110000     00100 011       1001000 01100100 01001111
     00000 1101010       010 01001     0100011 10
```

Layer 4

If arbitrary geometry addition as in layers 2 and 3 is continued here, the final product will be available after layer 6. But the remaining terms are simple enough to use carry-skip addition to finish in just five layers. The fourth layer is the first of the carry-skip adder. The `carry` and `propagate` outputs are drawn as colored in this text. We omit adding the seven leftmost bits so that only three RAMs are used by this layer.

```
   1110           00 1110000     00100 011       1001000 01100100 01001111
+     00000 1101010       010 01001     0100011 10
--------------------------------------------------------------------------
   11100000 11010100 1            0100 10000011 11001000 01100100 01001111
      000          0 01100010 01001
```

Layer 5

The final multiplier layer finishes the carry-skip addition. `Propagate` and `carry` signals replicate leftward so that all RAMs involved in this stage have what they need to reach the sum. The leftmost RAM would have been lightly used, so it’s able to pick up the seven bits that were skipped at layer 4. This layer uses three RAMs, and the complete multiplier uses 43.

```
        0
       00            0
   111000          0 0            0
+     00000 11010100 11100010 01001100 10000011 11001000 01100100 01001111
--------------------------------------------------------------------------
   11100000 11010100 11100010 01001100 10000011 11001000 01100100 01001111
```

To **multiply signed numbers**, the sign bit(s) of any signed factor(s) has a negative place value; e.g., bit position 7 of an 8-bit signed number has place value −128. The partial products and addition reductions thereafter will consume and generate a smattering of bits with negative place values. This won’t cause the SRAM logic elements any hardship: the only change will be to assure enough output bits for each RAM, group inputs in the correct address widths, and precompute its addition table correctly.

To **multiply numbers with any signedness**, build a signed multiplier with an extra bit position; e.g., build a 33-bit multiplier for 32-bit words, using the extra bits in layer 1 as a *signedness* bit instead of as a sign bit. This approach is better than applying the grade-school “a negative times a positive” rule to the entire problem. Although conceptually simple, the school method would either add many RAMs and almost double the gate delay, or use additional instructions for testing and branching.

The division of the original factors into subwords does not need to be symmetric, as long as all product and sum place values are grouped correctly. For example, an any-signedness multiplier for 32 bits using 64k × 16 SRAMs does not need to spill from 16 partial products into 25 in order to fit the signedness bits. One factor can have four subwords of [7 bits + signedness, 8 bits, 8 bits, 9 bits], and the other five subwords of [5 bits + signedness, 6 bits, 7 bits, 7 bits, 7 bits], for 20 partial products calculated by 20 SRAMs.[10] I have not counted the number of subsequent layers needed or the number of RAMs in each layer, as the work proposed will provide a tool to do so quickly. Another configuration would be to maintain 16 devices in layer 1 by enlarging some of the RAMs. This configuration adds 9 Mibit to layer 1, with unknown changes to the memory used by the other layers. Automating the design process will make it easy to identify multiplier configurations with the fewest gate delays and least cost.

10 This configuration could use mainly 32k × 16 and 16k × 16 RAMs, but the former sell for more money than 64k × 16, and the latter aren’t offered for sale. There is, at least, a slight reliability benefit in leaving much of a 64k × 16 RAM unused, as the opportunity for soft errors is proportionately reduced. There may also be some energy saved, depending on the cell design of the chips selected.

3. 2-LAYER ALU DESIGNS

3.1 An elegant 2-layer ALU for 36-bit words

A standard 64k × 16 RAM is functionally a dual, byte-wide device with shared address inputs. The two output bytes have separate enable lines and can be independently put into a high-impedance state. We can leverage this to build ALUs with very few components, by superposing a semi-swizzler onto the first layer of a carry-skip adder.

Plate 13 shows a 36-bit ALU made from just ten RAMs in two layers. These RAMs operate on 6-bit subwords to do all the work, and this illustration only shows the wiring for the subwords themselves. The various control signals are drawn separately for legibility as Plate 14, but note that the ten boxes refer to the same ten RAMs in both figures.

The top layer of the ALU does almost all the work. 16 functions accepting two inputs of six bits are stored in the upper bytes of the top-layer RAMs. The functions are selected via a 4-bit control input (Plate 14), and the byte select lines (not shown) enable the upper bytes while forcing the lower bytes to high impedance. This mode provides a traditional mix of additive and logical functions, as well as subword multiplications. There is no hardware support for 36-bit × 36-bit multiplication.

By enabling the lower bytes of the top-layer RAMs, and forcing the upper bytes to high impedance, the outputs are transposed in the manner of a semi-swizzler. These lower bytes provide 16 functions for swizzling, rotations, shifts, permutations, S-boxes, and other operations that benefit from transposition. The performance penalty is that the transposition only goes in one direction per instruction executed, so in nearly all cases it takes two CPU instructions to do anything with these lower bytes.

A second factor derates the speed of this ALU: the fan-out for propagate and carry signals peaks at seven, including startup hardware and a yet-unmentioned overrange output. This speed penalty is small enough that buffering would make the delay worse, yet large enough to increase the appeal of a three-layer design.

The bottom layer does nothing other than add whatever carries are required during addition and subtraction. All other functions, including the lower byte functions with transposed outputs, must force their propagate and carry outputs low so that the bottom layer does not spuriously change results. Also, although the bottom layer conceptually has five RAMs, the carry decision task for subwords Y_1 and Y_2 takes so few inputs that one RAM can settle both subwords.

This ALU does not *quite* finish all calculations in two layers. Because the subword outputs are not known until the bottom layer has finished processing, a zero flag is not yet computed for the whole 36-bit result. For this reason, five subword zero signals are shown emerging from the ALU. An external five-input NAND gate combines these signals into an aggregate zero flag. There will be enough surrounding delay in the datapath to absorb the NAND gate's delay.

The RAM on the bottom left has one more address line than the others to accommodate a carry output. This is a 2 Mibit, 128k × 16 device. The remaining RAMs are 64k × 16. This ALU has no carry input, because all address lines on the top right RAM are tasked for other uses. As the word size is already 36 bits, an occasional branch on the carry flag's value is typically enough support for adding and subtracting wider integers.

Table 1 maps out space in the first RAM layer for suggested operations of this ALU. This table's purpose is to show (i) an appropriate set of operations to implement, and (ii) that the complete set can be implemented within the address space available. Several provisions made in this table aren't immediately obvious, so here is some explanation.

This ALU always produces a 37-bit signed result when adding or subtracting irrespective of operand signedness, but the 2^{-36} bit, inaccurately termed the "sign bit," does not fit within a word and is not retained after the result is inspected for overflow. The inspection is fairly simple. If the 2^{-36} bit does not match the 2^{35} bit and the result is signed, overflow will occur upon truncation. If the result is unsigned and the 2^{-36} bit is set, overflow has already occurred. Overflow will not occur If neither situation applies. For signed results, the 2^{35} bit then becomes known as the 2^{-35} bit. All this would work out fine with 37-bit signed inputs, but what come from the registers are the customary 36-bit overloaded-signedness inputs. Our workaround is to have separate ALU operations for the several operand type combinations; this is why Table 1 requires three add and four subtract operations. The surrounding CPU will test for wrap-around and latch a Range flag when needed.

There aren't obvious identity, clear, or bitwise NOT operations for this ALU, but all three can by synthesized from XOR and constants. Short multiplication works across matching subwords and is always unsigned. Two cycles are needed, because six 12-bit results are produced. The low-order subwords are produced by the normal operation. The high-order subwords appear in the same place values as their operands, because that's where the RAMs are. This means that prior to adding subwords of a product together, the high subwords must be shifted six bits left. In anticipation of this, the high subword multiplication operation supplies the initial transposition for the shift, reducing by one the number of cycles needed.

The reader might wonder how overflow is avoided for short multiplication, as the result is potentially 42 bits. Usually nothing needs to happen, because the main use for short multiplication is to compute memory addresses of array elements. It's reasonable to expect the main memory to be SRAM, not DRAM. For a multiplication result to exceed 36 bits at 2020 prices, more than $400,000 in SRAM needs to be in the system.

Permutations, swizzles, and S-box operations usually require word transposition and are specified accordingly. Plain permutations within subwords also are supported. A set of 64 unary operations, selectable at the subword level, are included. One of these operations computes the fixed-point reciprocal $\lfloor (2^{36} - 1) \div \max(n, 1) \rfloor$, the largest unsigned word one can multiply by n without overflow. Most of the remaining 63 are simpler; a starting list of candidate operations appears as Table 21.

Shifts are rotations are implemented in two CPU cycles, because these require two permutation stages and two transpositions. The number of positions to move by must be replicated beforehand into each subword. Thus a rotation may take four cycles if the number of positions cannot be expressed as a constant: two to transpose, replicate, and transpose back the rotation amount, and two to accomplish the rotation itself. Negative shifts and rotations are unavailable, because the range −35 … +35 cannot be expressed within a 6-bit subword. As right rotations are expressible as left rotations, separate functions for right rotation are not helpful. When rotating right by a variable amount, no subtraction from 36 is required prior to replication into subwords, because one of the 64 available swizzle operations can combine the needed subtraction and permutation. No speed is lost by leaving out right rotation.

Plate 14 purposely omits some control logic that would easily confuse readers who are trying to understand this ALU for the first time. In addition to what is drawn, the propagate signals from the five RAMs which offer them are combined in a five-input NAND gate. The output of that gate is used only for left shifts, and indicates that signed or unsigned overflow resulted from a multiplication by a power of two. There are more subtleties. The rightmost RAM doesn't have an output bit left for overflow detection; it got used for zero detection. Here we are helped by a kludge. Cycle two of a left shift operates on transposed words, meaning that five of the six bits not checked during cycle one can be tested by other RAMs on cycle two. This leaves only the 2^0 bit not tested, a bit which can only overflow during shifts of more than 34 or 35 bit positions depending on signedness. Smaller fixed shifts needn't check the 2^0 bit at all. Of the remaining cases, the only shift amounts that can overflow *always* do when shifting a nonzero word.[11] The few programs that can't be written to preclude all possibility of such unorthodox shifts must assume responsibility for checking the 2^0 bit. But for most software, this ALU detects left shift overflow as perfectly as it detects additive wrap-around. Masking for left shift can't happen until cycle two, or overflow from the rightmost subword will go undetected.

The overrange conditions for addition, subtraction, and left shift are not

11 The surrounding instruction set implements logical left shifts as unsigned left shifts with the overflow signal suppressed. This is the ordinary case when the rightmost bit might shift more than 35 positions.

routine events comparable with the carry flag that other ALUs set. These conditions almost always involve processing errors, meaning that the indicating flag must only be cleared by an opcode exclusive to that purpose. A latching flag precludes the need to check after every arithmetic operation whether or not something bad happened. Instead, a long sequence of perhaps thousands of instructions can be run, followed by a single flag check prior to committing the result to a database or other observable outcome.

The parenthesized operations in Table 1 are useful but non-essential, and subject to displacement for good cause.

This ALU is not the most capable one a user can build on her own for high-confidence applications, but it already towers over the prior art. For uses where its speed and operations are suitable, anyone with the motivation can build or have built a practical, robust, inspectable, and tamper-evident ALU for 36 bits using ten small, fungible RAMs. I believe that on for the total component count, as well as the limited availability of parts that might be regarded as trustworthy, it would be difficult to improve on the overall design of this ALU. This device is conceptually mature, and the design frontier has moved to the CPU which surrounds it.

3.2 A tiny ALU for 18-bit words

The ALU of Plates 13 and 14 can't be cut down to build a 24-bit or 30-bit version, because 24 and 30 are not multiples of 16 and 25 (the squares of the number of subwords in the semi-swizzler) respectively. This means we cannot meet the requirement to have a self-inverse transposition and a single subword size at the same time. But we can build 12- and 18-bit versions, and these ALUs require only three or four RAMs respectively. The transpose wiring, shown for 18 bits on Plate 15, is rearranged to minimize loss of left shift overflow detection. Only the 2^0 and 2^1 bits are unchecked, so arithmetic shifts of up to 15 bit signed and 16 bits unsigned come with range checking. The right half of Plate 14 shows the control signals, where the worst fan-out (including two sinks not shown) is five. The instruction set can be as stated in Table 1, although the S-box outcomes will not agree with the results of their 36-bit counterparts.

4. 3-LAYER ALU DESIGNS

4.1 An elegant 3-layer ALU for 36-bit words

A three-layer carry-skip adder's middle layer is substantially empty, because the only thing happening is tabulation of carry decisions. By placing a swizzler in the middle layer's datapath, and overlaying a logarithmic shifter with the adder's first and third layers, we can build a robust ALU using 19 RAMs. Plate 16 shows this superposition. Several improvements over the two-layer design arise, at the cost of some additional hardware and approximately 15% added to the cycle time.[12]

Plates 17 and 18 show the general arrangement of a 3-layer, 36-bit ALU, with the latter plate showing how the bit slices are assembled. For clarity of reading, the transposition wiring across subwords is not drawn, but is identical to the lower-byte transposition on Plate 13. As the transposition is self-inverse and happens both before and after the middle layer, the outer layers see no transposition when the middle RAMs act transparently. The layers implement so many functions that they defy naming, so they are named α, β, and γ after their order of appearance. Their RAMs are numbered 0–5 from the right, so the notation α_5 designates the most-significant RAM of the first layer. The carry decision RAM, one in number, is denoted θ.

17 bits from the instruction decoder tell the ALU's RAMs what function to implement, of which 14 are visible on Plate 18.[13] These separate layer controls, plus additional control for θ's carry decision method, can get a lot of work accomplished in one pass. The right operand is fed to all three layers unmodified, allowing any layer to either compute a function of two inputs, or select any of 64 single-input functions on a per-subword basis.

This ALU has 10.5 Mibit of SRAM in 19 packages; Table 2 lists their individual sizes. As these byte-wide RAMs are not sold today, actual device widths will be 16

12 The cycle time penalty is estimated using five total datapath layers for the 2-layer ALU, 10 ns SRAM access time, 2 ns per layer delay for board capacitance, and a 3 ns fan-out penalty for the 2-layer circuit. The 3-layer ALU adds a net 9 ns to the datapath.

13 Section 4.11 accounts for the three control bits not shown.

bits, and total physical storage will be 21 Mibit. It’s possible to “fold” the 64k × 16 size into 128k × 8 units at the cost of some capacitive load and added inverters, thereby removing dependency on having the 128k size available for two of the RAMs. There is no advantage folding 32k × 16 into 64k × 8, as 32k is scarce and more expensive. The total cost for all the RAM is around US $30 in 2020.

Our term for a 6-bit subword is *tribble*, as it’s a multiple of three bits. This is analogous to a byte in power-of-two word size architectures. Unfortunately, tribbles are not wide enough to encode even US ASCII, which requires only seven bits; however, Unicode has greatly eroded the usefulness of bytes for symbol representation anyway.

Tables 3 and 4 show input and output bit assignments for all of the RAMs, and appear mainly as assurance that enough wires exist on the packages. They don’t require study for one to understand the ALU in general terms.

Table 5 summarizes opcodes this ALU implements by category. Many opcodes don’t appear in this list, because their naming semantics for the assembler have yet to be decided.[14] Tables 6–13 show the mechanisms that implement the named opcodes. The general format first lists the assembler mnemonic on the left. A given mnemonic can generate several different opcodes, depending on the signedness of its arguments and results. For example, the mnemonic A (add) in Table 6 includes overflow checking that requires awareness of the signedness of both arguments and the result, and this awareness is supplied by using eight different opcodes for A. This multiplicity is indicated in the second column. The next four columns indicate the function select inputs for the three ALU layers and carry decision RAM. To keep the tables compact and legible, these SRAM operations have single-character names, and are described in further detail in Tables 14–19. The next column shows what must be supplied as the right operand; by default this column simply reads “R”, meaning the natural right argument of the intended operation is supplied.

14 For example, there is no mnemonic yet for reversing the bits of a word, although the capability is present via a stacked unary operation.

4.2 Additive opcodes

Table 6 shows the basic addition and subtraction opcodes. The α and γ layers provide carry-skip addition, with β not participating. An extra subtract operation is offered with the arguments reversed, a special provision for CPUs with fast hardware multipliers.[15] Subject to whether or not ”with carry“ is specified in the opcode, θ distributes correct carry decisions to γ. The INC, DEC, and NEG assembler mnemonics don’t correspond to actual opcodes, but conveniently provide right arguments for add, subtract, and reverse subtract. This is why these mnemonics show zero for their number of opcodes.

Addition and subtraction are stratified by signedness, causing six mnemonics to assemble to 48 distinct opcodes, depending on the signedness of the arguments and result. This permits not only mixed-sign arithmetic; e.g., add a signed register to an unsigned register, but also alerts the CPU as to the signedness of the destination register so that overflow is detected correctly. Section 4.11 offers more specifics as to how these checks work.

4.3 Bitwise boolean opcodes

There exist sixteen bitwise boolean functions with two inputs, and as Table 7 shows, this ALU implements all of them. Interestingly, the individual layers only implement a handful of these. α supplies the common AND and OR operations. β is very limited in its flexibility to assist, because it’s operating on a transposed word. The only change it can safely make is to invert all bits, thereby extending what α can do to include NL, NAND, and NOR. γ doesn’t have enough space to offer many operations, because its carry input cuts in half the available space for their implementation. But by XORing what arrives from β with the original right operand, γ adds another five operations. γ’s multiplex and kill operations manage to fill out the remaining constants and identities.

15 The ALU’s arguments come from two copies of the register file which ordinarily have the same contents. The 72-bit result from multiplication gets split between these copies, causing that register’s value to depend on whether it is the left or right operand of a subsequent subtraction.

Although the opcodes of Table 7 technically implement the boolean functions listed, not all do so semantically. For instance, the boolean function NR ("not R," or bitwise invert) function theoretically ignores its left input, but this ALU cannot both ignore its left input and invert its right input due to SRAM space limits. Instead, the assembler has to substitute all ones in place of the left input register specified in the source code. This doesn't present a problem for users, compilers, or assembly language programmers, but it compromises style a tiny bit.

4.4 Compare opcodes

In Table 8, we find opcodes for compare and shift tasks. The maximum and minimum operations are unusual in CPUs: traditionally a compare and branch is used, but we accomplish these in one cycle. Not only this, but the two operands and results can be of any signedness. MAX and MIN are implemented by having α and β pass the left argument unchanged, and γ select between the output of β and the right operand depending on whether the carry decision bits are all ones or all zeros. θ is able to compute which argument is larger on the basis of tribble comparisons made by the α RAMs and delivered via the propagate and carry wires.

The BOUND operation is intended for array subscripts. Although BOUND's implementation is the same as for ordinary subtraction, separate opcodes are counted for BOUND so that the link editor can locate and optionally remove array boundary checking. Additionally, BOUND does not write any result to a register. The index tested is supplied in the *right* operand, which may be unsigned or signed but is evaluated as if unsigned. The left operand is the bound. Hypothetically the bound is always at least zero, but signed bounds are permitted to be passed (thus two opcodes) in order to accept whatever data type the register is declared to be. The operation performed is simply subtraction: the 2^{-36} bit will be set if the index is less than zero or at least as large as the bound. The logic that monitors this bit and interrupts the CPU is external to the ALU. BOUND will not work correctly for bounds of 2^{35} or higher, because indices that high will be interpreted as less than zero and therefore not within bounds. But there won't be enough memory for this problem to appear.[16] BOUND will never set the overrange flag; instead, the CPU is interrupted.

The CMP instruction does a simple comparison and is identical to subtraction, except the control unit skips the step where the result is written to a register. It likewise does not check to see if the difference fits in any particular format, and will never set the overrange flag.

4.5 Shift and rotate opcodes

The shift operations in Table 9 have unconventional definitions. *Arithmetic shift* means multiplication or division by a power of two, rounding towards negative infinity in the case of division. In traditional forums, arithmetic shift means that a signed number is being shifted. But it's moot whether the number is unsigned or unsigned; what is important is whether the intent is to perform arithmetic, and if this is the intent, the result must be tested for overrange irrespective of signedness. A shift in the absence of any intent to multiply or divide is called in this proposal a *logical shift*. Logical shifts are not tested for overflow, because the bits involved do not represent quantities.

The logarithmic shifter is implemented by the β and γ layers. The shift and rotate operations require the number of bit positions to be encoded into every tribble of the right argument. The notation {R} means to suggest a vector with all subwords having the same value. These identical copies can be provided by the compiler or assembler if the shift amount is fixed; otherwise a stacked unary operation (Table 22) is used to replicate the correct value into each tribble.

The γ layer's participation in the logarithmic shifter is expressed in terms of leftward rotation only, because RAM in that layer is very scarce. This means that all shift and rotation operations are normalized to a left perspective, so to shift right one position, the argument supplied must indicate a left rotation of 35 positions. The stacked unary operations that prepare shift and rotate arguments account for this.

16 Someone who can afford a 2^{36}-word SRAM data memory won't have a problem upgrading to a 64-bit, 3-layer ALU using $2^{21} \times 16$ SRAMs, whereupon BOUND will give correct results up to $2^{63} - 1$.

There are four sub-cases (shift vs. rotate, 0−35 bits vs. 36+ bits) not elaborated on here. Rotations in excess of 63 bits require extra instructions for the modulo-36 division needed.

Negative shift amounts aren't directly supported by this ALU, because a signed tribble can only represent numbers in the range −32 … +31, while the range of shifting is at least −35 … +35. If a variable shift amount might have either sign, a branch is required to straighten out matters.

Range checking for left arithmetic shifts is done in the α layer, which encodes the signed and unsigned overflow result for each tribble onto the propagate and carry wires. Two of θ's output bits are the logical OR of its propagate and carry inputs; these outputs return to the control unit so that overrange can be flagged.

4.6 Multiply opcodes

Although this ALU isn't specified with a hardware multiplier, it has several opcodes that significantly accelerate unsigned multiplication in software. Table 10 shows the implementation for these. For short multiplication, the right operand must be a factor in the range of 0 to 63, and replicated across all six tribbles. ML and MH multiply across the tribbles of the left and right operands. MH throws in a left shift to align with ML, so the results can be directly added for the final product. So if a, b, c, and t are unsigned registers, $0 \leq b \leq 63$, and $a \times b < 2^{36}$, we compute $c = a \times b$ in four instructions:

```
t = b copy 0   ; replicate tribble 0 of b across t
c = a mh t     ; short multiply high tribble
t = a ml t     ; short multiply low tribble; replaces t
c = c add t    ; final product
```

Short multiplication when b is constant only needs three instructions, because the compiler or assembler will provide the initial replication.

Short multiplication correctly detects the overrange condition, in which case the result is of little value and might best be considered undefined. Section 4.10

gives more particulars.

Long multiplication can be emulated in software. For 36-bit unsigned integers with 72-bit results, 53 cycles are needed. Except that we might not have CPU microcode that dispatches these instructions transparently, this performance is not far out of line from the Intel 80486, which required 18 cycles even to multiply by zero, and up to 42 cycles for larger numbers.

Assembly code showing long multiplication appears in Listing 1. Two opcodes not used for short multiplication include MHNS (“multiply high no shift”), which multiplies tribbles pairwise but without shifting the result 6 bits to the left, and DSL (“double shift left”), a which shifts the high tribble of the right operand into the low tribble of the left operand. We also see COPY, which selects a tribble from the left operand and replicates it six times.

The COPY opcode is actually a general call to any of the swizzles of Table 25, where the first six operations happen to copy the six tribbles by position. So COPY with a right operand outside the range of 0–5 probably isn’t a copy at all. This opcode may later be known by some other name, but is presently named for clarity as to how it is used in this proposal.

4.7 Bit scan opcodes

Table 11 shows implementations for some scan and mix opcodes. LEAD and TRAIL can be used for operations such as count leading ones. Here’s how they work. Suppose we wish to count the leading ones on the word

```
111111 111111 111100 011100 111111 111111
```

and a unary function is available for α or γ that counts leading ones on a per-tribble basis. That function returns the tribbles [6 6 4 0 6 6]. The LEAD operation scans tribbles from left to right, and clears every tribble after the first that is less than 6. The expectation is that α’s count of leading ones starts from the left and continues to be valid for as long as no zeros are encountered. Once α sees a zero it can finish that tribble’s count, but further results to the right from α do not count. So LEAD replaces

[6 6 4 0 6 6] with [6 6 4 0 0 0]. These six tribbles are then added together using the most efficient operations on hand to yield the full count, which is 16 leading ones.

LEAD and TRAIL work because α has an operation that tests the tribbles of L and R pairwise to see if the left tribble is lesser. θ receives those results via the carry wires, and has left and right scan functions to turn on the carry decision bits for tribbles left or right of the point where the bit count fell short. After that, γ has a function that zeros any tribbles where the carry decision input is high.

Efficiently adding the tribbles of a word together is a challenge, and involves simultaneously applying various unary functions to different layers. This need occurs not just for leading and trailing counts, but also determining the Hamming weight of a word, sometimes called population count or simply popcount, which is a count of the number of set bits in a word. For example, we might begin with

```
100000 011000 001110 110011 111110 111111
```

and use one of α's unary operations to count ones, returning [1 2 3 4 5 6].

But now we have to add the tribbles. For this purpose α, β, and γ offer 64 sets of "stacked unary" functions, listed in Table 22, that are selected by the right argument. One such set implements "add six sets of three bits to yield three sets of four bits," meaning a bit count of [1 2 3 4 5 6] can be reduced in one cycle to the sum [0 0 0 5 7 9]. How might this work? Two three-bit addends can fit in a single tribble. So α's lookup table could shift the three leftmost tribbles three bits left. β could move the three most significant bits of the three leftmost tribbles into the same positions in the three tribbles on the right. γ could then look up the 4-bit sum for the packed 3-bit addends on the three rightmost tribbles. By a similar process, the next CPU instruction can reduce [0 0 0 5 7 9] to [0 0 0 0 11 10], by splitting the four bits of the first addend into the two unused bits of the two other tribbles. But as things stand right now, the last addition will require two CPU instructions to shift the 11 addend right and complete the addition.

Thus we can obtain Hamming weight with no loops in five cycles: count ones, reduce to three addends, reduce to two addends, shift, and finally add. Counting trailing zeros uses six cycles: count trailing zeros for each tribble, TRAIL, reduce to three addends, reduce to two addends, shift, and add.

There is a faster way to count leading or trailing bits in just four cycles by avoiding the summation of six terms. It uses θ between α and γ to convert the word input to a word with at most one bit set—the bit where the trail ends—in the first cycle. Cycles two and three use β to locate that bit's tribble and offset within the tribble, and the fourth cycle adds the coarse and fine bit positions to give the final result. There are a few subtleties and tricks, and the method doesn't work for finding Hamming weights. Whether this approach matures into assigned opcodes, like the fate of LEAD and TRAIL, is subject to available space and depend on the degree to which other unary and stacked unary functions need to be included.

The NUDGE opcode adjusts the rightmost portion of a word, which could be a pointer, to conform to a supplied template. This template consists of a start bit, indicating the number of bits to replace, and the replacement bits themselves. For example, 0111010 NUDGE 0010110 replaces the rightmost four bits to give the result 0110110 in one instruction. Without NUDGE, this process would require two instructions and would happen as (0111010 AND 0001111) OR 0000110.

NUDGE is useful in part because many languages align memory that is allocated to a small power of two. Reportedly, GNU malloc always returns a pointer aligned to a multiple of eight, or on 64-bit machines, sixteen. To the extent a developer is informed of the alignment and can rule out unexpected change, fast access to various members of a structure can be available from close-enough pointers. As an example, a node for a binary tree might have this form in C:

```
struct j {
    struct j *left;
    struct j *right;
    struct j *parent;
    ...
};
```

If we wish to disconnect some node n from its parent, we need to dereference the parent member, then examine the left and right fields on the parent to see which matches n, and set the appropriate one to NULL. That's a lot of work. If struct j is aligned to a multiple of four and NUDGE is available, there is a faster arrangement:

```
struct j {
    struct j *left;
    struct j *right;
    struct j **parent;
    ...
};
```

With this format, we can NULL the dereferenced word within the parent without investigating which child n used to be. The difficulty is that the parent member is only useful for deletions, because it doesn't know where within the parent node it's pointing. The NUDGE opcode eliminates this problem. If this C program can access the NUDGE opcode, we can in one cycle locate any member of the parent by nudging the parent pointer's alignment within its known power of two. More generally, we can compute the address of a structure member in one cycle from the address of another member—even if we haven't kept track of which member the address calculation will start from.

NUDGE isn't new, in the sense the same operation can already be done at half the speed by using AND and OR. Moreover, the alignment behavior an implementer chooses for memory allocation on an SRAM CPU may differ from conventional processors. GNU malloc's alignments to multiples of 8 or 16 are in fact just multiples of two machine words, because many traditional CPUs address individual bytes. But byte addressing and need for aligned memory boundaries might not apply to an SRAM-based CPU. Similarly, the cache lines we ordinarily try to consider may not be present on SRAM machines. So NUDGE might not become a big trend, but it does serve as an example of overloading a carry decision mechanism for other uses. It's also a rare example of left-to-right carry propagation.

4.8 Bit permute opcodes

Table 12 shows how bit permutation opcodes are provided in the α and β layers. Most software doesn't need to permute bits, except for ordinary shifts and occasional endianness changes, and only newer instruction sets offer much beyond these. So when an arbitrary rearrangement of a word is needed, historically a lot of code and clock cycles are expended doing this rearrangement. A swap of even two bits requires six instructions (isolate bits, mask out bits, shift left, shift right, or, or), and an arbitrary permutation of a 32-bit word will take 32 repetitions of (shift, mask, or) for 96 cycles.

SRAM ALUs offer fast permutations, although how they are specified has to respect subword boundaries. Thus Intel's PDEP and PEXT instructions for parallel bit deposit and extract, available since 2013, can't be mapped onto the architecture of our research. Instead, we offer one instruction for permutations inside tribble boundaries, PIT, and another which permutes across these boundaries, PAT.

The more bits we need to move, the faster our ALU is compared to not having bit permutation support. If we need to swap two randomly chosen bits, the likelihood they are in the same subword or same position within a subword is about 30%. This lucky case requires just one instruction. The rest of the time, three instructions are needed. But if we need a random permutation of a 36-bit word, the kind that might need 108 instructions had we not planed for this need, we can "always" do this in five instructions. This is a big leap: it takes an average of 2.39 instructions to swap two bits, but we only need five instructions to arbitrarily arrange 36 bits.

At present, I offer a hierarchy of claims about how many instructions are needed for 36-bit permutations. The operands to the instructions that produce these permutations are tedious and ordinarily not suitable for manual computation. Instead, an algorithm converts the desired permutation into the sequence of instructions which accomplishes it. I can offer you algorithms that succeed in

- 6 instructions, with proof, or
- 5 instructions, tested on 100 million random permutations, or
- 4 instructions, might fail, and takes quadrillions of CPU cycles to find out.

PIT is implemented in the γ layer. PAT uses the β layer and is simply a transposition of PIT. It's helpful to regard a 36-bit word as not a row of bits, but as a 6×6 matrix. Any corner may be used as the origin; the point is that each row of the matrix represents one tribble of the machine word. Plate 19 shows how such a matrix can be operated on. The matrix is the left argument to PIT, which operates on the rows of the matrix, or PAT, which operates on the columns. The operation wanted for each row or each column is taken from the tribbles of the right argument, meaning we can invoke six different permutations on the six rows in one PIT, or six different permutations on the six columns in one PAT. So the good news is that a lot of things can happen at once simultaneously. The first part of the bad news is we can only move within rows or within columns during one instruction.

The second part of the bad news is that we aren't afforded all the possible permutations within the rows or columns in one instruction. It takes two instructions to fully manipulate the rows or columns, because the tribbles in the right argument can only distinguish between 64 operations, and there are $6! = 720$ permutations possible for any tribble. By composing two PITs or two PATs, and selecting twice from the 64 permutations we do offer, we can reach any of the 720 possible permutations in all of the rows or columns.

Plate 19 shows how six bit permutation instructions combine to produce arbitrary permutations of full words. The three steps are indicated by horizontal arrows (two PITs), and vertical arrows (two PATs), with the colored matrices showing the states before and after each step. A randomly-selected permutation appears in the upper left "start" matrix. The numbers in each box show that box's eventual destination as its content—a zero or one—moves through each step. Every six consecutive numbers is given a different background color, allowing visual differentiation of the subword each bit will end within.

In the first transition, drawn as four arrows, each row is rearranged such that no color appears in the same column twice, or equivalently, every column contains every color. This guarantees that the second transition, drawn with vertical arrows, will be able to move every bit to its destination row (tribble) without conflict. The algorithm for the first transition works one row at a time without backtracking. The first row never requires adjustment. The second row of this example had no color conflict with the first row. The third row contained three color conflicts, requiring relocation of the 16, 19, and 24 bits. The relocation algorithm is simply to try all 720 permutations, and use the first permutation that does not conflict with any preceding row. Eventually all six rows are decided, and the six permutations chosen are encoded into two PIT instructions.

A question appears as to how we know, for each tribble, which two permutations selected from the 64 implemented are needed for any of the 720 possible permutations. We know either by trial and error, with a maximum of 4096 trials for each tribble, or from a 720 × 12-bit table in the compiler or assembler.

The second transition is represented with six vertical arrows, because every column requires some adjustment for its bits to arrive in the correct rows. The permutations needed for each column are found by iterating through all 720: one and only one permutation will produce the correct result for each column. Two PAT instructions cause the desired migration.

The third transition, drawn with six leftwards arrows, uses two last PIT instructions to finish the word permutation. Here again, each tribble has one and only one correct permutation of the 720, although there will often be more than one composition of two PIT operands that can produce it.

So there's how arbitrary 36-bit permutations work in six instructions. What about five instructions? The second and third transitions don't offer any budget we can cut, because we need all 720 permutations available for each subword—two PATs and two PITs. The odds of being able to cover the second transition with a single instruction appear to be $(720 \div 64)^6$ against us, over two million to one. The third transition has the same prospects. But the first transition is flexible: any set of row permutations that result in no color conflicts within columns is acceptable. There are many such sets, so we can try the first transition experimentally on a whole bunch of random 36-bit permutations, and see if we can succeed with just the 64 permutations the ALU has on hand. Of 100 million cases I tried, all 100 million succeeded. So not only do we have a five-instruction method, but it even has a simpler algorithm to determine the instructions needed.

Now we consider the existence of a four-instruction, general permutation for 36 bits. Observe that two consecutive PIT instructions can only yield 720^6 possible outcomes, as there are only 720 orders six objects can be arranged in. But a PIT followed by PAT or vice versa does not encounter this bound; each of these has 64^6 possibilities, and their combination is 64^{12}. So if a four-instruction permutation sequence contains PITs next to PITs or PATs next to PATs, the total number of outcomes is at most $3 \times 720^6 \times 64^{12} \cong 2 \times 10^{39}$. This isn't enough to offer the $36! \cong 4 \times 10^{41}$ permutations of 36 bits, so the odds of making this work with immediately repeated PITs or PATs are 188 to 1 against.

If we can find 4-instruction random permutations of 36 bits, we must strictly alternate PITs with PATs, where we are bounded by $2 \times 64^{24} \cong 4 \times 10^{43}$. This is sufficiently above the number of 36-object permutations to offer, on average, 120 different ways of finding each. But this does not guarantee that every permutation is reachable, let alone offer a fast algorithm for producing the four instructions.

We do have *an* algorithm to try to find four instructions. It's applied twice for completeness: PIT PAT PIT PAT and PAT PIT PAT PIT in either order. There is one transition for each instruction, so if Plate 19 were redrawn for this operation, five matrices would be shown. The first transition is carte blanche: arbitrarily select any of the $64^6 = 2^{36}$ PIT or PAT arguments. In practice, we would loop through all of them until the algorithm succeeds. The second, third, and fourth transitions are identical to the three transitions on Plate 19, except we strictly limit to 64 alternatives per tribble at each step. Time complexity goes like this: the first transition always succeeds for its 2^{36} variants, and the second transition succeeds an average of 820 times for each of the first transitions. This estimate comes from simulation with 10 000 trials. So this algorithm enters the third transition a total of $2 \times 2^{36} \times 820$ times, which is good because the combined odds of success in the third and fourth transitions are presumably $(720 \div 64)^{12}$ to one against us, less than one in four trillion. So if we run the entire experiment to completion for many random permutations, on average we would expect to find around 27 sequences of four instructions that can produce each permutation.[17] This isn't a guarantee there aren't permutations which no series of four instructions can reach, but the chances of succeeding often at least appear pretty good. But the problem is CPU time: if you tell an assembler or compiler "I want the permutation _________, figure it out and do it in four cycles," it has to come out of the second transition successfully to try the third and fourth about four trillion times. This computation would take decades of CPU time on average to find the first working sequence of four instructions. Although the search is easily parallelized, we will ordinarily settle for five instructions.

Implementing our bit permute instructions requires selecting 64 permutations of six objects for the ALU to support out of the 720 possible. There are three criteria we want this subset of 64 to satisfy:

1. All 720 permutations must be available by composing at most two from the chosen subset.
2. All permutations of 36 bits must be available in five instructions via the chosen subset. This will be true if the first transition always succeeds in one instruction.
3. The subset's permutations should not be too "esoteric," to give the best chance of accomplishing simple permutations in just one instruction. For instance, the identity permutation should be among the 64 chosen.

The permutations chosen for this ALU appear in Table 23. From the above list, only criteria 1 and 3 were used when these permutations were selected. The middle requirement was left to chance, although there is more to say about that shortly. The order of items in the table is roughly the order I dreamed them up. Bit ordering in this table is not the conventional 543210 that would be used for arithmetic, but 012345 in order to make these permutations more spatially understandable by programmers. So the identity permutation is 012345, and exactly reversing the bits is

17 This expected value of 27, which relies on multiple assumptions of uniformity, is within an order of magnitude of the preceding paragraph's expected value of 120, which is more dependable.

543210. Table 23 was produced iteratively: I would think of permutations to add to the set, and then I would use a greedy algorithm to add as few permutations as it could to make the set “720 complete.” The minimum size needed for this completeness grew slowly. The more permutations I added to the subset, the fewer the algorithm needed to add to reach 720 within two instructions. So I might find I had 20 left to choose, and after choosing 10, I’d find there were still 15 left to choose. In the end I was able to choose 58 of the permutations by the seat of my pants, and the algorithm only needed to choose six. But I couldn’t have chosen these last six without automation.

If someone had to design an unusually crowded ALU, possibly using the same function select inputs for permutations and swizzles, perhaps there wouldn’t be room to include 64 permutations. How small a subset would still work? As $26 + 26^2 = 702$, it’s clear that a minimal set will have at least 27 elements. I haven’t succeeded at computing such a set, although I spent more than 18 months of CPU time trying. But there wasn’t a lot of human effort expended. The algorithm I wrote had a fast implementation in C, but a better-considered algorithm might have a faster implementation.

During the search, I found four subsets of 30 permutations that can reach all 720 within two instructions. The first of these appeared in just hours, before I was even looking at the program output consistently before clearing the screen. I was stunned when I saw it, because I knew by that time that a set that small is very difficult to compute, and the search of that time ran 275 times slower than it would after I ported it from Python to C. Table 24 shows this set of 30 permutations.

Importantly, the subset of 30 permutations cannot always implement whole-word permutations in five instructions. Sometimes there isn’t a one-instruction solution for the first transition: in 100 million trials, nearly 1.80% of the cases did not succeed. When this happens, the next thing to try is swapping PIT and PAT, which will usually resolve the issue. So the set of 30 works in five instructions most of the time, but needs six instructions about one time out of every 3000 permutations needed. This very high success rate with just 30 permutations in the subset gives me a lot of confidence that the much larger set of 64 permutations, having exponentially larger opportunities to find solutions, will always be able to permute 36-bit words in five instructions. Certainly in the absence of proof, compilers and assemblers need a tested fallback to generate six instructions for word permutations. This fallback is trivially implemented by ordering the 64-subset with the identity first, only writing code for for the 720-permutation searches (not bothering to try 64 first), and suppressing output of identity permutations. This scheme has the added benefit of exploiting any one-instruction or identity transitions when only a few bits are being moved.

4.9 Mix opcodes

Opcodes for mixing are stated in Table 13. MIX maximizes the ALU's capacity to use S-boxes quickly for hashing and pseudorandom number generation. This ALU contains 64 × 18 = 1,152 different S-boxes; that is, 64 S-boxes for each RAM in the α, β, and γ layers. They are applied to the left operand by the MIX instruction, and selected by the right operand. This implements a key-dependent mixing operation for 36 bits in one machine cycle. The mixing needs to be iterated for a number of cycles with different key material supplied as the right argument. Every bit of the 36-bit output from MIX depends on every input bit. If exactly one input bit changes, the expected number of output bits that change has not been computed, as this quantity is easier to determine by simulation than by messy combinatorics and probability theory. As a simulator will be forthcoming, I'm inclined to wait for it. A simulation also is less likely to err.

Another estimate to be obtained from simulations is how many rounds are desirable for various applications. Possible measures include using Brown's Dieharder test suite for random number generators [19], as well as using self-organizing maps to see how long visible features can be distinguished in the MIX output.

The XIM opcode reverses the MIX function. Words of the key must be provided correctly and in reverse order to recover the original input. The S-boxes for XIM derive easily by inverting the MIX S-boxes, and swapping between the α and γ layers.

MIX and XIM have potential for use in cryptography, provided that experts specify and validate the ciphers. This is a specialization I am not qualified in, but here are a couple of problems with these operations. First, I have specified S-blocks that are not optimized in any manner against differential cryptanalysis. This is somewhat on purpose: optimizing for a specific attack could break a cryptosystem with respect to some other, unanticipated attack. Most efforts to design attacks against block ciphers and S-blocks are done in secret. So although the S-blocks specified are potentially terrible with respect to differential cryptanalysis, I can't charge forward as if differential cryptanalysis is our only concern. My approach here is not shared by contemporary standardization processes for ciphers: the safeguards I neglect are considered mandatory for new cipher submissions to standards bodies. I'm all right with this, and I won't be submitting a new cipher. And as differential cryptanalysis attacks go, many that show "breaks" in the literature are not practical against deployed systems. The scope of these and related considerations is large and outside of our proposed research.

The other problem with MIX and XIM for secrecy is the 36 bit block size is too small and cannot be enlarged easily. The key would need to be changed with just about every packet sent or received due to the potential for birthday attacks, as collisions in real-world protocols have been demonstrated for considerably larger block ciphers [20]. Although special cipher modes have been designed to expand the birthday bound [21], 36 bits is still too small a block for these new modes to be of much help. But implementing a cipher for larger blocks requires that we choose between either many more instructions per word encrypted, or a much bigger circuit board.

The block size threat, infelicities of S-box selection, and other problems to be identified do not preclude efficient use of MIX and XIM for secure communication and authentication. But they do put the problem above my pay grade, so I have to recuse myself with respect to cryptography.

Listing 2 shows a Python 3 specification for the MIX opcode's suggested S-boxes. The boxes are filled from a so-called "nothing up my sleeve number," which presumably I haven't enough control over to build a backdoor into. I opted to use an easy-to-compute irrational number, rather than have to store a long number, or make people go find it online or compute it with some utility, and then still have to parse it.

The square root of one half ≅ 0.7071 is possibly the easiest irrational of all to write code to compute; also it converges *really* fast using Newton's method. Listing 2 does this using Python's built-in arithmetic for large integers. We compute the first 341,000 bits of our irrational in a few moments, and then start multiplying by the number of choices remaining in whichever S-box we're working on. The bits to the left of the implied decimal point become our "pseudorandom" number, and the bits to the right become the remaining irrational number for subsequent extractions.

For the benefit of people testing alternate implementations: the final S-box computed is for the leftmost γ tribble with right operand 63, and is [2 52 22 7 17 63 12 56 38 48 44 4 29 10 30 8 21 54 28 25 62 15 49 32 11 18 42 43 36 19 24 61 3 35 6 39 16 31 53 40 46 34 27 9 58 5 37 20 50 14 57 33 59 51 55 1 0 45 47 41 13 60 23 26]. The output is the same if more than 341,000 bits of the irrational are computed, but if the length is reduced below 340,987 bits, the output destabilizes due to premature exhaustion of numeric precision.

4.10 α layer operation

Having described the opcodes to some extent, here are some specifics as to the SRAM layers which do the computations. Table 14 shows the overall function of the α layer, particularly the function of the four function select bits into $α_0$–$α_4$. As we'll see in a moment, $α_5$ works in the essentially same manner with one alteration.

Most commonly, α provides bitwise AND and OR, and the first layer of a carry-skip adder for addition, subtraction, and reverse subtraction. These are the slots named α.&, α.¦, α.+, α.−, and α.m. For the three additive operations, appropriate propagate and carry signals are generated.

There are two identity operations in this layer. The first is α.=, which also examines the left and right operands and sets the propagate and carry outputs to indicate which is larger for each tribble, or if equality holds. The other identity is α.o, which also determines if a left arithmetic shift would overflow on the basis on the bits each tribble contains. For α.o, the propagate and carry bits are coded to indicate whether signed and/or unsigned overflow will occur during the shift.

The α.s and α.S operations provide S-box mixing. The left operand is the input word, and the right operand contains the key material (which S-box to apply) for each tribble. The forward mix is α.s and has inverse γ.S; α.S is the inverse of γ.s.

Up to 64 unary operations, selected by the right operand, are provided by α.u. These operations can be mixed and matched across tribbles, so up to six different unary functions can be applied simultaneously during one instruction. Table 21 lists some anticipated operations which, depending on the ALU function select lines, might or might not be used in combination with the other layers. Some might be given assembler mnemonics in the future. Right now, none use the propagate or carry outputs.

Because α.u requires a right operand, which is copied to layers β and γ, we cannot have those layers implement a different unary operation during the same instruction. But there are a few combinations of operations between layers that would be frequently used, although for each combination the right operand must be held constant. This is the purpose of the "stacked unary" operations, implemented via α.w, which are assigned right operands coordinated across all three layers. Table 22 shows a list of currently known stacked unary operations. Unlike α.u, α.w has control over the propagate and carry signals and can make use of them.

The NUDGE operation from Section 4.7 begins with α.n, which presumes that for every tribble, the leftmost set bit of the right operand is the start bit. That tribble's output is adjusted according to the NUDGE specification, and its carry output is set. θ and γ will assure that only the actual start bit is treated as such; all tribbles right of the start bit will be replaced with the right operand during γ.

Tables 14 and 15 also summarizes the propagate and carry outputs for α’s operations. Table 16 explains these outputs in finer detail.

Signed right arithmetic shifts must replicate up to 35 copies of the 2^{-35} bit, ordinarily misnamed the “sign bit.” β can handle most of these replications, but the transposition between α and β only distributes the 2^{-35} bit to the β_5 RAM. The α.r operation solves this by handling the sign extension for the leftmost tribble prior to transposition.

α does most of the work for short multiplication. By “short,” we mean the product fits in one word, both factors are unsigned, and the right factor fits in one tribble (is less than 64). These conditions support a fast multiplication operation with no loops. When tribbles are multiplied pairwise between the left and right arguments, the result of each multiplication is 12 bits instead of 6, so there are separate operations, α.x and α.X, to fetch the low and high tribbles of the results. The high bits must be shifted one tribble left, with overflow checking, to align their place values correctly. This is done during the same instruction as the tribble high multiplications, with α providing the range check and β providing the 6-bit left rotation. So short multiplication by a variable amount is done in four instructions: COPY, ML, MH, A(dd). Short multiplication by a tribble constant uses three instructions: ML, MH, A.

Because β rotates instead of shifts for the MH opcode, overflow won’t just wrap modulo 2^{36}. Rather, the high word of the result will be added to the low word. Either scenario would be given an incorrect result, but this one avoids the need to allocate a slot in β for an express left shift. In any event, the overflow flag is set correctly and can be checked at a convenient time.

The α.* operation overwrites the most significant tribble of L with its counterpart in R. This is half the work required by the DSL opcode, which has β rotate the outcome six bits leftwards. The effect is that L is shifted one tribble left, with its new low tribble shifted out of the high tribble of R. What’s that spell? Unsigned long multiplication with a 72-bit result in 53 instructions. Listing 1 shows the full assembler code.

Layer α is now full except for unallocated unary operations. Expanding α for binary operations is costly: the next operation would add 7 Mibit to the ALU. We've accomplished a lot using the small RAMs this layer contains.

4.11 α layer operation, leftmost tribble

The α_5 RAM is charged with extending words that are to be added or subtracted to 37 bits, so that full-time detection of wraparound and capacity to do mixed-sign arithmetic both happen. As Tables 4 and 3 indicate, this output connects directly to γ_5 and bypasses the β layer. To extend the sign correctly, as well as to compare magnitudes correctly, α_5 has to know the signedness of both operands. This requires that the =, +, −, and m operations of α_0–α_4 be stratified by signedness into 16 different operations for α_5. These additional operations appear in Table 15.

The differentiated operation of α_5 is not compatible with the control inputs for the rest of layer α, so the control unit must provide additional bits. The slots are ordered in a way that lets α_5 reuse the two least significant control lines of α_0–α_4. No further reuse is possible, so the control unit (a circuit outside the ALU) extends three additional bits to α_5 to select among its 32 operations.

Final overflow determination is not done by α, but by the control unit using output from θ and γ. Table 20 shows how wraparound is ultimately caught as a result of α_5's sign extension.

4.12 β layer operation

Table 17 shows the operations for β. In this table, α and α^T refer to the output of the α layer immediately prior to and just after transposition, respectively. The identity is β.= and simply keeps out of the way. The five operations β.ℓ, β.L, β.R, β.{, and β.} form the first layer of the logarithmic shifter, providing left rotation (β .ℓ), logical and unsigned arithmetic shift (β .L, β.R) , and signed arithmetic shift (β.{, β.}). Note for the signed shifts, α had to sign extend the leftmost tribble in order to get the sign to all places needed in β. The second layer of the logarithmic shifter consists only of γ.ℓ.

S-boxes are provided by β.s, and the inverses of these same S-boxes are in β.S. Stacked unary operations (see also Table 22) are β.w. There are no “regular” unary operations so-named, as these turn out to be swizzle operations. They’re accessible at β.z and described in Table 25, which still has a lot of space to be filled in.

β.! and β.^ do bitwise negation and exclusive OR. Negation sees heavy use by this layer, as the other two layers don’t offer it. Negation and identity are the only two β operations where this layer’s transposition is irrelevant. XOR via this layer is interesting and would probably see rare use. One use might be with a zero left argument, causing the output to be the transpose of the right argument.

β offers a 6-bit left rotation in the middle layer irrespective of the right argument. This operation is β.*, and is used by the “multiply high” MH opcode as well as the “double shift left” DSL opcode for short and long multiplication, respectively. This is really an ordinary swizzle we would rather use β.z to implement, but we have no mechanism to simultaneously deliver different right operands to α and β.

Lastly, permutations on the transposed tribbles at the β layer are provided by β.p and used by the PAT opcode, as explained in Section 4.8.

β has unallocated space for two more operations. Dissertation research may uncover noble uses for this space.

4.13 γ layer operation

The γ layer is the last row of the carry-skip adder and also implements other functions listed in Table 18. In this table, β refers to the output of the β layer after transposition back to ordinary word configuration. This table differs from α and β in that only eight slot numbers are available for operations, as the need for a carry input reduces the γ RAMs’ address space by one bit. But as only four operations actually require a carry input, the four remaining slots can use the carry wire as an additional select bit. Recall that the carry decision input does not come from the CPU’s carry flag, but from θ, which offers full control over the carry decision signal.

So as of this writing, γ can do as many as twelve distinct operations.

If there is a “main” operation for this layer, it’s γ.+, which increments any tribble (with wrapping) that has its carry decision bit set. This is used for addition, subtraction, and cases where γ makes no changes to the word. For these identity operations, θ is programmed to zero all of the carry decisions.

There are two multiplex operations for this layer. γ.? outputs the right operand of tribbles with their carry decisions set, and the left operand of the remaining. Its main use is the MAX and MIN opcodes, where two operands are supplied, compared, and one is selected for output. γ.k supplies the same operation, except that zeros are output in lieu of the right argument, thereby freeing the right argument for other uses. This permits LEAD and TRAIL to zero out tribbles that appear left or right of a decision point.

γ.w has its right argument coordinated with α.w and β.w to supply stacked unary operations on words. There is great flexibility with these 64 operations, except that none have a right argument available for use. Table 22 lists the stacked unary operations that have been assigned to date.

γ.s provides the $6 \times 64 = 384$ forward S-boxes for this layer. The inverse of α’s S-boxes are available in γ.S. More detail appeared in Section 4.9.

γ.u provides unary functions that can be fully contained in γ and don’t require any memory from the other SRAM layers. These functions are also available in α and are listed in Table 21.

Of this ALU’s two exclusive-OR processes, γ.^ is the most canonical. It’s used in 7 out of 16 bitwise boolean opcodes, making it the most-often specified non-identity logical operation. XOR with one input transposed is available in β, from which ordinary XOR may be composed in two cycles with a choice of transposed or non-transposed output.

γ.ℓ provides the second stage of a logarithmic shifter. Only left rotation is provided at this layer; masking and range checking for shifts are via β and α respectively. As with the other layers, the shift or rotate amount must be replicated across the tribbles of the right operand.

γ.p supplies the PIT opcode's permutations in accordance with Section 4.8.

At this time, there remain either two unallocated spaces for operations not requiring carry decisions, or one space for an operation that needs carry decisions.

4.14 θ operation

The θ SRAM is a single 64k × 16 chip that acts partially like an extension to the control unit. It specifies the right-to-left behavior of carry for addition and subtraction, the left-to-right priority of tribbles for magnitude comparison, a latch function for bit scanning, and an ability to force its output to all zeros or all ones for overloading operation slots in γ. These operations are shown in Table 19.

θ.+ is the ordinary carry propagation mechanism: for any tribbles with the propagate input set and the carry input clear, the carry decision will be identical to the carry decision from one tribble to the right. This behavior is recursive. There is no carry input from the right of tribble zero.

θ.c is the same as θ.+, except that the CPU's Temporal flag from a previous addition or subtraction is included when carry decisions are calculated.

θ.< and θ.> interpret each tribble's propagate and carry inputs as defined in the "compare" column of Table 16. The overall function is to decide which, if either, of the left or right operands is lesser or greater. Priority is given to the decision from the leftmost tribble. The result of the overall comparison is distributed to all carry decision outputs identically. These operations are primarily for the MIN and MAX opcodes, but conceivably could have other uses.

The purpose of θ./ is to allow underinformed decisions that are made in isolation by the α layer's SRAMs to be overruled within the γ layer. The / symbol is chosen to suggest a rightward motion. The premise is that the output of a bit scanning, counting, or NUDGE operation in α is valid until some condition is met. The α RAMs all scan for this condition and set the carry lines true in tribbles where the condition appears. θ./ turns on all of the carry decision flags to the right of the leftmost tribble where the condition appeared, thereby instructing γ to apply some alternative calculation to the remaining tribbles. The leftward counterpart to θ./ is θ.\. A corresponding operation to force the carry decision to zero uses the propagate bits instead of the carry bits. Finally, an operation to invert the previous carry decision is signalled by setting both the carry and propagate bits from the same tribble.

θ.0 and θ.1 force all carry decisions to 0 or 1 simultaneously. These are used to suppress carries for non-additive functions, or to partially specify slots within γ for operations that do not require carry decisions.

4.15 Miscellaneous unary operations

Computations with only one input generally use the left argument, leaving the right argument free to specify which of 64 operations to compute. This affords SRAM ALUs a lot of freebies we wouldn’t otherwise enjoy. The simplest cases occur when everything is done within 6-bit subwords, and the tribbles are completely independent of each other. These are merely lookup tables, and can be used from either the α or γ layer.

Table 21 provides a list of the simple unary operations that are specified to date. Note that the right operand is specified on a per-tribble basis, meaning that up to six operations from Table 21 may be applied across a word during one instruction. Slots 0–8 provide constants 0, 1, and all ones, pass-through and bit inversion, and tests for zero and nonzero with output alternatives. Slots 9–14 offer counts of zeros and ones with total, leading, and trailing options. Slots 15–16 allow tribbles to be individually incremented and decremented.

Slot 17 is the sole operation where the tribble outputs are not identical when given the same input. The left input is a number between 0 and 63, and must be provided across all six tribbles. The right input is 17 (the operation to perform) in all tribbles. The result is the largest 36-bit word the left input can be multiplied by without overflow, assuming the product is unsigned. Another way to view this operation is as a fixed-point reciprocal of the input value, with all ones output when dividing by zero.

Table 22 lists more complex unary operations. These require participation of multiple layers and sometimes need θ to support operations across subwords. For this reason, the slots specified by the right operand are numerically consistent from layer to layer. The name “stacked unary” arises from the involvement of these lock-stepped layers.

Slots 0–2 are a mild instance of self-modifying code, because they permit a word to be negated, zeroed, or left intact on the basis of their right argument. That right argument is computed by a previous stacked unary operation, either slot 3, which identifies whether a word is negative or not, or slot 4, which distinguishes between zero, negative, and positive slots. Using these operations, absolute value can be composed in two instructions, and a sign or signum can be copied from a signed number onto a positive number in two instructions. It remains to be seen whether absolute value will detect overflow when the input is -2^{35}. Slot 4 employs an off-label repurposing of θ.> to determine if any tribble of α is nonzero and quickly disseminate that information to the carry decision inputs of γ. The 2^{-35} input bit for slots 3 and 4 are copied across tribbles via layer β.

There is a one-instruction absolute value operation in slot 5; however, it only works if the result is less than 2^{30}. This operation depends on the fact that tribble addition and subtraction doesn’t ever need to set the propagate and carry outputs simultaneously, allowing simultaneous appearing of these signals to be interpreted as a special condition. This has a brief footnote in Table 16. The α layer assumes the input is negative, but does not invert the bits. But it does output propagate and carry signals as if its bit-negated input were being incremented; e.g., α initiates a sign reversal, but does not alter the word. α_5 operates differently: α_5 outputs all ones if the 2^{-35} input is on, and turns its propagate and carry off. Otherwise, α_5 outputs all zeros and turns its propagate and carry on. When β receives the transposed input word, every tribble now has a copy of the 2^{-35} bit, allowing β to invert the 30-bit word if the input was negative. β will also ensure that γ_5’s inputs are all zero. The carry decision RAM θ is doing an ordinary carry decision as for add and subtract, which γ then uses to increment the 30-bit word that β negated—provided the left argument was negative. But if θ sees the propagate and carry bits both set from α_5, a condition never occurring for other additive operations, θ aborts and sets all carry decisions zero, because the left argument was not negative as presumed. With no carry decision inputs, γ won't alter its input from β, which didn't invert its input from α, which doesn't modify its input to begin with. Thus via a spectrum of down-the-rabbit-hole kludges, we have absolute value in one CPU cycle. But it only works up through 30-bit words, and it is not known yet whether out-of-range arguments will be detected by this operation.

Slots 6–9 are used by shifts and rotates where the number of bit positions is a computed value not known by the assembler or compiler. The left operand is the desired number of places in tribble 0 and is a positive value; the remaining tribbles are ignored. The output is the right argument for a rotate or shift operation. The overall process is described in Section 4.5.

Slots 10 and 11 are used to add the subwords within a word efficiently. This is useful for operations such as Hamming weight, where the number of ones in each tribble are indicated and we desire the total number of ones for the entire word. Slot 10 adds six numbers between 0 and 6, and outputs three numbers between 0 and 12. To achieve this, α replicates the three lowest bits of each tribble into the three upper bits, β packs the six tribbles into three, and γ splits each tribble into two three-bit halves and adds the halves together. Slot 11 further eliminates one of the addends by distributing one of the four-bit inputs into the two remaining bits of each of the other tribbles, which are replaced by sums of up to 24 and up to 15 in those two tribbles. Regrettably, adding the last two tribbles requires two instructions.

Slot 12 reverses all bits of the left operand in one machine cycle as conceived. The PIT and PAT instructions can do this also, but they require two machine cycles. Slot 13 computes the parity of a word, returning a single bit to the right of 35 zeros.

Slots 14 and 16 are special increment and decrement operations on words. They are not checked for wrap-around, and are intended for use when wrap-around should not be checked, such as for Section 4.16's leading and trailing bit manipulation. Bit-reversed increment and decrement occupy slots 15 and 17, wherein the least significant bit is leftmost instead of rightmost. Slot 18 gives bit-reversed negation of a signed word, also without checking for wrap-around.

4.16 Leading and trailing bit manipulation

Several CPUs, including recent x86s, have instructions for trailing bit manipulation. These instructions scan from the right for the first zero or one, and upon finding can invert that bit, isolate the bit by masking out all others, fill up to the bit from the right, mask with ones up to the bit and zero bits to the left, or mask with ones through the bit and zero the bits to the left. Some of these instructions are available with complemented output. Not all cases are covered by these CPUs, and at least x86 doesn't support leading bit manipulation; that is, scanning from the left.

The full set of these manipulations could be implemented by our ALU as stacked unary operations for leading and trailing use, making them accessible via one instruction. But there are 40 of these: search for 0 vs. 1 × invert output or not × leading vs. trailing × five "what to do." This would be a lot of address space to set aside for these rarely-used operations, so we haven't. Instead, all 40 can be accomplished in two instructions by composing already-existing operations: leftwards and rightwards increment and decrement from Table 22 and the bitwise boolean opcodes from Table 7. These 40 formulations are tabulated compactly with examples in Table 26.

5. IMPLICATIONS

5.1 Characteristics of surrounding CPU

SRAM is as useful for implementing CPUs as for ALUs, although the work proposed is for ALUs exclusively. That being said, we can predict some characteristics of these CPUs. They are important to note for three reasons. First, they explain otherwise unexplained inclusions in the ALU. Second, they result in crucial security improvements that motivate our whole undertaking. Third, they encapsulate knowledge that is soon to be very useful.

Static RAM is much faster, more secure, simpler, and more costly than dynamic RAM. I'm fine with three out of four. Unless a computer is to be airgapped, DRAM should not be used for central processing. Buffer memory in an attached storage device might use DRAM, provided that device is carefully isolated. Perhaps an attached monitor has a lot of DRAM, although monitor security is a whole new dissertation topic. But for network cards [22] and main memory [16], DRAM is out.

The Von Neumann architecture where programs and data nest in the same physical address space make assurances of segregation difficult and introduces vulnerabilities. It also creates bottlenecks due to simultaneous needs. Complex workarounds have been implemented in VLSI, which themselves have come with unwelcome surprises. Everything will be simpler with a Harvard architecture, where program memory is on physically separate silicon, and instructions that write to program memory are privileged and identifiable from their opcode alone. No such instructions are to even appear in unprivileged code, and this restriction is enforceable in code as it gets loaded.

The instruction format is minimalist and works like this: 9-bit opcode, 9-bit left operand register, 9-bit right operand register, 9-bit destination register. That's almost our whole world. For branches, a 9-bit opcode and 27-bit absolute destination.[18]

18 The instruction word format is illustrative and may be widened, in which case stated limits change. But this 27-bit space costs about $7 500 to fill as of early 2020 and provides 576 Mibytes of program storage. It's unlikely we can audit a program of that size, so we probably shouldn't run one.

There are no relative branches: we don't need them. Relocatable code remains possible by resolving addresses at load time. There are no indirect branches; these offer a rare speedup such as pointers to functions, but can be exploited to dislodge the program counter from ground truth. Consequently, there are no subroutine return addresses either in data memory or registers. This doesn't mean there are no subroutines, but it does mean control always returns to the right home when these routines end. We will have to forego `setjmp` and `longjmp` in C-like languages. There is no recursion and no stack overflow. This worked out fine in old FORTRAN, and it's still a good idea today. Code can be made recursive where necessary, but whatever stacks are needed are allocated from the heap, and in no way relate to the program counter.

This is a multitasking CPU, so care was taken to design an ALU with no retained state. We don't want special hardware for context switches, nor undue delay, so all registers are stratified by *user*, a small integer akin to a process id. As the operand registers are 9 bits, there's enough space for a 7-bit user to index two 64k × 18 RAMs as the register file. When control is transferred to another user (that is, process), not a single register needs saved or restored. The only change is the 7-bit user identifier. The CPU at liberty to change users with every machine cycle with no penalty; in fact, the CPU can be made faster by doing so. More about this in a few moments.

The program counter also resides in a RAM and is stratified by user. It is neither saved nor restored during context switches. As this RAM doesn't need to hold registers, one might think it can be smaller than the register file RAM. Actually, it's the same size, because it's not a program counter RAM. It's a subroutine return address RAM, and the top of each user's stack is the program counter for that user. The only nonprivileged instructions that manipulate this stack are calls and returns. Underflow is not permitted; returning from the top level either causes an interrupt or simply has no effect. I prefer no effect, because the semantics, implementation, and auditing are simpler.

The flags also reside in RAM. At a minimum, there are flags for zero, negative, temporal range, and range, denoted Zero, Negative, Temporal, and Range. Flags Z and N are familiar and appear in other processors. T indicates that the most recent instruction that could cause arithmetic overflow did, in fact, cause overflow. It's analogous to the carry flag on familiar architectures. R is set whenever T is set, but is only cleared by a flag manipulation opcode. This permits tedious or lengthy calculations to be implemented with no overhead or distraction for checking, and when it eventually must be known if overflow occurred, the R flag can be inspected just once.

Design choices remain to be made around flag handling, such as whether Z, T, and N persist across calls and returns. As the program memory's physical address space is only 27 bits, the return address SRAMs might have unused width where flags can be stored for free. In fact wherever flags are stored, there might be some extra bits available, so the design may provide extra flags. These could have names like X-ray and Yankee, and offer extra amenities for the programmer.

The data memory's hardware protection uses a one-layer SRAM page lookup, indexed by the memory segment and user. For example if 4 Mi words of RAM are installed, the physical address space is 22 bits, possibly arranged as a 9-bit page and a 13-bit offset, making the page size 8 192 words. The physical page comes from a RAM that is indexed by the 9-bit logical page with the 7-bit user. In contrast, the program memory doesn't need any hardware protection or page relocation: every branch is absolute and stored alongside a branch opcode, so the loader easily allocates program memory and segregates users securely.

Because the program memory is not protected by hardware, and branches are always to immediate values, dynamic linking is very simple, and calls into the operating system and libraries are easily restricted to designated entry points. But an argument passing mechanism would need to be worked out. For the most part, I am not an advocate for dynamic linking (too many dependencies), but I am not a fan of so-called "containers" either (too much bloat).

A problem crops up implementing the program counter: there aren't any decent counter chips on the market, and possibly never have been. There aren't even particularly good shift registers, although tristate latches can be found. This doesn’t bode well for ripple carry, or for that matter any other carry, and using an SRAM carry-skip adder seems overkill for this use. My suggestion is to use a shift register with feedback instead of a program counter, either outright across the entire address space, which has drawbacks determining what memory belongs to which user and challenges supporting multiple memory sizes, or in segments of perhaps 13 bits where a branch to cross into the next segment appears every 8192 instructions. This would not be the first CPU to use a shift register as its program counter with a mechanism to switch pages. The TMS 1000 microcontroller, introduced in 1974 by Texas Instruments, worked exactly this way.[19] One perquisite of the paged approach for our multitasking CPU is that code fragmentation becomes of no concern.

The 512 registers available for each program running will eliminate most loads and stores. As long as the registers are not exhausted, they can referenced in programs by variable name instead of by number, with final register allocation done by the linker or even the program loader. The number of registers needed is no greater than the number of variables simultaneously in scope. Data structures will continue to live in normal data memory, both to conserve registers and because no indirect register access will be supported.

By increasing instruction word size beyond 36 bits, still more registers can be added per user at the cost of reducing the number of programs that can run simultaneously. Alternatively, a very wide instruction word can allow programs to access any or all registers in the system, 65536 at a minimum. The kernel would allocate registers to programs as they are loaded, so that specialized applications that call for a large number can be accommodated, while no registers are held back from use simply because only a few programs are running.

There is no security issue letting the loader allocate registers, because running programs can only access the registers that appear in their object code, and the system controls the loading of that code. But there is an ironic cost when increasing the program word size to, for example, 64 bits (262144 registers and a 10-bit

19 TI did not document this 6-bit counter’s operation, so for posterity, ABCDEF (new) = BCDEFG, where G = (A XNOR B) XOR NAND(BCDEF). The complexity is to make the period 64 instead of 63; otherwise, G = A XNOR B would have been adequate. Formulas derived using assembler listings from [34].

opcode), because the program memory grows faster than the register file. Using early 2020 pricing, giving programs up 512×512 registers instead of fixed at 512 increases cost of manufacture by about $50 per million words of program memory. But there is no speed penalty, except for initial register allocation.

There are a few dual-port RAMs on the market which could allow simultaneous retrieval of the left and right operands, but they aren't fungible and their prices reflect their seeming indispensability. They also paint a target on our system in terms of supply chain attacks: buyers of ordinary SRAM might be doing mundane things, but buyers of dual-port SRAM are building networking equipment, CPUs, or something else of interest to adversaries. While I'm out of my area of knowledge, I conjecture that the greater complexity of dual-port RAM might makes it harder to inspect by microscopy, emissions analysis, or fuzzing for irregularities.

Rather than use dual-port RAM, I would build two register files that use the same address lines, and name them "left" and "right" after the operands they supply. ALU results would be written to both copies, perhaps facilitated by the otherwise-unused extra byte outputs of the γ RAMs.

Duplicate register files are particularly helpful if the system includes a hardware multiplier along the lines of Section 2.9. Most likely, the multiplier would be connected in parallel with the ALU, with the tristate outputs and possibly clock inputs of the two subsystems coordinating which one is active. The multiplier will produce a two-word result that needs stored simultaneously, or there will be penalties in complexity and performance. With two register files, the left and right registers can receive the high and low order words. But this means multiplication instructions will break an invariant that would otherwise hold: the result of a multiplication appears to change depending on whether it is the left or right operand of a subsequent instruction, and programs must account for this.

Assume for simplicity of illustration that $A \times B$ is less than 2^{35}. The left copy of the register the product is written to is zero, and the right copy contains the full product. Now suppose the expression we want to compute is $A \times B - C$. If we try this using a subtract instruction, the result is $-C$, because the ALU is given zero when it loads A × B as the left operand. Instead, we need a reverse subtract instruction that computes − C + A × B, which will load the low word of the product as the right argument of the subtraction. This is why reverse subtraction appears throughout Sections 4.2, 4.10, and 4.11, as well as Tables 6, 14, and 15. Product subtraction is often needed by programs, particularly in vector processing where speed matters. But we don't go so far as to offer reversed-argument versions of any opcodes other than subtraction: the architecture doesn't support it, and the ALU memory isn't available. We can always resynchronize unbalanced registers in one cycle by adding zero, the XL or XR opcode (Table 7), or innumerable other constructions. Subtraction is treated as a special case.[20]

This CPU doesn't offer many modes for addressing data memory, but the ALU can be tasked for address computation during load and store instructions. The virtual address can be any ALU function the control unit knows how to dispatch. When the CPU stores to memory, the last nine bits of the instruction word specify the source register instead of the usual destination register. The address functions would include at least A(dd), S(ubtract), and NUDGE, and possibly many more depending how many opcodes are available to assign. These operations would each have load and store variants. Opcodes to write to program memory don't have to be as numerous or diverse, as they are only used when programs are loaded.

Here is a simplified order for instruction execution:

1. fetch instruction from program memory
2. decode instruction
3. fetch left and right operands from register file
4. ALU α
5. ALU β and θ (simultaneous)
6. ALU γ
7. map logical to physical page (memory operations only)
8. write result to register *or* memory-register transfer

20 To improve binary compatibility across a variety of similar CPUs, reverse subtraction should exist for all CPUs, including ones that don't support hardware multiplication.

Each of the steps uses one SRAM time slot, so we begin to see the parameters that affect how fast the CPU will run and what circuits need to be included. I have not tried to optimize anything here, although there are opportunities. A critical contingency is that if we try to somehow pipeline the CPU, steps 8 and 3 present a hazard to overcome. The simplest mechanism is to always wait until an instruction's result has been written to the register file before fetching the next instruction's operands. This won't catastrophically reduce the clock speed, but it would be the long pole in the tent. Possible speedups include:

- Relocate instruction fetch and decode to the cycle preceding execution. This can happen in combination with other remedies.
- Never allow one program to run during consecutive cycles. This method is already popular and is the reason many CPU cores claim to run two threads at a time. What's actually happening is that these cores can't keep up with their own operation, so they alternate between running two programs at half speed while calling this oddity a benefit.
- Eliminate one time slot by computing addresses with a 2-layer carry-skip adder instead of with the ALU. This adds 5 SRAMs for data memories of up to 16 Mi words, otherwise 7 SRAMs.
- Require addresses to be fully computed prior to loads and stores. This lets page translation to happen during the ALU's time allocation and requires no additional RAMs, but adds another instruction to most store and load operations. It's unclear whether a net performance gain or net loss would result.
- Alter cycle timings based on the type of instruction executed.
- Provide direct operand forwarding from γ to α, possibly only supporting the left operand. One or more of the layers may require clocking. The redundant output bytes from γ may provide helpful isolation. Forwarding must take care to accommodate multitasking, and vice versa.

Of these speedups, I would begin with the first listed—instruction fetch and decode one cycle ahead—and perhaps go no further, on the condition that any additional clocking or circuits the solution adds won't negate the advantage derived. A problem to work around is that as a branch is being decided, the next instruction is already being fetched into the pipeline. If fetches happen late in cycles and branch decisions happen early, perhaps unwanted fetches can be prevented in advance. A second option is to pass a CPU flag (that must be retained across context switches) that prevents the next instruction from writing results anywhere. A third option, my favorite actually, is to embrace the hazard and tell the programmer that all instructions that immediately follow branches will be unconditionally executed. In many cases it wouldn't matter: a register would be loaded with a value that is never used. In scenarios where something might go wrong, the programmer, assembler, or compiler can place no-operation instructions after questionable branches. In some situations, instructions can be reordered to take advantage of this behavior by getting an early start on the code being branched to.

The second speedup—so-called hardware multithreading—isn't a speedup in the conventional sense, but it does allow maximal use of the CPU on the condition that at least two programs are in a runnable state. It would be straightforward and not much hardware added to have a small table of 7-bit users and cycle through them between instructions. The kernel could manipulate the table for a great variety of CPU time allocations. But there is one catch: if there will normally only be one program that needs to run, this "enhancement" will slow the processor down, not speed it up.

The CPU will require opcodes and hardware for loading immediate values into registers. Data for these values presumably replaces the left and right operand register numbers in the instruction word. Immediates can't be loaded from data memory, because the addresses the immediates are loaded from are immediate values themselves.

One security consideration, which extends beyond the CPU, is that the operating system must control all bus activity. Just because a device is physically attached or soldered to the system board does not assure that device is benign.

Direct access between devices and data memory is fast, simple, convenient, elegant, and prone for exploitation. Without a secure bus, there is no secure machine.

All peripherals call for careful vetting, but mass storage devices are especially concerning due to their centralized access to information and the eventuality that we'll still buy them from a small number of factories. Here is a principal need for the mixing opcodes of Section 4.9. I suggest using a pseudorandom variable to split local storage across two SSDs, so that neither storage device has the information, connectivity, or processing power to infer or divulge anything about their contents. A CPU-devouring pseudorandom function is probably not necessary, as the only computing hardware available to an adversary would be the isolated controllers for the two SSDs. Retrieving the information need not involve a key: the information can be split such that an XOR across the two volumes will recover everything. Note that this control measure has nothing to do with encrypting or authenticating data at rest, but serves only to blind the storage controllers to the data they handle.

5.2 Mixed signedness and practical overrange checks

Most CPUs with a word size of n bits do arithmetic modulo 2^n, and until recently it hasn't been in fashion to pay attention to the possibility of wraparound. Now that there is wider realization of vulnerabilities that stem from discontinuities in arithmetic, unhealthy workaround have appeared. For instance in the C language, code that historically read

```
c = a + b;
```

now has to read

```
if (   b > 0 && a > (INT_MAX - b)
    || b < 0 && a < (INT_MIN - b) )
    longjmp(CATCHIT, SIGNED_ADD_OVERFLOW);
else
    c = a + b;
```

to comply with CERT rules for "secure coding" [23]. This isn't secure coding. This is untenable. This scheme not only destroys the C language, but also cripples the CPU in more ways than we have space to consider.

This really isn't an SRAM ALU issue, but a prehistoric gene that has been passed down too many generations of silicon. But as we are here, there is a moral imperative to do rightly. The correct paradigm is

```
c = a + b;
...          /* all the calculations that need done*/

if (something_bad_ever_happened())
    deal_with_it_here();
```

Places where a program needs to know whether it's still on or off its rocker are called *observation points* in some literature [24]. This reference places responsibility for range checking on compilers—clearing the source code of vulgar gymnastics, but profoundly disrespecting the machine code and CPU. There is a better way.

Knowledge of whether or not an addition or any other arithmetic operation has a result that fits within its destination is as critical to correctness as knowledge of the result itself. Our ALU of Part 4 ensures that along with operands for arithmetic tasks, the semantics—particularly the signed or unsigned intention—for each operand is supplied. This ALU goes still further, by indicating for both result interpretations whether the 36-bit output is valid *for the operands supplied*. The programmer, compiler, assembler, or interpreter is freed from reasoning as to whether overflow might be possible for every smackerel of arithmetic. Not only is this important on account of human engineering considerations, but also because the possibility of wraparound is undecidable according to Rice's theorem [25].

When unexpected wraparound occurs, it is not necessary to drop everything. Calculations can continue, with the stipulation that the results at the end will be incorrect. Once the calculations come to the end, we can then ask the CPU if there might be something wrong the results, and deal with that possibility after any deep call stacks or nested conditionals have been unwound. But the CPU must not have forgotten, as most today do, that a result did not fit. This is why our CPU is proposed with an Range flag and a Temporal flag. The T flag changes from instruction to instruction; it indicates that the most recent arithmetic result did not fit in its destination. Setting T also sets R, but resetting T does not modify R, which is only cleared by an express flag manipulation. Most software needn't check T ever. T is for code that operates on integers larger than a word and other legitimate but infrequent wraparound scenarios.

Arithmetic semantics can cause errors needlessly, as in cases where there is no result word, but flags aren't set as intended. Our ALU design is careful about such things as comparing signed numbers with unsigned numbers: *we make this legal and guarantee correct results* for all inputs and input semantics. So we can compare 2^{35} (0x8 0000 0000) with -2^{35} (also 0x8 0000 0000), and the ALU knows these are unequal and that -2^{35} is lesser. It can also negate the pole: $-(-2^{35})$ will succeed when written to an unsigned register, but will warn using the T and R flags when written to a signed register.

There is no revolutionary technology here, and it doesn't matter whether we are early or late in the line to propose a solution. I believe that all network-connected ALUs need to handle arithmetic semantics correctly, as our ALU endeavors to do.

5.3 Security

The CPUs the proposed research leads toward significantly contract surfaces for attack. Here are a few highlights. First, the suggested designs have no components that ought to be capable of persisting any state. It's technologically possible to embed sinister microcontrollers in fake RAM, but such devices would have to be sold at scale via the right channels to the right victims and soldered into the right locations on the right devices. These counterfeits would need to match or beat prevailing industry prices, still contain the advertised RAM, and evade detection when random samples are inspected electronically or microscopically. This scenario is biased strongly in the defender's favor.

No ALU component ever sees more than six bits of any operand; however, the register file might see 18 consecutive or even all 36 bits of every register, unless the registers are striped. The data memory and peripherals are even larger targets. In the CPU itself, there are three places for attackers to find cover: compromising multiple components (difficult to coordinate), altering the circuit board (difficult to conceal), or altering software to feed leaked information a few bits at a time through a malicious component (difficult to evade audits). It should be noted that physical alteration of computing machinery for the purpose of compromise is a mature career field [26], as is diversion of new victim systems from distribution facilities [27].

CPUs derived from the ALUs of this research are immune to all attacks that exploit the temporality of dynamic RAM, as no DRAM is present, and all attacks on cache timing, as no cache exists.

The CPUs we advance are also immune to malicious insertion of code segment or stack contents via malformed input. They protect the contents of their instruction pointers to an extent not found in traditional computing machinery.

The emergent CPUs offer exceptional protection against attempts to exploit numeric overflow, provided that software performs a trivial and unintrusive check. They are simultaneously resilient with respect to signedness semantics.

The CPUs that our ALUs will enable can have dazzlingly simple memory protection and multitasking mechanisms. Likewise with privilege restrictions, which do not require any additional hardware. By filtering privileged opcodes at load time, the kernel can ensure that none exist in memory for nonprivileged programs to execute in the first place. Branches and calls to privileged code are also filtered by the loader.

The proposed ALUs are not ideal in their cryptographic prowess, but they are not too bad. At the very least, they minimize their own prospects for building in cryptographic backdoors. They also excel at applying their full computational capacity for encryption and decryption, albeit not with ciphers that have been standardized yet.

The CPUs we are enabling can be specified, adapted, built, repaired, and used without restriction by their owners, not by outsiders who don’t have skin in the game. Except for within a very small selection of interchangeable commodity chips, there will be no unknown, unreadable, unauditable, or unserviceable microcode, firmware, or circuitry. They will be inspectable after manufacture. They will not support so-called digital rights management, serial numbers, or other anti-consumer measures. They are incapable of becoming "bricked." They will allow unrestricted reverse engineering, academic study, security research, documentation, replication, and most importantly, offline and networked use.

5.4 Anticipated performance

For an apparatus that is assembled by soldering, Part 4's ALU is very fast. Except for the task of obtaining the CPU's zero flag, where we're afforded a little more time, there are exactly three gate delays. The "gate" in this instance is a small asynchronous static RAM, and delay per gate for low-cost devices is 10 ns. Parts for one ALU cost about $40 as of early 2020. Using the fastest chips available raises cost for parts to $100 and reduces access time to 7 ns per gate. The remainder of this section will assume that the cheaper, more broadly available 10 ns parts are opted.

The CPU cycle steps listed at page 78 offer hope that a CPU can reach 10 MIPS without heroic measures. Our bare ALU, when allowed a full 10 ns for capacitance and board trace setbacks, tops out around 25 MIPS, but no thought has been given to inputs or outputs we might wish to clock. These figures sound lackluster compared an iPad, but neither architecture is trying to be the other. As I pause to reflect that we can build a 10 MIPS CPU in our kitchen using fungible components while under a stay-at-home order, I discover we aren't as powerless as once thought with respect to our silicon.

We can't make apples-to-apples comparisons of CPUs or architectures. Instead, we need to consider suitability for the particular uses we intend. Intel's 80486 was all the rage when I was an undergraduate—correction, many of us found no need to upgrade our 286s (that's not a typo) until well after I graduated. A 10 MIPS machine based on our ALU can do short multiplication as fast as a 40 MHz 80486, and our

ALU doesn't even have a multiplier! And our word size is a tad larger.

There are a couple more comparisons we can draw between the capabilities of our ALU and the 80486. The 486 was the first CPU in the x86 line that could add integers in one clock cycle. Our ALU also adds in one cycle. We also shift or rotate any number of bit positions in one cycle. The 486 required two; it would be the Pentium that introduced one-cycle shifts to Intel's lineup. Our ALU doesn't include floating point operations, yet might not fall much behind the 486's numeric speed for single-precision tasks. But this is only because Intel's CPU did not offer single-precision instructions. The 80486 also wins in another respect: its clock rate would eventually reach 100 MHz. The architecture of this proposal might never see even half of that speed. Speed is one of the things VLSI does best.

One other performance characteristic: what is the physical size of this ALU? Plate 20 shows an actual-size mockup using the smallest readily-available components, although these are not the easiest packages to position and solder by hand. The dark rectangles are the SRAMs, and the light squares are analog switches used to initialize the RAMs during startup. An ATM card silhouette is visible behind the components for comparison.

5.5 Suitable uses and limitations

I can testify that It's practical to write hundred-page documents on 286-class desktop computers, write and build software, use spreadsheets, and send and read email. For all of these, there are eventually limits. But the proposed ALU is a much stronger design than the 80286: the word is 20 bits wider, the memory model is flat, there are no wait states, preemptive multitasking and memory protection are specified, and fewer cycles are needed for just about everything. So our ALU can be used for practical text-oriented desktop work and basic correspondence.

This ALU is fast enough to control most systems that physically move: industrial and commercial devices, factory automation, electric grid switching, wells and pumps, heavy machinery, trains, dams, traffic control, chemical plants, engines, and turbines. Its suitability for use in life support equipment, broadly construed to not just mean medical equipment but also aircraft, space travel, defense systems, and other uses where failure assures death is less certain. It can probably be certified for use, but effective controls against bit upset from radiation, electrical noise, mechanical strain, temperature, moisture, power supplies, and damage must be engineered.

Our ALU is also fast enough for some uses not involving motion, such as peripheral and device controllers, telephony, even Ethernet switches to medium speeds. A key reason Ethernet can work is that routing and firewall decisions derive from packet headers, not their payloads. TCP over IPv6 at 100 Mbit/s is only 8 333 packets per second at 1 500 bytes each. We can keep up if filtering and NAT rules aren't too complex. But when there is no payload, which mercifully isn't very useful, there could be 208 000 packets per second, each containing a 15-word header. Hardware that can sustain this faster rate would be more involved, but probably can be built.

It might sound archaic to consider building 100 Mbit switches, but the reality switch manufacturers have given us is dystopian. A leading U.S. producer admitted six times in 2018 alone to building secret backdoor passwords into its networking and network management products. Unfortunately, these are not isolated cases, and confidence in scores of once-respected brands has been extinguished. Open-architecture, transparent network hardware can help us rebuild, even if switching speeds will be modest as a consequence. My lab, with less than 20 Mbps download and less than 1 Mbps upload to the outside world, has used an improvised medium-speed device as its firewall for two years.

There are applications our ALU can't accommodate without either specialized designs and heavy parallelization or offloading to other kinds of processors. These include image and video processing, supervised and unsupervised learning, self-driving vehicles, fast rasterized or vector graphics, common symmetric ciphers at high speeds, frequent asymmetric cryptography, Bitcoin mining, large database with fast response times, ambitious parsing, fast page layout, and unmanned aerial vehicles with stringent weight budgets. In lay terms, we can't surf the Web like we do today. Digital signal processing, like Ethernet switching, falls in the middle. Some audio tasks, even some software-defined radio tasks at low intermediate frequencies, should be no problem. But fast radar front ends will have to use a different ALU.

5.6 Social and psychological benefit

As I write in April 2020, much of the world isn't thinking deeply about surveillance or censorship; instead, humans are connecting online as never before. But these conversations force a choice between two perils: either people will say too much, or they will say too little.

On June 7, 2013 Barack Obama told the country "When it comes to telephone calls, nobody is listening to your telephone calls" [28]. Of course nobody is listening. There aren't enough personnel. But most governments have the network capacity, CPUs, electricity, storage, and perceived authority to record all calls and transcribe them to text, just in case. To what extent they do or do not is uncertain, varies with jurisdiction, and is subject to change.

Who will tell their mothers the truth on the telephone, let alone write to them online? Today's close relationships are accustomed to instantaneous connection, but most are not local. Responses vary from individual to individual. Many, feeling they have nothing to hide, try not to be inhibited by risk of interception. Many others are more guarded about telecommunication. A third set cite an ethical obligation to supervise the use of government's power and to curb that power where necessary, whether or not they would otherwise have nothing to hide.

I would personally breathe more easily if at least one of my computers was both usable and inspectable. My TRS-80 collection is the latter, but these were only designed as executive toys. The text editor inserts characters by moving everything that follows over by one byte. That doesn't scale. As for my usable computers, a majority of contemporary attacks owe their viability and continuity to the noninspectablity of the equipment. It’s not enough to run software that is free of defects. Without secure hardware, there is no secure software.

The 1992 riots in Los Angeles left the county I lived in unexpectedly short of fuel and cash, with no more of either being delivered. Today, millions face not only new concerns about availability of lifesaving machinery, but also life savings, not to mention small- and large-scale commodities like elastic for sewing and tissue for baths. To first order, there is *no* computing hardware that works transparently enough to be counted on when we go online. This would be a terrible position to start from when a different catastrophe forces the entire world to seek shelter offline.

5.7 Originality of research and contributions made

The computer science and computer engineering literature is vast, and technology that can index by concepts within article text generally does not exist. Moreover, paywalls of various faces, whether requests for money in exchange for access or distribution solely on paper, block most of the indexing that could be done. This limits what claims I can make. Where I can prove I was not the first to discover something, I can say that confidently. Otherwise, the most I can say is that I believe I found something independently, and that I made reasonable efforts where appropriate to search for prior work. There is some fairness, however, in the knowledge that for anything first discovered by me, failures of our search tools could ultimately credit another researcher for its discovery.

I owe a shout-out to the dozens of hobbyists, students, and instructors who have built and documented their own CPUs. A partial list is online at [29], and all make interesting and worthwhile reading. Of these, [30] is noteworthy for running an actual operating system and occasionally facilitating remote use by visitors. This “glue logic” system, although operating below 1 MIPS, encouraged me to move away from using separately-packaged transistors to build CPUs.

Another noteworthy CPU is Olivier Bailleux’s Gray-1 [31], unique because of its nearly-all-memory implementation. Even the registers of this 8-bit CPU are made from EPROM. The ALU's operations are add, subtract, shift one position left or right, and NAND. The EPROM tally is 164 Mibit, and the CPU can execute 30 000 instructions per second [32]. Notwithstanding its modesty, the Gray-1 is our work's nearest known predecessor.

Where I mention SRAM in the following claims, it is meant to encompass anything that could work as SRAM works, whether ROM, EPROM, any kind of table lookup mechanism, PLD, FPGA, flash memory, print in books, whatever.

With respect to SRAM ALUs in particular, I have found no literature suggesting their use to implement carry-skip adders, whether these are 2-layer, 3-layer, or hybrids of both that can add words of a few hundred bits within three gate delays. Carry-skip adders are standard, of course, but their SRAM versions are novel and, as discussed in Section 2.4, exhibit distinct behaviors.

The generalization of the propagate and carry bits to handle non-additive functions, surely someone must have thought about, and probably with no consideration given to SRAM. But what this concept might even be called eludes me, and the branch of mathematics it's in might not be one I've heard of. With this said, I have shown the usefulness of overloading the propagate and carry bits from the stages of an adder to do many other useful operations, as Section 4.14 shows.

Fast SRAM multipliers as illustrated in Section 2.9, as well as the arbitrary geometry adders of Section 2.3 that they depend on, aren't in the literature as such. But the contribution by me is, as with my other contributions, incremental at most. Wallace [17], Dadda [18], and others reasoned about how partial products could added efficiently years before even I was born, and I definitely read up on their work in the process of doing mine. That my multipliers reduce the subproducts in fewer steps than Dadda's or Wallace's, I can't take credit for. Instead, it's SRAM's expressivity in operation. Moreover, the established VLSI multipliers do work faster and use fewer transistors than we can. But our multipliers are auditable, and we can manufacture them ourselves.

The short and long in-software multiplication schemes of Section 4.6, I have not encountered elsewhere for SRAM, but table-driven multiplication schemes were around before dirt was new. Some ingenious speedups have found use over the years when circumstances permitted, such as the method of quarter-squares for small factors [33]. Unfortunately, what it would take to implement a practical quarter-square-like method for SRAM does not buy us much speed. Building half-word multipliers and using them four times doesn't fare any better. My experience with SRAM ALU research is that without special hardware, multiplication is slow. With hardware added, the system costs more. In-between approaches have the drawbacks of both, and the benefits of neither. But there is a significant accomplishment in our work for software multiplication schemes: even without a hardware multiplier as such, the α layer tables out-race the 80486 for short multiplication, and they aren't much slower than the 486 for long multiplication.

Logarithmic shifters are standard in VLSI, but I don't believe that they appear in the literature for SRAM. Moreover, VLSI shifters are more "purely" logarithmic in that they shift in large amounts, then in small amounts, or vice versa. SRAM logarithmic shifters don't really shift anything: they swizzle and then permute, or permute and then swizzle. They are a creature of SRAM's limitations and power. I went through several iterations of shifter designs before an elegant one manifested.

Using SRAM to multiplex or swizzle is an abuse of silicon machinery, just like everything else in this proposal. I've not found this technique suggested by anyone—too many transistors, and way slower than called for. Except that human misbehavior has poisoned the more rational methods with prospects of deceit. Thus our SRAM approach to swizzling turns out to be useful, independently discovered, and potentially novel.

ALUs comprised of superposed carry-skip adders and logarithmic shifters are, I am confident, original to this work. I have found neither of the two essential SRAM components mentioned anywhere. Here again, I arrived in small steps. By 2016, I had been thinking about end-user-built CPUs seriously enough to make some notes and drawings, but my expectation was that I would use discrete transistors and connect them using a pick-and-place machine and reflow oven. This expectation persisted into early 2019, when relaxing my criteria somewhat, I started designing a CPU around surface-mounted “glue logic” chips. Before long it became evident that the famed components of the 1970s are either no longer made or not updated to contemporary speeds. An SRAM carry-skip adder followed from necessity, but multiplexers were still to be used for shifting. But then there were no decent multiplexers, so RAMs with extra input bits next to the subwords were used to nudge a position or so in either direction. This design was known to waste much RAM and could not shift in constant time. Eventually I figured the logarithmic shifter out, and drew it into a 32-bit ALU with a 2-layer adder that wasn’t superposed with the shifter. It was a very good ALU, and I worked out a lot of specifics. That ALU called for 80 Mibit of SRAM. But then in January 2020, a night of poor rest and relentless ideas gave me the 3-layer ALU with adder and shifter superposed as shown in Plate 17. Its design was more symmetric and only required 21 Mibit of RAM components, within which 10.5 Mibit might not be used. The semi-swizzler of Section 2.7 and 2-layer ALUs of Part 3 were not realized by me until March 2020.

S-box substitutions as our ALU does in α and γ are generally by table and routinely implemented in RAM. What I do not know is what ciphers might exist, if there are any, that do their permutation steps using S boxes, as we do in layer β. More commonly, mixing between subwords is done using wires or by another transformation that is faster than S-boxes.

I have never seen the NUDGE pointer operation before; I designed it to address my own observations of the inefficiencies of pointers within structures. I am aware of no CPU with an operation of this type.

I do not know of CPUs that handle arithmetic semantics correctly in light of contemporary needs. I would expect research literature to shine some light here, but no urgency to be felt by manufacturers or software developers to depart from decades of sunk costs and vendor lock-in. There was little motivation to search for papers about arithmetic overflow: I was decided already as to what I would do, and whatever useful material may have been published by others at the hardware level, no one seems to be heeding it.

I have several features planned for the self-hosted assembler that might be unusual or novel, but they are applicable to assemblers in general as opposed to specifically for SRAM ALUs. These features, therefore, are only relevant to this proposal to the extent that our assembler implements them. Considering that the assembler has not been written as of this date, it is not yet time to enumerate these features or estimate their novelty.

6. REFERENCES

1. Jo Van Bulck et al. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. *41st IEEE Symposium on Security and Privacy (S&P 20)*, (pandemic all-digital conference).
2. Mark Ermolov. 2020. Intel x86 root of trust: loss of trust. (March 2020). Retrieved April 4, 2020 from https://blog.ptsecurity.com/2020/03/intelx86-root-of-trust-loss-of-trust.html
3. Paul Kocher et al. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, (San Francisco, CA), IEEE, 1–19.
4. Moritz Lipp et al. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium*, (Baltimore, MD), USENIX Association, 973–990.
5. Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. At *Black Hat USA 2018*, (Las Vegas, NV), white paper.
6. CTS Labs. 2018. Severe Security Advisory on AMD Processors. CTS Labs, Tel Aviv, Israel.
7. Mark Ermolov and Maxim Goryachy. 2017. How to hack a turned-off computer, or running unsigned code in Intel Management Engine. At *Black Hat Europe 2017*, (London, UK), slides.
8. Erica Portnoy and Peter Eckersley. 2017. Intel's Management Engine is a security hazard, and users need a way to disable it. (May 2017). Retrieved April 4, 2020 from https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it/.
9. Joanna Rutkowska. 2015. Intel x86 considered harmful. (October 2015). Retrieved April 4, 2020 from https://blog.invisiblethings.org/papers/2015/ x86_harmful.pdf
10. Georg T. Becker et al. 2013. Stealthy Dopant-Level Hardware Trojans. In *Cryptographic Hardware and Embedded Systems – CHES 2013*, (Santa Barbara, CA), Springer, 197–214.

11. Sergey Bratus et al. 2012. Perimeter-crossing buses: A new attack surface for embedded systems. In *Proceedings of the 7th Workshop on Embedded Systems Security (WESS 2012)*, (Tampere, Finland).
12. Pavel Dovgalyuk. 2019. Relay computer computes 7 digits of pi. Video retrieved April 4, 2020 from https://www.youtube.com/watch?v=bOOCfx2EN10
13. Mischa Schwartz and Jeremiah Hayes. A history of transatlantic cables. *IEEE Communications Magazine*, 46, 9, (Sep. 12, 2008), 42–48. DOI: https://doi.org/10.1109/MCOM.2008.4623705
14. Jin-Woo Han et al. 2019. Nanoscale vacuum channel transistors fabricated on silicon carbide wafers. *Nature Electronics*, 2, (Aug. 26, 2019), 405–411. DOI: https://doi.org/10.1038/s41928-019-0289-z
15. The MOnSter 6502. Retrieved from https://monster6502.com/.
16. Onur Mutlu and Jeremie S. Kim. 2019. RowHammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* DOI: https://doi.org/10.1109/TCAD.2019.2915318
17. Christopher Stewart Wallace. 1964. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, 1, (Feb. 1964), 14–17.
18. Luigi Dadda. 1965. Some schemes for parallel multipliers. *Alta Frequenza*, 34, 5, (May 1965), 349–356.
19. Robert G. Brown et al. 2020. Dieharder: A random number test suite. Retrieved from https://webhome.phy.duke.edu/~rgb/General/dieharder.php
20. Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, 456–467. DOI: https://doi.org/10.1145/2976749.2978423
21. Tetsu Iwata. 2006. New blockcipher modes of operation with beyond the birthday bound security. Retrieved April 4, 2020 from https://www.nuee.nagoya-u.ac.jp/labs/tiwata/cenc/docs/cenc-fse2006full.pdf
22. Andrei Tatar et al. 2018. Throwhammer: Rowhammer attacks over the network

and defenses. In *2018 USENIX Annual Technical Conference*, (Boston, MA), USENIX Association, 213–225.

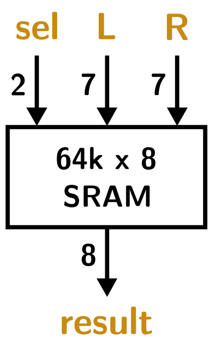
23. Robert C. Seacord. 2014. *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems* (2nd. ed.). Addison-Wesley, New York, NY, 112–118, 126–135.
24. David Keaton et al. 2009. *As-if Infinitely Ranged Integer Model* (2nd. ed.). Technical Note CMU/SEI-2010-TN-008. Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA.
25. Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74, 2 (March 1953), 358–366. DOI: https://doi.org/10.1090/s0002-9947-1953-0053041-6
26. National Security Agency Advanced Network Technology Division. 2008. NSA ANT catalog. Retrieved April 4, 2020 from https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf
27. Jacob Appelbaum et al. 2013. Documents reveal top NSA hacking unit. *Spiegel*, Dec. 29, 2013. Retrieved April 4, 2020 from https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html
28. Barack Obama. 2013. Statement by the President. June 7, 2013, (San Jose, CA), transcript. Retrieved April 4, 2020 from https://obamawhitehouse.archives.gov/the-press-office/2013/06/07/statement-president
29. Warren Toomey (Ed.). Homebrew computers web-ring. Retrieved from https://www.homebrewcpuring.org/.
30. Bill Buzbee. 2010. Magic-1 is a completely homebuilt minicomputer. Retrieved from http://www.homebrewcpu.com/.
31. Olivier Bailleux. 2016. The Gray-1, a homebrew CPU exclusively composed of memory. Retrieved April 4, 2020 from https://bailleux.net/pub/ob-project-gray1.pdf
32. Olivier Bailleux. 2016. A CPU made of ROMs. Watched April 4, 2020 from

https://www.youtube.com/watch?v=J-pyCxMg-xg

33. James Whitbread Lee Glaisher. 1889. The method of quarter-squares. *Nature* 40 (Oct. 10, 1889), 573–576. DOI: https://doi.org/10.1038/040573c0

34. Texas Instruments Inc. 1975. *TMS 1000 Series MOS/LSI One-Chip Microcomputers Programmer's Reference Manual*. Texas Instruments Inc., Dallas, TX.

Colors in this proposal are Wright State University branding, with additional colors from https://colorbrewer2.org by Cynthia A. Brewer, Geography, Pennsylvania State University.



sel	result
00	L NAND R
01	L XOR R
10	L + R with carry
11	L − R with borrow

Plate 1. Simple lookup element

```
    101                 01001
    0100                110  00
     001           +    0 01010
     110              ---------
+    101              010001110
  ------
  011010
```

Plate 2. Arbitrary geometry addition

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	C
1	0	0	0	0	0	0	C	1
2	0	0	0	0	0	C	1	1
3	0	0	0	0	C	1	1	1
4	0	0	0	C	1	1	1	1
5	0	0	C	1	1	1	1	1
6	0	C	1	1	1	1	1	1
7	C	1	1	1	1	1	1	1

Plate 3. Subword carry decisions for 3-bit addition

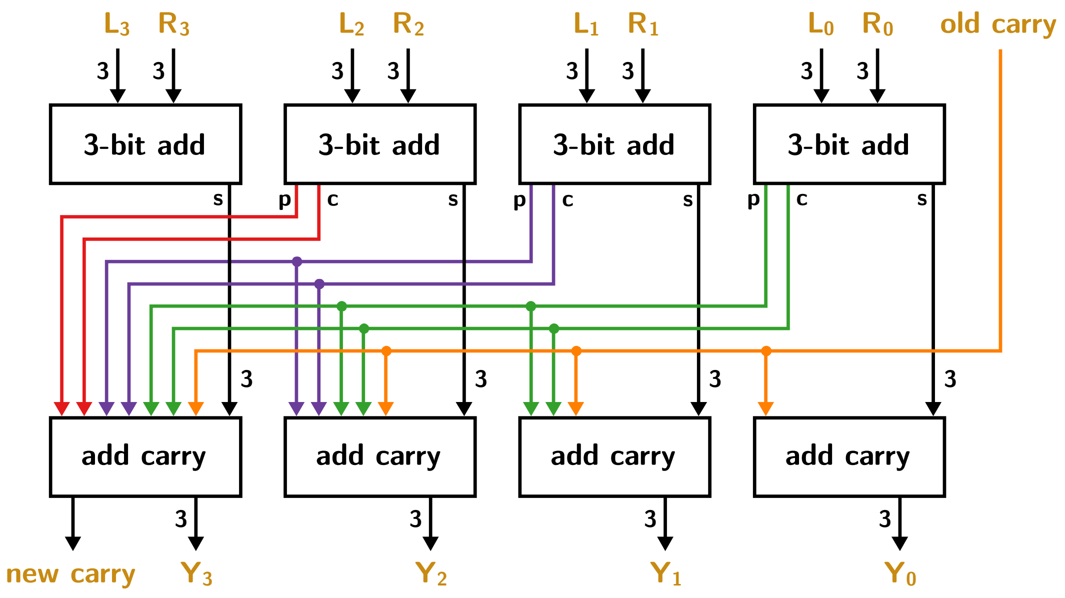


Plate 4. 2-layer carry-skip adder

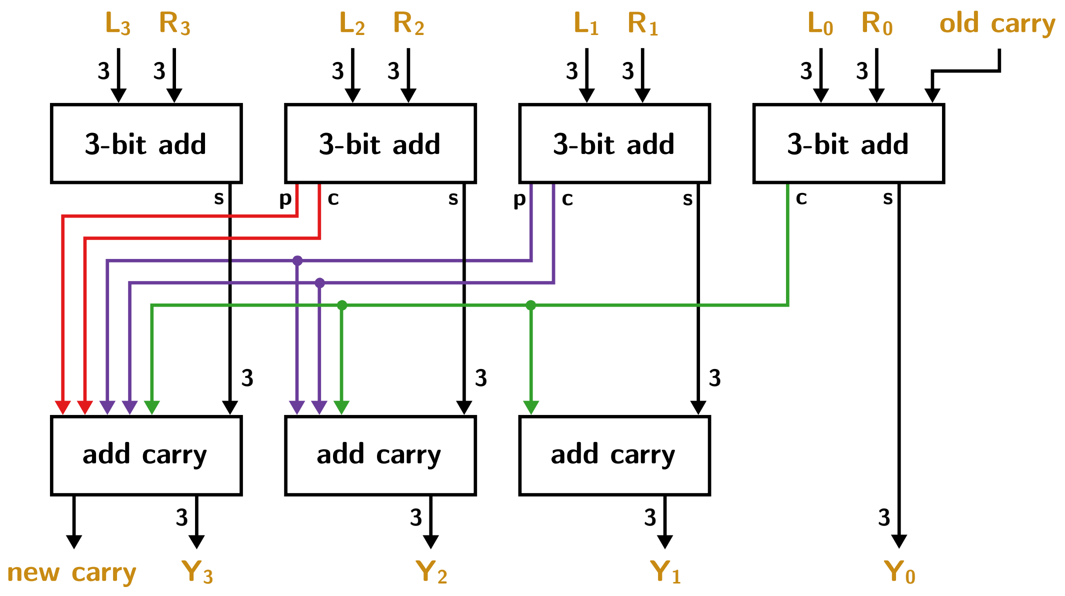


Plate 5. 2-layer carry-skip adder: old carry via top

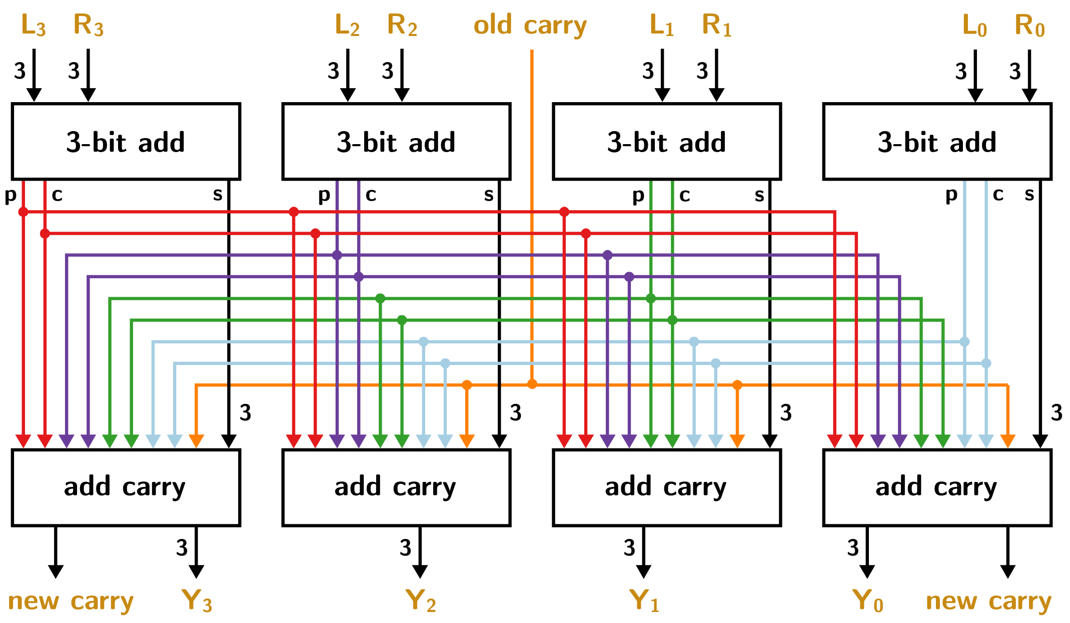


Plate 6. 2-layer carry-skip adder: bidirectional

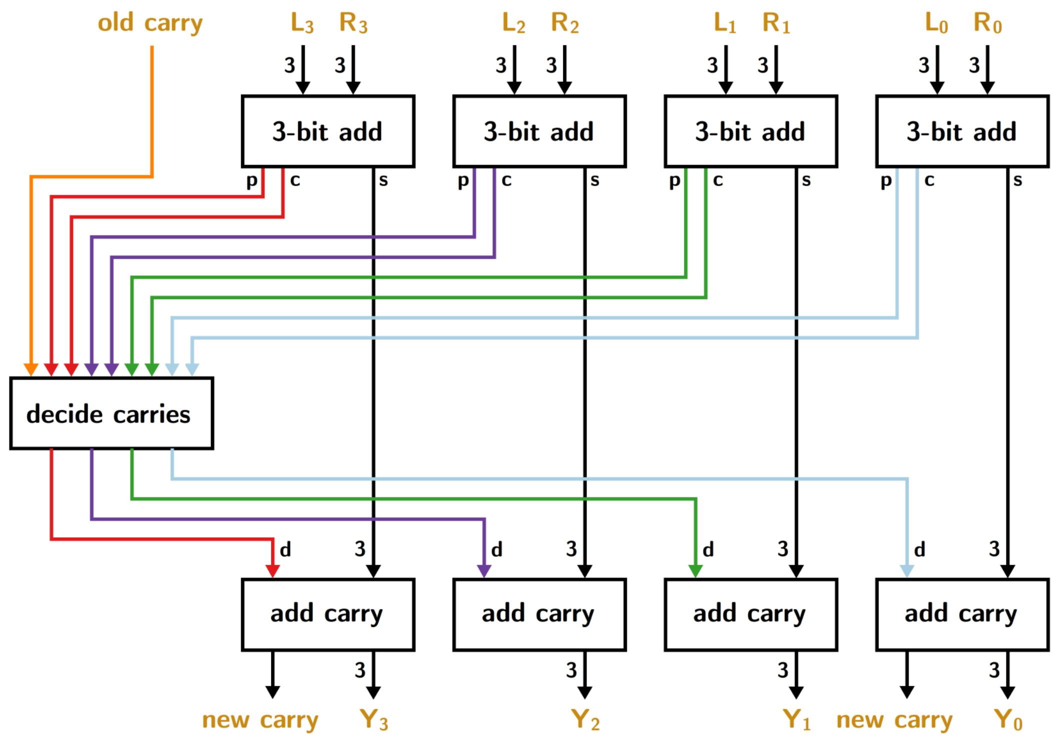


Plate 7. 3-layer carry-skip adder

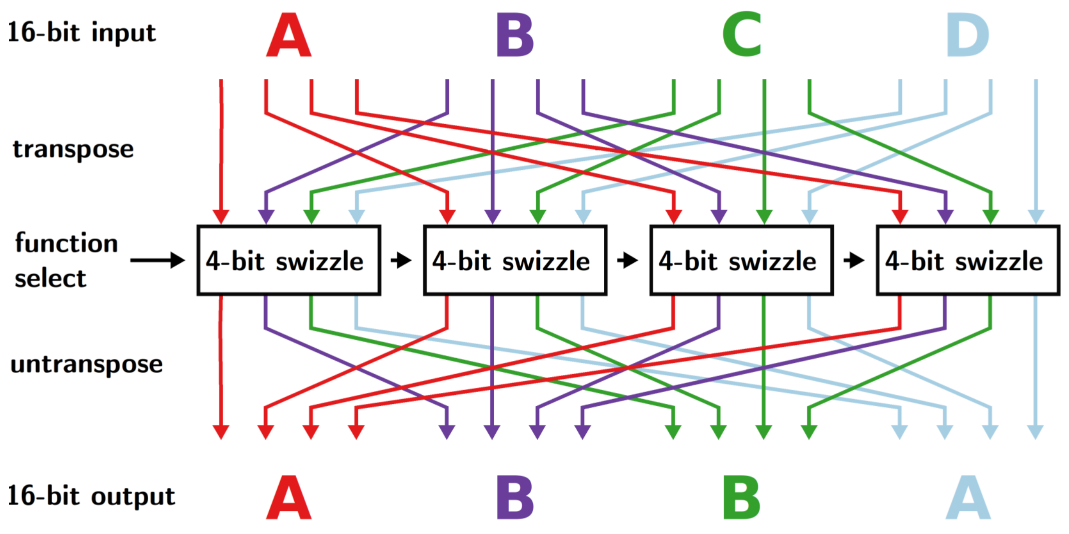


Plate 8. 4 × 4 swizzler

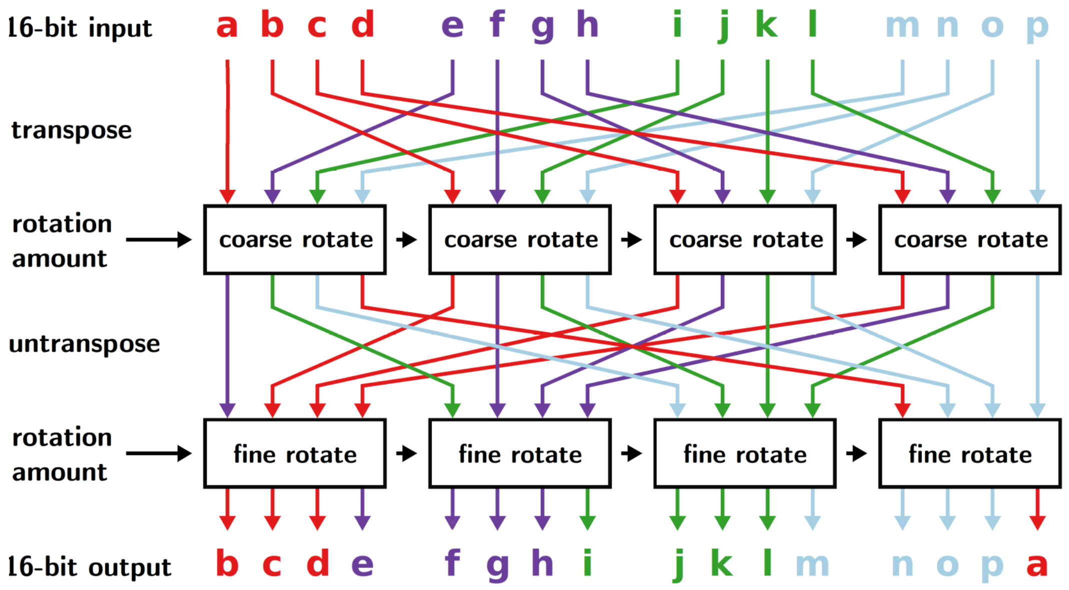


Plate 9. 16-bit logarithmic shifter

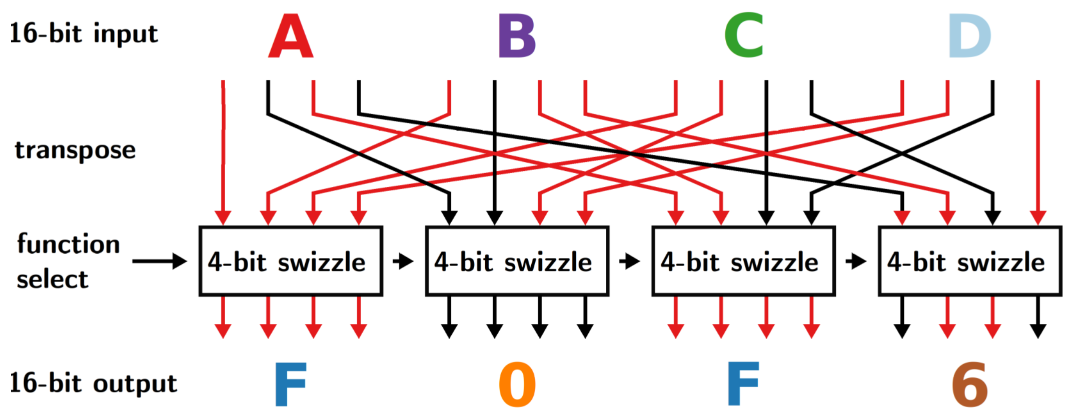


Plate 10. 4 × 4 semi-swizzler

in	out	in	out
0	2	8	a
1	e	9	f
2	b	a	7
3	c	b	4
4	9	c	6
5	1	d	0
6	8	e	d
7	3	f	5

Plate 11. 4-bit S-box

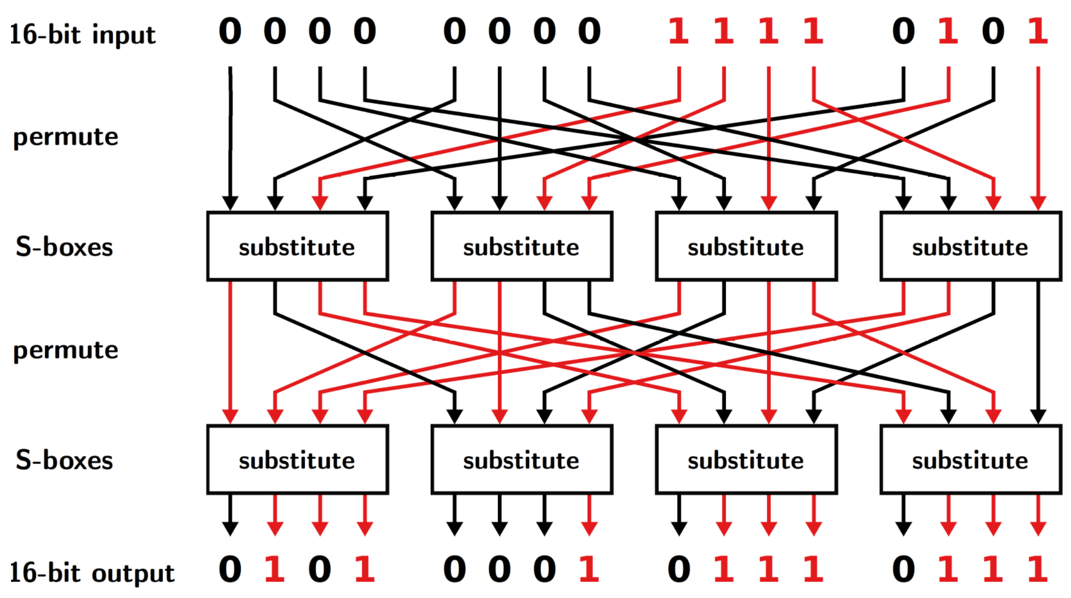


Plate 12. 16-bit substitution-permutation network

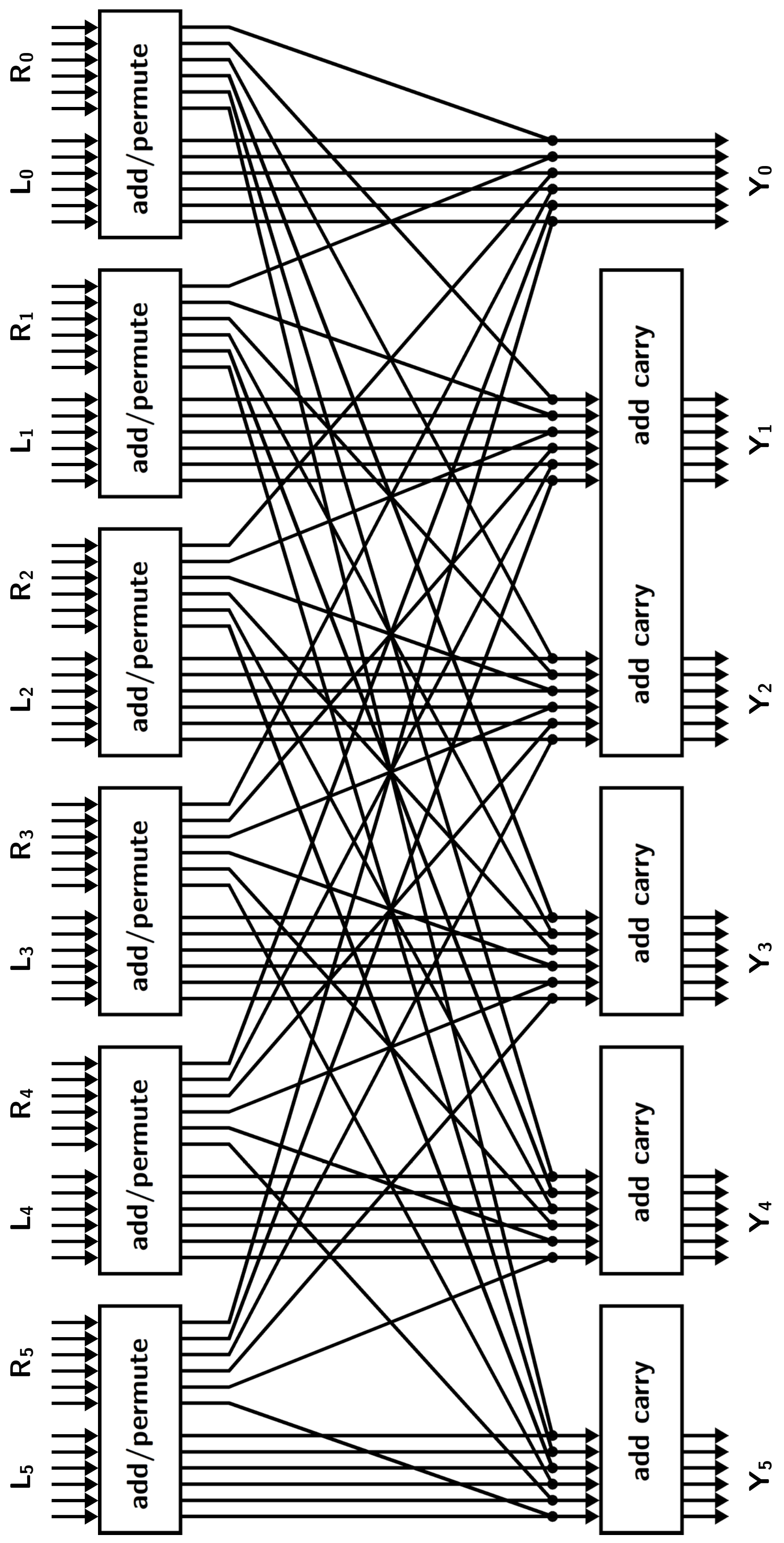


Plate 13. Data signals for a 36-bit, 2-layer ALU

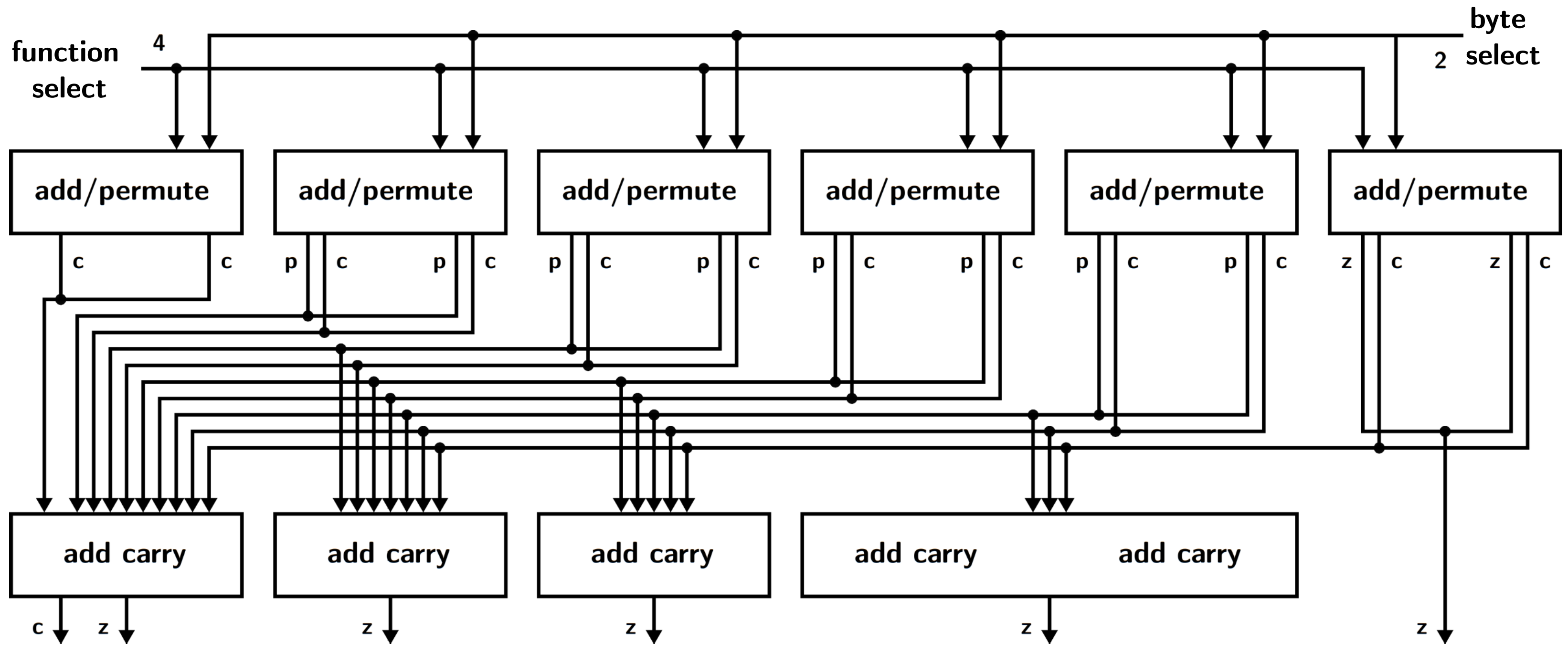


Plate 14. Control signals for a 36-bit, 2-layer ALU

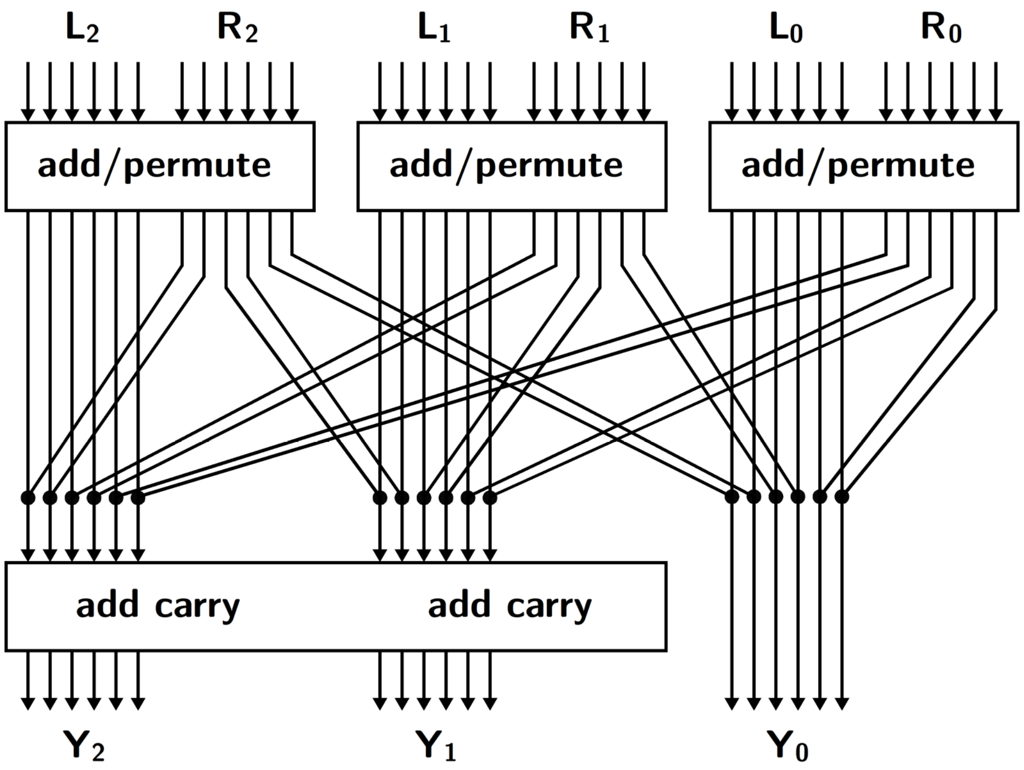


Plate 15. Data signals for an 18-bit ALU

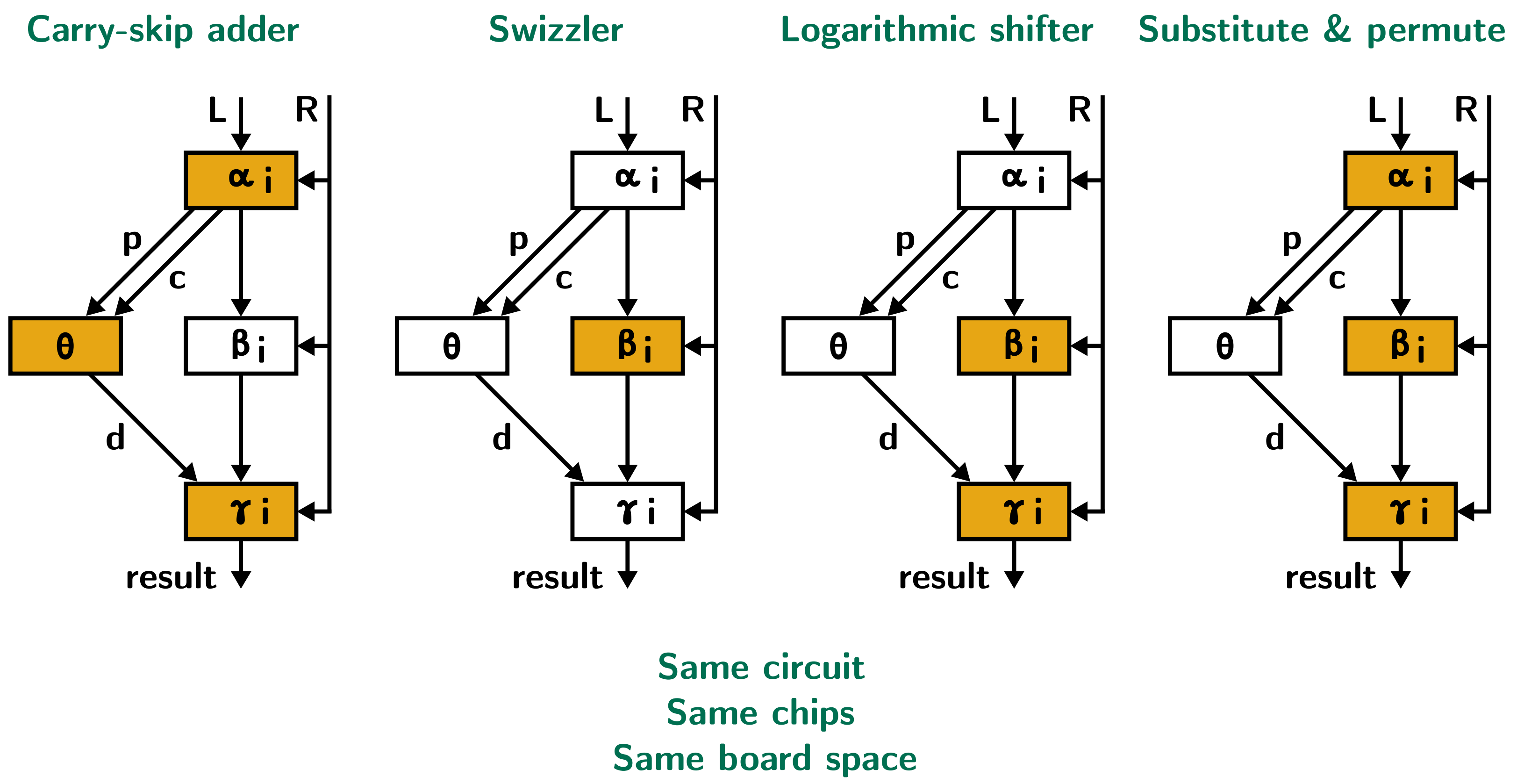


Plate 16. Superposition of major components of 3-layer, 36-bit ALU

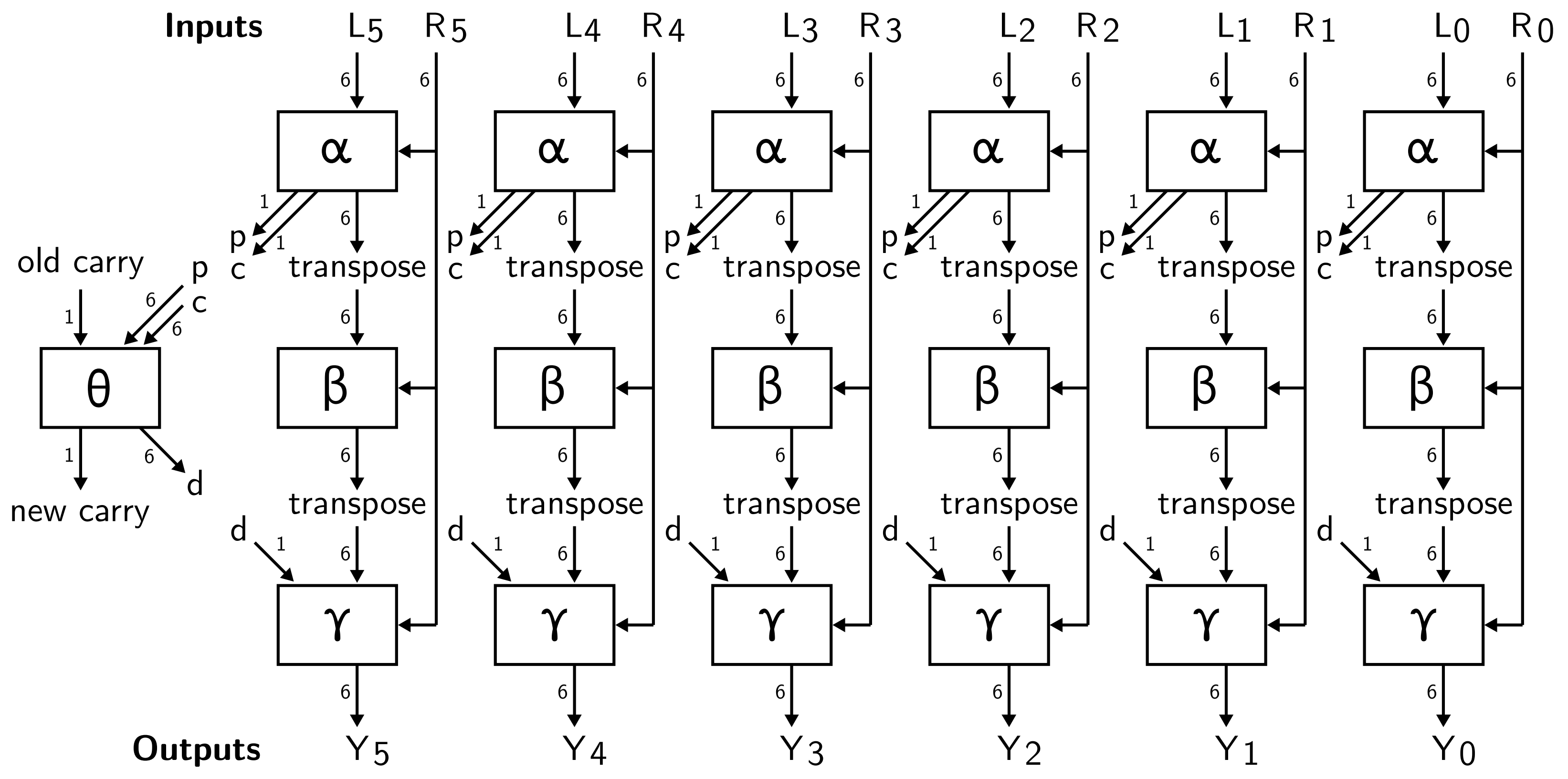


Plate 17. Block diagram of 36-bit ALU

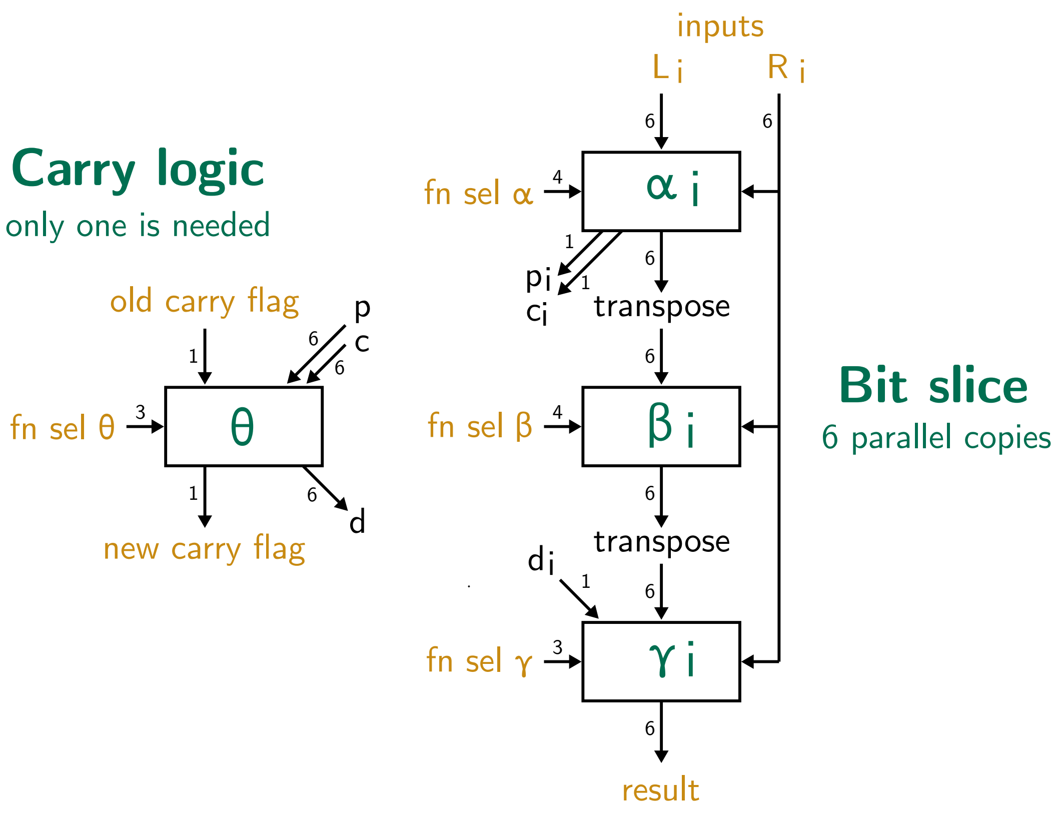
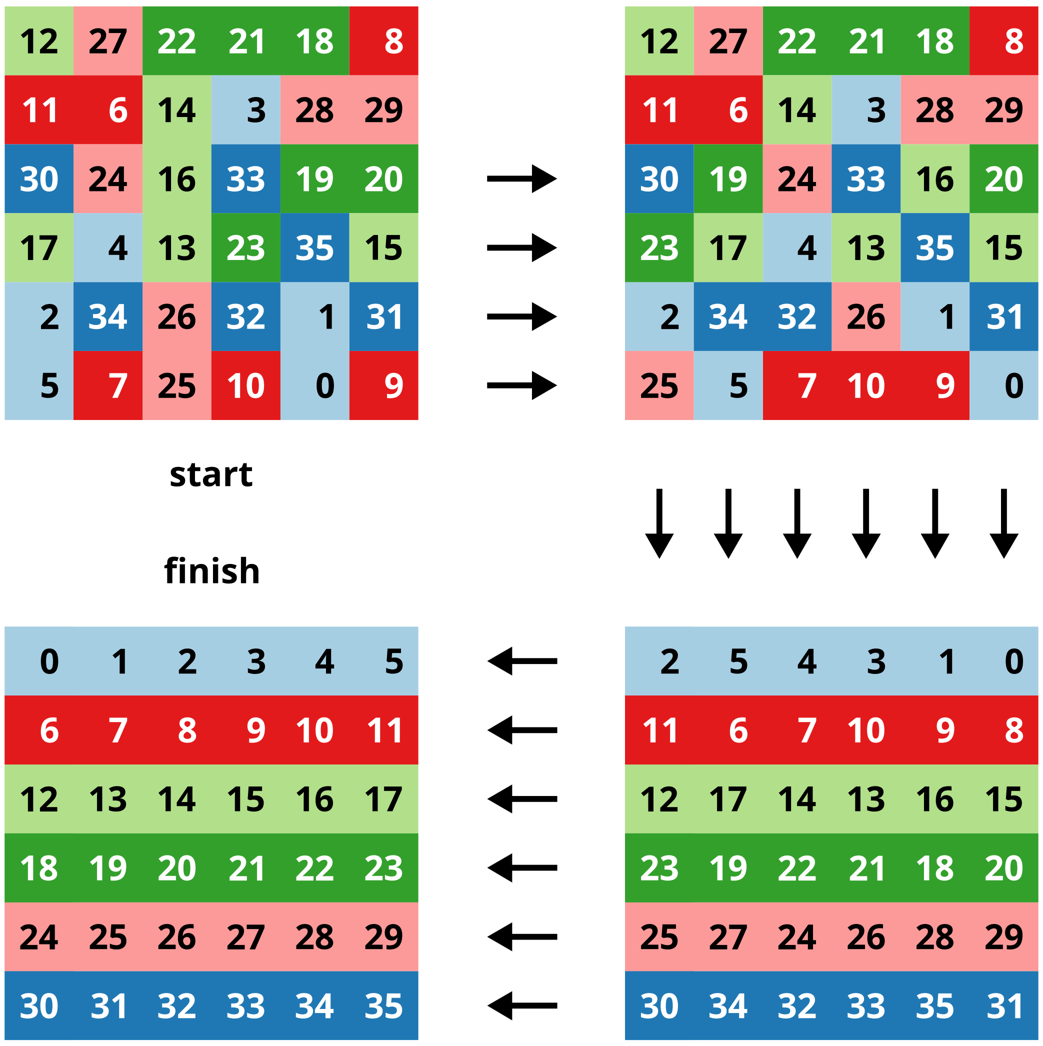


Plate 18. Bit slice and carry hardware for a 36-bit, 3-layer ALU



12	27	22	21	18	8
11	6	14	3	28	29
30	24	16	33	19	20
17	4	13	23	35	15
2	34	26	32	1	31
5	7	25	10	0	9

12	27	22	21	18	8
11	6	14	3	28	29
30	19	24	33	16	20
23	17	4	13	35	15
2	34	32	26	1	31
25	5	7	10	9	0

2	5	4	3	1	0
11	6	7	10	9	8
12	17	14	13	16	15
23	19	22	21	18	20
25	27	24	26	28	29
30	34	32	33	35	31

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

Plate 19. Decomposition of a random 36-bit permutation

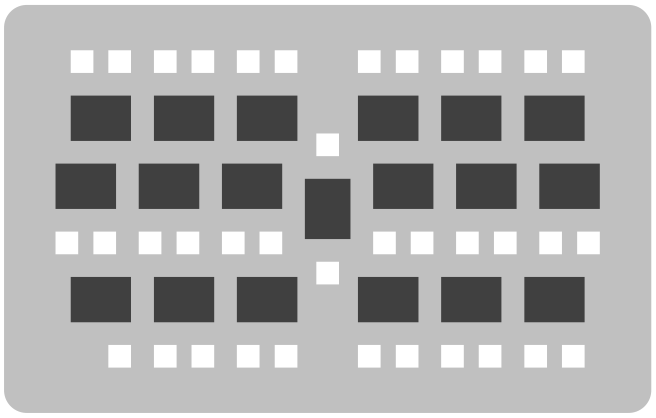


Plate 20. Size comparison of 3-layer, 36-bit ALU with bank card

Table 1. Suggested operations for 36-bit, 2-layer ALU

slot	normal operation	transposing operation
0	unsigned add	unsigned shift left I
1	signed add	unsigned shift left II
2	mixed add	unsigned shift right I
3	unsigned subtract	unsigned shift right II
4	signed subtract	signed shift left I
5	mixed subtract	signed shift left II
6	reverse mixed subtract	signed shift right I
7	XOR	signed shift right II
8	AND	rotate left I
9	OR	rotate left II
a	short multiply I	short multiply II
b	permutations	permutations
c	unary subword ops	S-box
d	(NAND)	inverse S-box
e	(AND NOT)	swizzles
f	(OR NOT)	(transposing XOR)

Table 2. SRAM sizes for 36-bit, 3-layer ALU

SRAM	Mibit	geometry
α_5	1	128k × 8
γ_5	1	128k × 8
rest	½	64k × 8

Table 3. SRAM input bit assignments

layer	tribble	bit	description
α	all	0–5	L input
α	all	6–11	R input
α	all	12–15	function select α
α	5	16	function select for widened $α_5$
β	all	0–5	transposed input from α
β	all	6–11	R input
β	all	12–15	function select β
γ	all	0–5	untransposed input from β
γ	all	6–11	R input
γ	all	12	tribble carry decision from θ
γ	all	13–15	function select γ
γ	5	16	extended sign into widened γ5
θ	n/a	0–5	carry signals from α
θ	n/a	6–11	propagate signals from α
θ	n/a	12	old carry flag from control unit
θ	n/a	13–15	function select θ

Table 4. SRAM output bit assignments

layer	tribble	bit	description
α	all	0–5	tribble to β layer
α	0–5	6	propagate for θ
α	0–5	7	carry for θ
α	5	7*	extended sign for γ_5.+ extra input
β	all	0–5	tribble to γ layer
β	all	6	- unallocated -
β	all	7	- unallocated -
γ	all	0–5	ALU output
γ	5	5*	bit 35 for range check
γ	5	6	bit 36 for range check
γ	all	7	tribble zero detect
θ	n/a	0–5	carry signals for γ
θ	n/a	6	OR of all carry flags from α
θ	n/a	7	OR of all propagate flags from α

***not a conflict**

Table 5. Opcode summary for 36-bit, 3-layer ALU

Bitwise boolean

IGF	ignorant false
IGT	ignorant true
XL	exactly L
XR	exactly R
NL	not L
NR	not R
AND	and
NAND	nand
OR	or
NOR	nor
XOR	exclusive or
XNOR	exclusive nor
LANR	L and not R
RANL	R and not L
LONR	L or not R
RONL	R or not L

Additive

A	add
S	subtract
RS	reverse subtract
AC	add with carry
SC	subtract with carry
RSC	reverse subtract with carry
INC	increment
DEC	decrement
NEG	negate

Mix

MIX	key-dependent S-box mix
XIM	key-dependent S-box unmix

Shift and rotate

ASL	arithmetic shift left
ASR	arithmetic shift right
LSL	logical shift left
LSR	logical shift right
ROL	rotate left

Bit scan

LEAD	zero all tribbles right of any < 6
TRAIL	zero all tribbles left of any < 6
NUDGE	adjust pointer alignment

Bit permute

PIT	permute inside tribbles
PAT	permute across tribbles

Compare

MAX	maximum
MIN	minimum
BOUND	trap if R < 0 or R ≥ L
CMP	compare and set flags

Multiply

ML	tribble multiply low result
MH	tribble multiply high result
MHNS	as above, without left shift
DSL	shift tribble across registers
COPY	copy tribble to all positions

Table 6. Additive opcodes

mnemonic	# of	α	β	γ	θ	arg	description
A	8	+	=	+	+	R	add
S	8	−	=	+	+	R	subtract
RS	8	m	=	+	+	R	reverse subtract
AC	8	+	=	+	c	R	add with carry
SC	8	−	=	+	c	R	subtract with carry
RSC	8	m	=	+	c	R	reverse subtract with carry
INC	0	+	=	+	+	R = 1	increment
DEC	0	−	=	+	+	R = 1	decrement
NEG	0	m	=	+	+	R = 0	negate

Table 7. Bitwise boolean opcodes

mnemonic	# of	α	β	γ	θ	arg	description
IGF	1	=	=	k	1	R = 0	ignorant false
IGT	1	=	=	?	1	R = −1	ignorant true
XL	1	=	=	?	0	R	exactly L
XR	1	=	=	?	1	R	exactly R
NL	1	=	!	+	0	R	not L
NR	1	=	=	^	0	L = −1	not R
AND	1	&	=	+	0	R	and
NAND	1	&	!	+	0	R	nand
OR	1	¦	=	+	0	R	or
NOR	1	¦	!	+	0	R	nor
XOR	1	=	=	^	0	R	exclusive or
XNOR	1	=	!	^	0	R	exclusive nor
LANR	1	¦	=	^	0	R	L and not R
RANL	1	&	=	^	0	R	R and not L
LONR	1	&	!	^	0	R	L or not R*
RONL	1	¦	!	^	0	R	R or not L

*also known as R implies L

Table 8. Compare opcodes

mnemonic	# of	α	β	γ	θ	arg	description
MAX	8	=	=	?	<	R	maximum
MIN	8	=	=	?	>	R	minimum
BOUND	2	−	=	+	+	R	trap if $R < 0$ or $R \geq L$
CMP	4	−	=	+	+	R	compare and set flags

Table 9. Shift and rotate opcodes

mnemonic	# of	α	β	γ	θ	arg	description
ASL	2	o	{	ℓ	0	{R}	arithmetic shift left*
ASR	1	=	}	r	1	{R}	unsigned arithmetic shift right*
ASR	1	r	}	r	1	{R}	signed arithmetic shift right*
LSL	1	=	L	ℓ	0	{R}	logical shift left
LSR	1	=	R	r	1	{R}	logical shift right
ROL	1	=	ℓ	ℓ	0	{R}	rotate L left

*unusual specification; see Section 4.5

Table 10. Multiply opcodes

mnemonic	# of	α	β	γ	θ	arg	description
ML	1	x	=	+	0	R	L × R low tribble result
MH	1	X	*	+	0	R	L × R high tribble result
MHNS	1	X	=	+	0	R	as above without left shift
DSL	1	*	*	+	0	R	unsigned shift R_5 into L_0
COPY	1	=	z	+	0	{R}	6 copies of R^{th} tribble of L

Table 11. Bit scan opcodes

mnemonic	# of	α	β	γ	θ	arg	description
LEAD	1	=	=	k	/	{6}	zero all tribbles right of any < 6
TRAIL	1	=	=	k	\	{6}	zero all tribbles left of any < 6
NUDGE	1	n	=	?	/	R	adjust pointer alignment

Table 12. Bit permute opcodes

mnemonic	# of	α	β	γ	θ	arg	description
PIT	1	=	=	p	0	R	permute inside tribbles
PAT	1	=	p	+	0	R	permute across tribbles

Table 13. Mix opcodes

mnemonic	# of	α	β	γ	θ	arg	description
MIX	1	s	s	s	0	key	key-dependent S-box mix
XIM	1	S	S	S	0	key	key-dependent S-box unmix

Table 14. α operations

slot	name	p & c	output to β
0	=	m	L
1	+	c	L + R
2	−	c	L − R
3	m	c	R − L
4	&		L AND R
5	¦		L OR R
6	x		least sig. tribble of unsigned L × R
7	X	r	most sig. tribble of unsigned L × R
8	*	r	check L_5 for overflow & replace with R_5
9	o	r	L (range check for left shifts)
a	s		R^{th} S-box of L
b	S		R^{th} inverse S-box of L
c	u		R(L) with R being a unary function
d	w		stacked unary word operations
e	n	n	nudge pointer L to align like R
f	r		L with $α_5$ sign bit extended for right shift

	propagate and carry output uses
c	propagate and carry for add and subtract
m	magnitude comparison: L < R, L = R, L > R
n	right tribble is nonzero; maybe start bit
r	signed and unsigned left shift overflow

Table 11. 15. α operations dealing with signedness

Only α_5 uses these variants.

slot	name	sign	p & c	output to β
10	=	u u	m	L
11	+	u u	c	L + R
12	−	u u	c	L − R
13	m	u u	c	R − L
14	=	u s	m	L
15	+	u s	c	L + R
16	−	u s	c	L − R
17	m	u s	c	R − L
18	=	s u	m	L
19	+	s u	c	L + R
1a	−	s u	c	L − R
1b	m	s u	c	R − L
1c	=	s s	m	L
1d	+	s s	c	L + R
1e	−	s s	c	L − R
1f	m	s s	c	R − L

	propagate and carry output uses
c	propagate and carry for add and subtract
m	magnitude comparison: L < R, L = R, L > R

Table 16. Propagate and carry encodings from α

p	c	add & subtract	compare	left shift
0	0	$d_i = 0$	L = R	no overflow
0	1	$d_i = 1$	L < R	overflow if result unsigned
1	0	$d_i = d_{i-1}$	L > R	overflow if result signed
1	1	$\forall i, d_i = 0$*	not used	overflow regardless

*only used for absolute value stacked unary

Table 17. β operations

slot	name	operation implemented
0	=	α
1	ℓ	rotate α left by R bits
2	L	shift α left by R bits
3	R	shift α right by R bits
4	{	shift α left by R bits preserving sign
5	}	shift α right by R bits copying sign
6	s	R^{th} S-box of α^T
7	S	R^{th} inverse S-box of α^T
8	w	stacked unary word operations
9	!	NOT α
a	^	α^T XOR R (reminder: R is not transposed)
b	*	rotate α one tribble left (for multiplication)
c	p	permute across tribbles
d	z	frequently used swizzle operations
e		- unallocated -
f		- unallocated -

Table 18. γ operations

slot	carry	name	ALU output
0		?	mux: R if carry set, else β
1		+	β + carry decision from θ
2		k	kill: 0 if carry set, else β
3		w	stacked unary word operations
4	0	s	R^{th} S-box of β
4	1	S	R^{th} inverse S-box of β
5	0	u	R(β) with R being a unary function
5	1	^	β XOR R
6	0	ℓ	rotate β left by R bits
6	1	p	permute within tribbles
7	0		- unallocated -
7	1		- unallocated -

Table 19. θ operations

All slots compute θ_6 and θ_7 as stated in Table 4.

slot	name	carry decision output to γ
0	+	carry decision for standalone addition
1	c	carry decision for addition including old carry
2	<	111111_2 if L < R, else 000000_2
3	>	111111_2 if L > R, else 000000_2
4	\	∀ i > 0, set d_{i+1} per table below; $d_0 = 0$
5	/	∀ i < 5, set d_{i-1} per table below; $d_5 = 0$
6	0	000000_2
7	1	111111_2

p_i	c_i	adjacent carry decision for \ and /
0	0	d_i
0	1	1
1	0	0
1	1	NOT d_i
1	1	replicate d_0 (0) or d_5 (\) to all d_i during output

Table 20. Arithmetic overflow conditions

operation	result type	overflow occurred iff
add, subtract	unsigned	γ_{36} is set
add, subtract	signed	$\gamma_{35} \neq \gamma_{36}$
left arithmetic shift	unsigned	θ_6 is set
left arithmetic shift	signed	θ_5 is set

Table 21. α.u and γ.u unary operations

slot	operation
0	000000_2
1	000001_2
2	111111_2
3	identity
4	bitwise NOT
5	000001_2 if tribble = 0 else 000000_2
6	000001_2 if tribble ≠ 0 else 000000_2
7	111111_2 if tribble = 0 else 000000_2
8	111111_2 if tribble ≠ 0 else 000000_2
9	count zeros in tribble
10	count ones in tribble
11	count leading zeros in tribble
12	count trailing zeros in tribble
13	count leading ones in tribble
14	count trailing ones in tribble
15	increment tribble with wrap
16	decrement tribble with wrap
17	$\lfloor (2^{36} - 1) \div \max(\{\text{tribble}\}, 1) \rfloor$
	- unallocated -
63	highest available slot

Table 22. Stacked unary operations

These use α, β, γ, and θ simultaneously.

slot	θ	operation
0	\	negate (with overflow check?)
1	\	{0}
2	\	identity
3	+	{0} if negative else {2}
4	>	{0} if negative, {1} if zero, {2} if positive
5	+	$\lvert x\rvert$, only works when $\lvert x\rvert < 2^{30}$ (with overflow check?)
6	+	unsigned integer to left shift control word
7	+	unsigned integer to right shift control word
8	+	unsigned integer to left rotate control word
9	+	unsigned integer to right rotate control word
10	+	tribble summation: 6×3-bit → 3×4-bit
11	+	tribble summation: 3×4-bit → 2×5-bit
12	+	reverse bits
13	/	parity, result in 2^0 position only
14	\	increment word modulo 2^{36}
15	/	increment bit-reversed word modulo 2^{36}
16	\	decrement word modulo 2^{36}
17	/	decrement bit-reversed word modulo 2^{36}
18	/	negate bit-reversed word (no overflow check)
19		- unallocated -
63		highest available slot

Table 23. Single-instruction β and γ tribble permutations

012345	123450	234501	345012	450123	501234	rotations on identity
432105	321054	210543	105432	054321	543210	reflections of the above
102345	210345	312045	412305	512340		pair swaps
021345	032145	042315	052341	013245		
014325	015342	012435	012543	012354		
452301	014523	230145				other move pairs together
024135	031425					2 × 3 and 3 × 2 transposes
031524	142035	253140	304251	415302	520413	count out of circle by 3s
035124	140253	251304	302415	413520	524031	count out by 3s backward
102354	315042	542310	513240			other 2-pair rotations
021435	034125	043215				
103254	240513	453201	521430	534120		other 3-pair rotations
120534	201453					symmetric half rotations
034512	035421	105243	130542	254310	510432	needed to compose all 6!

Table 24. Compact set of tribble permutations for two instructions

023541	031542	032451	045231	054132
054321	103245	120453	140325	204531
235041	240351	253401	305421	324051
341205	342501	350241	403125	403251
423015	425301	430521	435102	452031
502341	520431	534120	534201	543021

Table 25. Frequently used swizzle operations for β.z

slot	operation
0	6 copies of tribble 0
1	6 copies of tribble 1
2	6 copies of tribble 2
3	6 copies of tribble 3
4	6 copies of tribble 4
5	6 copies of tribble 5
	- unallocated -
63	highest available slot

Table 26. 2-cycle leading and trailing bit manipulation

	example: find 1	normal output		inverted output	
operation	normal output	find 0	find 1	find 0	find 1
invert bit	0101100 → 0101000	x OR y	w AND x	x NOR y	w NAND x
isolate bit	0101100 → 0000100	x RANL y	w RANL x	x LONR y	w LONR x
fill to bit	0101100 → 0101111	x AND y	w OR x	x NAND y	w NOR x
mask until bit	0101100 → 0000011	x LANR y	w LANR x	x RONL y	w RONL x
mask thru bit	0101100 → 0000111	x XOR y	w XOR x	x XNOR y	w XNOR x

These formulas are for trailing bit manipulation. The input is x, $w = (x - 1) \bmod 2^{36}$, $y = (x + 1) \bmod 2^{36}$. Use the stacked unary operations for w and y to avoid false positive overflow errors from the INC and DEC opcodes. For leading bit manipulation in two cycles, use the bit-reversed stacked unary operations for w and y.

Listing 1. Assembly code for 36-bit unsigned multiplication

72-bit result d : c = a * b requires 53 CPU cycles.

```
t = b copy {5}            t = b copy {3}            t = b copy {1}
; m not used yet          m = a mhns t              m = a mhns t
c = a mhns t              c = c add m               c = c add m
d = d adc 0               d = d adc 0               d = d adc 0
d = d dsl c               d = d dsl c               d = d dsl c
c = c lsl {6}             c = c lsl {6}             c = c lsl {6}
m = a ml t                m = a ml t                m = a ml t
c = c add m               c = c add m               c = c add m
d = d adc 0               d = d adc 0               d = d adc 0

t = b copy {4}            t = b copy {2}            t = b copy {0}
m = a mhns t              m = a mhns t              m = a mhns t
c = c add m               c = c add m               c = c add m
d = d adc 0               d = d adc 0               d = d adc 0
d = d dsl c               d = d dsl c               d = d dsl c
c = c lsl {6}             c = c lsl {6}             c = c lsl {6}
m = a ml t                m = a ml t                m = a ml t
c = c add m               c = c add m               c = c add m
d = d adc 0               d = d adc 0               d = d adc 0
```

Listing 2. Python 3 specification of S-boxes for the MIX opcode

```
#!/usr/bin/python3

def perm():                    # obtain a permutation of 64 objects
    avail = list(range(64))
    done = []
    while avail:
        perm.fract *= len(avail)
        rand = perm.fract >> root_bits
        perm.fract &= root_mask
        done.append(avail[rand])
        del avail[rand]
    return tuple(done)

def boxes():                   # obtain 64 permutations for 1 RAM
    return tuple(perm() for i in range(64))

def row():                     # obtain perms for a row of 6 RAMs
    return tuple(boxes() for i in range(6))

def run():                     # obtain perms for 3 rows of RAMs
    return row(), row(), row()

root_bits = 341000             # needed = 18 * 64 * (64!).bit_length()
root_mask = (1 << root_bits) - 1
half_bits = 2 * root_bits      # must be even
half = 1 << half_bits - 1      # 1/2 in this radix
root = 3 << root_bits - 2      # initial estimate for (1/2)**(1/2)

print("Computing %s bits of sqrt(2)/2 " % root_bits, end="")
while abs(half - root * root) > 2 * root:
    root = (root + half // root) >> 1
    print(end = ".", flush = True)
print(" done!")
perm.fract = root              # initial irrational to extract from

v = run()                      # v is the output of this run
print(v[2][5][63])             # show last permutation as a check
```