A Solder-Defined Computer Architecture for Backdoor and Malware Resistance

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. A Solder-Defined Computer Architecture for Backdoor and Malware Resistance.

This research is about securing control of those devices we most depend on for integrity and confidentiality. An emerging concern is that complex integrated circuits may be subject to design defects or backdoors, and measures for inspection and audit of these chips are neither supported nor scalable. One approach for providing a "supply chain firewall" may be to forego use of such components, and instead to build CPUs and other complex logic from simple, generic components. This work investigates capability and speed limits of fusing open-source hardware methodologies with maker-scale assembly tools and visible-scale final inspection.

The author has designed, and demonstrated in simulation, a 36-bit arithmetic logic unit (ALU) that uses only static RAM and trivial glue logic chips as components. The propagation delay is 30 ns. To improve performance, this ALU includes operations that reduce need for software loops, such as a hash function for associative arrays, a pseudorandom number generator, and bit permutation primitives. This ALU also converts transparently between signed and unsigned integers with rigorous detection and latching of overflow conditions, so that a single flag check can determine the validity of a lengthy arithmetic sequence.

The chief task of this research is to assess the feasibility and utility of extending the ALU into a transparently functioning, "solder-defined" minicomputer with preemptive multitasking, protected memory, and subsystems for memory and I/O. Computers of its family could be built either on a benchtop with a reflow oven, or on a small surface mount technology assembly line, using components that are readily available as of 2020. A 36-bit architecture is anticipated with a board footprint of approximately 20×20 cm and a target speed of 10 MIPS. Such an architecture, if practical, may be applicable to uses where trust in the hardware's source is essential, and remote exploits cannot be tolerated.

Contents

Tables

Figures

1. Overview

Irregularities in the behavior of computing hardware are a perennial source of software-related defects, including exploitable defects that are regarded as security vulnerabilities. The author groups these irregularities loosely into three categories, summarized in Table 1.

Category I represents ordinary hardware semantics or limits that programmers tend to overlook. These semantics and limits can lead to assumptions of invariance, to which attacks supply counterexamples. For instance, arithmetic results may go out of range. Memory buffers have finite size. Clocks that keep time are not always monotonic. Shifts and division have peculiar, architecture-dependent semantics for unusual operands. Suggested workarounds for Category I irregularities have included increasing the scrutiny done by programmers [31], and increasing the separation between the instructions that programmers write and the silicon executing them [21].

Category II irregularities are anomalies that contradict the documented or understood operation of computing hardware. These are some of the bugs that appear in the news, as well as more primary sources like [3, 4, 8, 12, 14, 22, 23, 25, 29, 32, 34]. They have whimsical names like "hidden God mode," "Meltdown," and "Silent Bob is Silent." These pitfalls can sidestep the most attentive programmer's precautions, and they can surface years after a hardware, operating system, or application platform is no longer supported by its original producer. To first order, vulnerabilities from Category II irregularities cannot be solved through software, but require architectural changes to future generations of VLSI by the supplier.

Category III irregularities are hidden characteristics introduced within a hardware supply chain that do not align with the buyer's security objectives. They can either be active logic that provides a backdoor for an unseen adversary to control a system, a passive mechanism for eavesdropping, or some combination of both [1, 10, 26, 28]. Such vulnerabilities are of particular urgency when national boundaries are crossed [27]. Another model that falls under Category III is

Table 1. Classes of vulnerability-inducing hardware irregularities

Category	I	II	III
Origin	purposeful	unexpected	malicious
Example	arithmetic wrap	RowHammer	hidden backdoor
Software fix?	yes	no	no
VLSI fix?	yes	yes	no
Manufacturing fix?	yes	yes	yes

the outright counterfeiting of critical equipment by unknown manufacturers, such as reported in July 2020 for network switches [20].

Because Category III irregularities are adversarial in origin, there is incentive to place them at points where the buyer can neither inspect nor repair the product, such as within VLSI or inaccessible microcode. In general, these vulnerabilities cannot be addressed satisfactorily either via software or through hopes that a supplier will abstain from inserting them. Instead, the buyer needs to extend some form of control over the manufacturing process and/or the final assembled product.

This proposal's essential target is hardware irregularity Category III, the case where a supply chain for complex logic, such as microprocessors and peripheral controllers, may be tainted by an adversary prior to delivery to a buyer. In such circumstances, computing equipment may contain a backdoor or a data exfiltration mechanism. A drastic approach is advanced: relocate the supply chain from manufacturing technology and processes that an adversary may control to technology and processes that are under the buyer's control. There is a straightforward way to do this: have the buyer manufacture its own complex logic. There is also a backup approach: have the buyer inspect its purchased logic.

Neither of these approaches can place a VLSI foundry at the buyer's disposal. Semiconductor fabs cost billions to build, and a short-term lease would be financially and logistically implausible. What may be practical, however, is to look back 50 years to a time when computers were assembled from simple components by inexpensive tools at millimeter scale. The labor was costly, the dimensions were large, the connections were many, and the unit price and weight were

beyond reach for many uses. But in time since, components and assembly processes have undergone revolutionary change. What computing machinery can be built now, using basic components, inexpensive tools, and millimeter dimensions? And what are the security implications of this machinery?

The central hypothesis of this proposal is that it is possible to build a reproducible minicomputer[1] today that meets the following informally stated supply chain tamper resistance criteria:

Supply chain tamper resistance criteria

1. The computer's electronic components are simple and generic enough, that it would not be practical for an adversary to introduce exploitable defects through the component supply chain.

2. Each electronic component already exists in the marketplace for other uses, and at least two manufacturers currently produce any particular component.

3. The computer can be assembled using current surface-mount practices, using either manual or automated placement and soldering.

4. The assembled computer can be inspected against open-source specifications, resulting in a modest level of confidence that exploitable defects were not injected into the product.

5. The computer can be warranted by the manufacturer to exactly match identified open-source specifications.

6. The computer can be warranted by the manufacturer to be assembled from components supplied by the buyer, or from a specified bill of materials.

In addition to the supply chain tamper resistance criteria, an architectural specification with estimated performance figures is described in the sections that follow. In brief, the proposed architecture claims a 36-bit word size, memory protection, and preemptive multitasking. The planned speed is ten million instructions per second (10 MIPS), and the board footprint is estimated around 20×20 cm. There does not appear to be much margin to improve either metric using currently available components.

1 A definition for *minicomputer* appears on page 6.

In light of the low ceiling on instruction throughput, the design seeks a high yield of computational work as each CPU instruction is executed. There are two prongs to this strategy. First, the relationship between CPU cycles and instructions run is one-to-one. For the design proposed, a 10 MHz clock is equivalent to a 10 MIPS computer. Second, the architecture is designed to minimize the number of instructions needed to accomplish a task. For example, a context switch from running one program to running a different program is simply a write to a register, and takes just one instruction. No registers or flags need to be spilled to RAM in order to switch programs. Another example is the generation of statistically robust pseudorandom numbers, which despite only requiring two instructions per 36-bit word generated, passes all components of a leading suite of random number generator tests [6].

Although buyer control the assembly process is primarily to suppress Category III hardware irregularities (that is, malicious abnormalities), it would be an oversight for the buyer to leave unnecessary exploitable Category I and II irregularities as residual risks. For this reason, the proposed architecture departs from current norms to roll back some of the complexity which has led to Category II surprises, as well as adopts instruction set semantics that simplify management of Category I concerns.

As the central hypothesis of this work is the constructability of a computer with a specified architecture, without any additions to the parts available to assemble the machine from, the test of this hypothesis will be to attempt its construction and report the outcome. The distance in effort between producing a correct simulation and producing a functioning machine is not a large one, but there is a significant difference in how the two endpoints would be perceived. The best value for this project, and the best prospect for its impact, are to bring the working (or not working) system to the dissertation defense. So this is the path that the research will take.

The minicomputer of this proposal would be fast enough to control most systems that physically move: industrial and commercial devices, factory automation, electric grids, wells and pumps, heavy machinery, trains, dams, traffic lights, chemical plants, engines, and turbines. It would also be fast enough for many uses

not involving motion, such as measurement and sensing, peripheral and device controllers, telephony, and even Ethernet switches to medium speeds. It would also permit desktop use writing and editing documents, making spreadsheets, sending and reading email, and writing and building software, although desktop software would need to be specifically written for or adapted to the architecture. The proposed architecture is not small or fast enough for smartphones or video.

For servers, the applicability of the proposed architecture will depend on workload and surrounding components, especially software. A web platform intended for an eight-core CPU and 64 GB of RAM is not within reach of this technology. Even if an application on this scale could be accommodated, the sheer size of the software will often present a larger attack surface than the hardware it runs on. On the other hand, server applications specifically designed to run on and thoughtfully matched to the proposed architecture will run fine. Over the past 50 years, humans have proven adept at making systems fit within computing hardware constraints when sufficient motivation is present.

2. Definitions

This proposal uses terms defined in [17], plus the following definitions for concepts that do not have well-known terms.

Buyer. An authority responsible for the selection, procurement, installation, operation, and security of a computing platform on behalf of a risk owner.

Complex logic. Sequential circuitry of sufficient complexity to be capable of concealing exploitable defects.

Discounted logic. Logic for which a documented risk assessment and other controls validly establish that exploitable defects are unlikely to exist.

Maker-scale assembly tools. Tools and methods for constructing electronics that are within economic reach of most technically qualified builders.

Microcomputer. From convention and [7], a "computer system that utilizes a microprocessor as its central control and arithmetic element."

Minicomputer. A computer where all complex logic, including but not limited to the central processing unit, that is not discounted is solder-defined.

Primary storage. Memory that a CPU accesses via load and store instructions. Primary storage is what people usually mean when they say "RAM."

RAM. Unless otherwise stated, another term for static RAM. In this document, most SRAM is not primary storage, but implements logic or registers.

Solder-defined behavior. Operational behavior that is determined by solder-defined hardware.

Solder-defined hardware. Circuitry that can be assembled from a set of components containing no complex logic, using maker-scale assembly tools.

Supply-chain firewall. A point in a supply chain through which one or more classes of exploitable defects will not readily pass.

3. Research questions

It is canon that VLSI is the key which during the 1980s unlocked all practical computing. The cost efficiencies of the microprocessor and its descendants teach the field that practicality and miniaturization cannot be separated, just as one cannot separate waves from particles or matter from energy. Efforts to exclude semiconductor fabrication from certain stages of computer manufacturing in order to increase transparency and accountability will sacrifice much practicality. But *how much* practicality? Orders of magnitude, if practicality is measured by speed. But is speed the right metric? For video rendering, yes, speed is a critical metric. What about for treating wastewater? Or for tabulating votes? What kinds of applications are compatible with the architecture of this proposal? Which are not? How flexible is the compatibility boundary, and how easy is it to influence which applications may be served and which may not? How good can a near-term "high bar" be with respect to optimizing production cost, attack surface, and supportable applications?

What are the economics of installing a solution that may not just run 1000 times slower, but also cost 1000 times more? And that's just the hardware cost. But consider avoided costs: IBM Security reports in July 2020 that the cost to remedy data breaches of more than 50 million records averages $392 million [16]. Also, there are liabilities around the planet that could be even more serious, as evidenced by the August 4 explosion of 2700 tons of ammonium nitrate in Beirut [13]. The cause of this incident remains under investigation and may not relate to computing, but potential for other atomic weapon-scale explosions which may relate undoubtedly exists. Air handlers for biosafety level 4 laboratories are another application where a $1 ARM processor may not be the architecture of choice. The proposed work won't yield an actuarial model for these questions, but it can estimate baseline costs for models that follow.

Other questions stem from whether and how the proposed architecture is going to work. What are the essential system blocks just above the RAM level, and how can they interoperate? What table-based mechanisms exist that can reduce the number of cycles needed for multiplication, division, single-precision

floating arithmetic, permutations, pseudorandom number generation, hashing for associative arrays, and other necessary (but often performance-killing) operations? Is it effective to include special hardware, such as fast multipliers, for some of these operations when a significant increase in component count would result? Can practical encryption be implemented on a network-connected register machine with a word size much smaller than 128 bits?

What are the limits to cycle time and pipelining on SRAM-based CPUs? Will it be possible to achieve, for example, 10 MIPS throughput on a system if the basic logic gate has a best-case delay of 7 ns, assuming such features as protected memory and preemptive multitasking are mandatory? And if that throughput is 10 MIPS, can this be 10 MIPS within the same thread, or will data hazards necessitate simultaneous multithreading—that is, interleaving instructions from two independent threads—to achieve this speed?

If the CPU's speed could be drastically increased by sacrificing security-related features such as overrange checking and protected memory, can we arrive at the same net usefulness on smaller, cheaper hardware by employing a meticulously scrutinized interpreter for a language reminiscent of Python, Java, or (more likely) Lua?

Many scientific discoveries happen serendipitously before research questions can be formalized. Radio astronomy is a frequently cited example, born of an engineer who pondered interference to transatlantic radio conversations [19]. The same has been true of preliminary research leading to this proposal. As an example, the author assumed that a 32-bit minicomputer would be a good starting point for migrating applications to solder-defined machines. But after a few iterations of 32-bit ALU designs, it became evident that SRAM exhibits a natural optimality for a 36-bit word size. Who knew? And by apparent sheer coincidence, other system components such as primary storage turn out to be as readily obtained in the market for 36 bits as for 32 bits. That 36-bit words appear to be a preferred implementation is an important scientific result with many practical consequences and advantages, but no one embarked to make such a discovery. No questions were ever posed on this subject—and had they been, they would probably have made biased assumptions and found in favor of 32 bits. What happened

instead is, the author set out to do a project, and its output was new scientific understanding. The discovery was intentional, no more accidental than when a drift net catches fish, but the process was not guided by questions. Even the general subject matter of the result could not have been anticipated, but what was learned turns out to be important. The author set out on a fishing expedition to catch something unknown, much like simulated annealing and self-organizing maps search wanderingly for optimal configurations and informative patterns. It's all valid and efficacious science.

The discovery that 36-bit words are a good specification was not an isolated event. It was likewise observed that repeated design cycles of SRAM ALUs tend to "pull" their architecture into very regular, dense networks like that drawn in Figure 3 on page 19. These networks offer a high ratio of computational expressivity to component count, and their proclivities stem from branches of mathematics, network science, and combinatorics unfamiliar to the author and perhaps not widely explored at all. But the observation is noteworthy and consequential. Likewise, the SRAM-derived logarithmic shifter is believed to be novel, and its discovery was accidental. Earlier designs for the ALU involved shifters that used 8-to-1 multiplexer circuits and/or exponentially wasteful SRAM input configurations. They offered lackluster, but not worrisome, performance. But elsewhere in the ALU, 32 bits at the time, a mechanism was sought for quickly changing the endianness (byte order) of a word, as that operation was one of several where SRAM ALUs were suspected to bolster performance. The endianness problem's solution came out isometric to what is now the first layer of the logarithmic shifter of Figure 2 on page 18, and the second layer was already drawn below it as a mechanism for fast permutations of 4-bit subwords. Seen together, these two provisions explained how to shift or rotate any number of bit positions in one machine cycle, while employing very few components. There are more instances like this, where contributing discoveries were made, not only without questions being asked first, but possibly *because* carefully targeted questions were never asked.

It would be very exciting if similar gains emerge in the proposed CPU, memory subsystem, and I/O subsystem. The author has long dreamed of CPUs made from connected memories where the datapath is not a loop, but computa-

tions proceed instead by allowing data to reciprocate, as if regulated by a system of second-order differential equations, among a small number of RAMs with their data lines sharing some kind of common bus. Alternatively, perhaps a bit-serial architecture can implement minicomputers with few components and yet somehow be fast enough to be contributing. No ideas have emerged as to how to implement either vision, so the "architect's conception" of Figure 4 on page 29 is an SRAM extension of a very traditional RISC pipeline. It could change. Project-driven research provides one type of incubator where the requisite leaps sometimes appear overnight, as they already have in this work to date.

In summary, questions from many directions attach to the proposed work, in part because no recent precedent exists for practical solder-defined architectures. This research will contribute knowledge for several of these questions, although which will be most contributing may turn out to be a surprise. Moreover, the thrust of this study to produce a high-quality, reproducible prototype is likely to spark insights into questions the author has yet to consider.

4. Logic family selection

The logic family to be used for this research is assembled from small asynchronous static RAMs and fast glue logic. To help understand why this family is favored by this proposal, some alternatives are described.

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 employed at scale [11].

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 [30].

Nanoscale vacuum-channel transistors may someday work like vacuum tubes, but at present are only theoretical [15].

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 and board capacitance [24].

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

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.

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. Some current 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.

In addition to the above one-bit-wide logic families, memory devices can implement elaborate combinational logic on small words by using preloaded truth tables. Available memory families include the following.

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 a parallel interface is suitable for combinational logic, offering speeds to 55 ns. Storage density is not as extraordinary as NAND flash, but 128M $\times$ 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 re-

tention is typically 10 to 20 years, so these devices must be periodically refreshed by means of additional components or temporary removal from sockets. 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. Such a controller may be useful for loading microcode into SRAM-based CPUs. Here again, a servicing mechanism will 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 [25, 32], as well as to cosmic ray and package decay soft errors. The main upside to DRAM is that an oversupply resulting from strong demand makes it disproportionately inexpensive compared to better memory.

Programmable logic devices, or **PLDs**, and **field programmable gate arrays**, or **FPGAs**, are specifically designed for conveniently implementing complex logic. Their drawbacks are that they are not inspectable, are not auditable, not fungible, ship with undocumented microcode and potentially other state, have a central view of the logic being instantiated, and have a very small number of suppliers controlling the market. They are amazing, affordable products with myriad applications, but they may also be the ultimate delivery vehicle for supply chain backdoors. They are easily reconfigured without leaving visible evidence of tampering.

Static RAM, or **SRAM**, has the parallel interface necessary 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 February 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 primary storage, 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 are eventually selected for computation, the 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 this trust of SRAM for storage is not misplaced, then also using SRAM for computation may not expand the attack surface as much as adding something else instead.

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. As SRAM is 600 times as expensive, factors beyond transistor count influence price. Even so, the price of SRAM has fallen more than 100-fold over the last 40 years.

Asynchronous SRAM is available in 2020 in sizes to about 32 Mibit. 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 interchangeabil-

ity. The JEDEC pinouts are somewhat undermined by manufacturers ensuring they offer non-overlapping—although standardized—package geometries.

The chief drawback of using RAM as a building block is that the actual need is for ROM. The contents of function-computing memory only need altered if the architecture or features change. But 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 advertised parallel EPROM offers a 45 ns access time, contrasted with 10 ns for inexpensive SRAM.

Analog switches have been identified which may be appropriate for delivering firmware into SRAMs at power-up. These include[3] the PI3B3253 from Diodes Inc. and IDT's QS3VH253, and are available with footprints down to 3×2 mm. The capacitance added is of similar order to 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 $\times$ 16 RAM would require roughly two switch ICs on an amortized basis, plus additional logic farther away from where computations are made. Cost varies by model but can approach $0.25 per switch chip as of July 2020. So initialization hardware is projected to add about 50% over the SRAM cost.

3 Analog switch models and manufacturers are mentioned to facilitate comprehension only. The author has tested none, and endorses neither.

5. Arithmetic logic unit

The expressivity and conciseness of static random access memory as a logic family, in distinction from traditional Boolean logic gates, strongly influence the manner in which complex logic should be constructed. Although SRAM arithmetic logic unit (ALU) designs can be informed by designs based on other logic families, they should not be direct transcriptions.

Static RAM-based adders and shifters are two novel contributions of this research. A short explanation of these elements in isolation may be helpful to understanding how carry-skip adders and logarithmic shifters can be superposed and used in practical ALUs.

Figure 1 presents a simple case of a carry-skip adder. The word size is 12 bits, organized as four subwords of three bits. Each of the nine boxes shown represents a small SRAM. The four top RAMs simply add 3-bit integers by table lookup, yielding a 3-bit result and carry (c) bit for each subword. The main problem is handling the rightward propagation of these carry bits without delay accumulating at each stage.

The carry-skip solution is different for SRAM logic architectures than from traditional ones. Familiar carry-skip adders exhibit fast leftward carry propagation. But in SRAM, the subword carry decisions are solved simultaneously by table lookup. In Figure 1, carry decisions (d) are made for each subword by the lone RAM in the middle. Two inputs are used from each subword: the familiar carry input, and an input called *propagate* (p) that is true whenever the tentative subword sum is all ones. A subword's propagate bit is true if the final carry output for a subword should be identical to the carry input; that is, an all-ones result will roll to all zeros. Otherwise, the propagate bit is unset, and a subword's tentative carry decision stands as the correct final carry. This middle RAM is also the easiest place to introduce any carry from a previous 12-bit addition.

The bottom row of RAMs in a carry-skip adder simply adds one to each subword, possibly wrapping from seven to zero in this three-bit illustration, based on the supplied carry decisions. A final carry bit is output, usually on the left side. A right-side carry output may accommodate unusual needs.

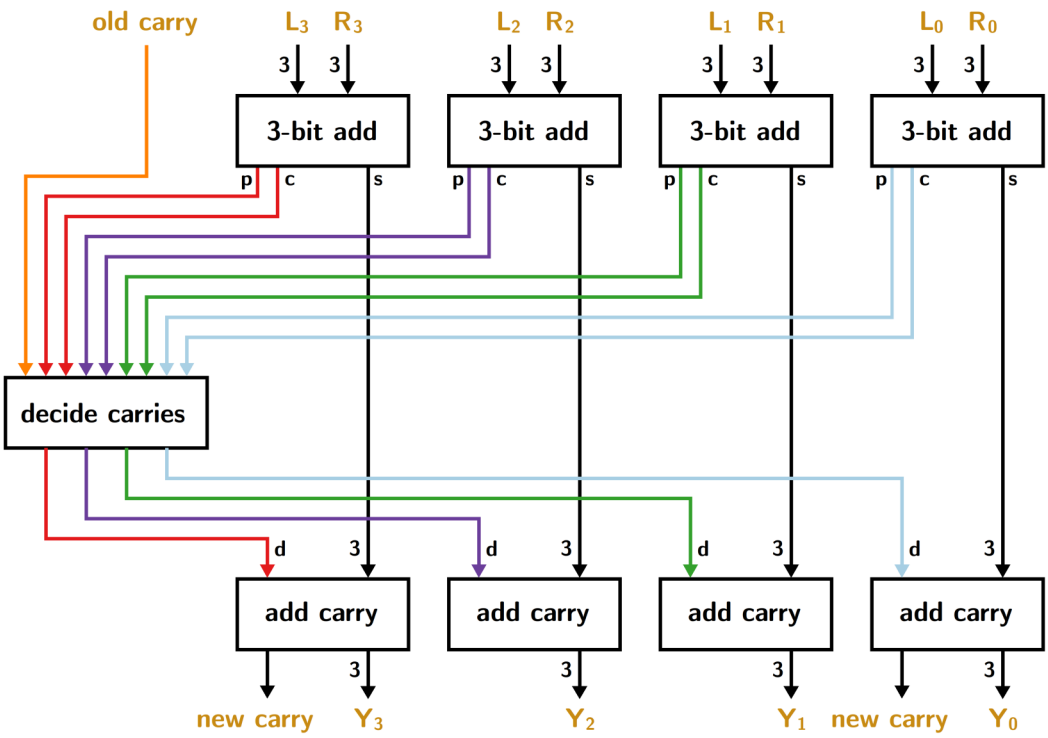


Figure 1. 12-bit carry-skip adder

Figure 2 shows an SRAM implementation of a logarithmic shifter. Its subword striping scheme differs from commonly described barrel shifters that don't need to not split into subwords [35], but its general strategy is similar in principle. In the example drawn, the word size is 16 bits organized as four subwords of four bits. There are two layers of four RAMs each. The first layer does "coarse" rotations, meaning moving bits between subwords. The second layer does "fine" rotations, meaning moving bits within subwords.

The first layer of RAM has its input and output wiring transposed in a self-inverse manner. This allows each RAM in layer 1 to accept one bit from each subword input, and send one bit to each subword output. The bit positions are identified by letters in order to show their migration, and subwords can be tracked by their color. The circuit can rotate the word by any number of positions. The drawing is labeled to reflect a left rotation of one position. The only bits which need to move across subwords are `a`, `e`, `i`, and `m`, which conveniently all arrive to and depart from the same RAM at the top left. This RAM's color shift indicates

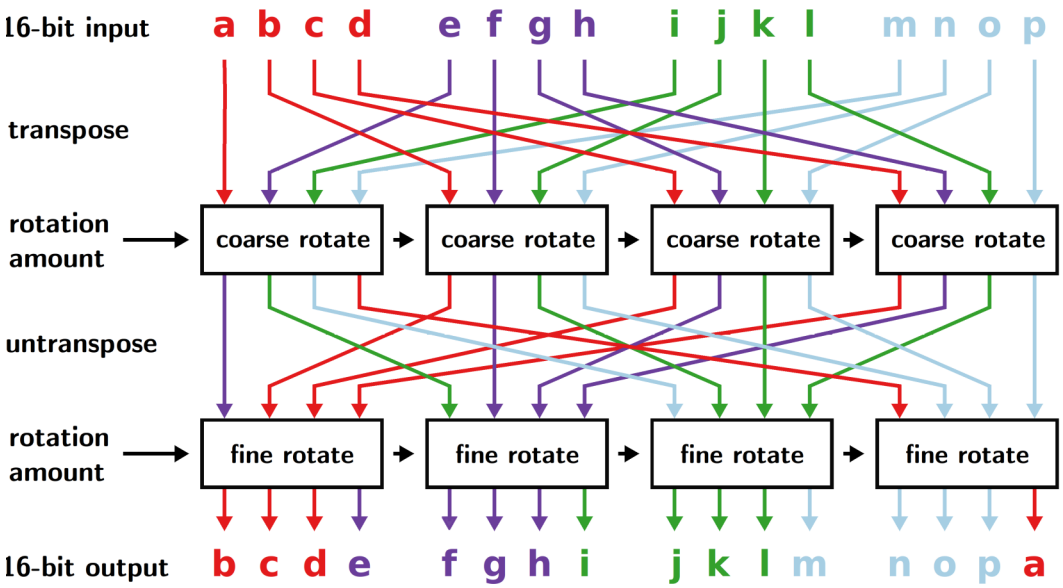


Figure 2. 16-bit logarithmic shifter

this rotation. The three other RAMs pass their bits through without adjustment, as shown by the unchanged colors.

The second layer of RAM gets to clean up from a minor mishap: the first layer got each bit to its correct subword, but not to the correct position within its subword. The order of the bits arriving at layer two, after the untranspose hustle, is `ebcd ifgh mjkl anop`. It is a simple, table-driven operation to bring these bits into their final sequence of `bcde fghi jklm nopa`. If a left shift were desired instead of a left rotation, either layer can mask out the `a` bit.

Note that Figure 1's adder, nothing happens with the machine word as carries are decided by the RAM in the middle. This idle time is wasteful, and happens to be the same duration as the delay through the first layer of Figure 2's logarithmic shifter. So two transpositions (with no delay other than wire length) and a row of RAMs can be added to a 16-bit carry-skip adder as a new middle layer parallel to the carry decision RAM, and the bottom layers of the adder and shifter can be superposed into the same RAMs at different addresses. The outcome is a simple ALU that can either add or shift during a machine cycle. This concept is expanded now into a nontrivial ALU with additional capabilities.

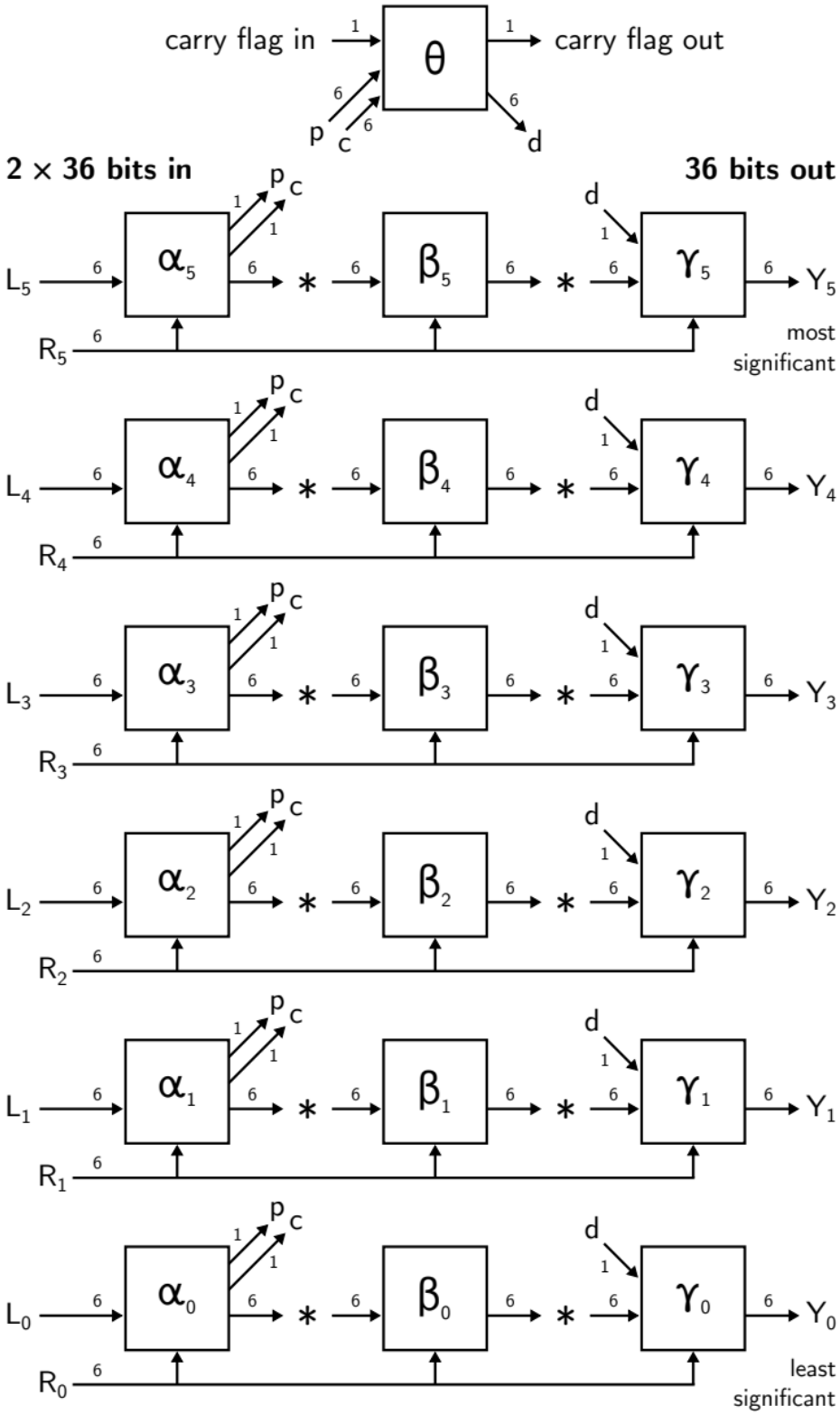


Figure 3. Block diagram of 36-bit ALU

The ALU of Figure 3 offers an example of the regular structure and high flexibility that SRAM can bring to complex logic. The drawing has been simplified for legibility, and shows the interconnections between 19 surface-mount RAM chips. Not shown are a 20th RAM that computes status flags, five glue logic gates,1 and roughly 40 analog switch ICs that initialize the RAM at system startup. Once initialized, the ALU SRAM content does not change, and the chips function as fast ROM.

All but one of the RAMs has 16 address lines and 16 data lines, and therefore are a 64k $\times$ 16 or 1 Mibit configuration. The exception is marked $\alpha 5$ and requires 128k $\times$ 16. Both of these sizes are commonly available. Although many of the data lines go unused by this ALU, narrower configurations are less available and cost more.

The data's path through Figure 3 is from left to right. The essential organization of the ALU is that two 36-bit operands, designated L (left) and R (right), are fed through three layers of RAM. In order to handle all 36 bits, each layer uses six RAMs that each look up functions of two 6-bit inputs applied to the address lines, and produce a 6-bit output at the data lines. Additional address lines come from the control decoder and select the function to be computed.

The RAM layers are simply named α, β, and γ after the order of their appearance. They defy more descriptive names, because each layer has multiple functions that depend on the type and particulars of the operation being computed. Most operations configure the layers into one of these general arrangements:

1. Carry-skip adder. RAMs α_0–α_5 add 6-bit subwords, passing the result through the β layer unchanged to RAMs γ_0–γ_5. The γ layer adds one, where necessary, to the subwords to obtain the correct 36-bit sum or difference. The determination whether or not to add one for each subword is decided by RAM ϑ, based on carry and propagate bits computed by α_0–α_5. The upshot is that this ALU is able to add or subtract 36-bit integers in one pass. The time required is three RAM access times (about 30 ns), plus delays from board capacitance.

2. Logarithmic shifter. This arrangement provides shifts and rotations of the left operand. The number of positions to shift is encoded in the right oper-

and and is replicated across the six subword positions. For this use, the α layer passes the left operand unchanged. Connections into and out of the β layer are transposed, meaning that each β RAM accepts one bit from each α RAM, and distributes one bit to each γ RAM. This allows β to rotate by any multiple of six bits, producing a "coarse" shift operation. The γ layer does "fine" subword rotations of zero to five bits. Proper coordination between β and γ thus permits rotation by any amount encoded into the right operand. Shifts are implemented as rotations with additional overrange checks at the α layer, masking at the β layer, and sign extension when needed at the γ layer.

3. Substitution-permutation network. By supplying a different S-box within each of the 18 RAMs, an invertable mixing function is applied to 6-bit subwords in the α and γ layer, and across 6-bit subwords in the β layer. The result is intentional chaos. On average, a single-bit change in the left (right) operand results in 15.37 bits (16.47 bits) changing in the result.[4] Note that an ideal mixing function would change half of the 36 bits on average, or 18 bits. The mixing operation can be used directly as a hash function for associative arrays, using one CPU cycle per word output.

A second SPN use is to feed the mixing function back via the left operand while a linear feedback shift register ALU operation drives the right operand.[5] This results in a strong (but not cryptographically secure) pseudorandom number generator that passes all Dieharder [6] statistical tests, yet requires only two CPU cycles per word output.

A third use is as a one-cycle round function for a 36-bit block cipher. There are some cryptographic susceptibilities to overcome that are beyond the scope of this research, such as differential analysis (S-box issues) and birthday attacks (block size issues) [5, 18]. Also, ciphers designed for this ALU will not perform efficiently on other computer architectures, and vice versa. But if a secure cipher can be derived using this scheme, it could greatly speed encryption for network-connected minicomputers.

---

4 Metrics would vary depending on the S-boxes chosen. The tested configuration filled the boxes using a nothing-up-my-sleeve binary expansion of the square root of 2.

5 The LFSR characteristic polynomial tested was $x^{36} + x^{31} + x^{13} + x^7 + x^6 + x^5 + x^3 + x^2 + 1$.

4. Bitwise Boolean logic. This ALU supports all 16 Boolean functions that take two inputs. Because the RAMs are crowded and memory is scarce, the 16 functions are not all directly implemented, but built up through the layers. For example, the Boolean function LONR (Left Or Not Right) is composed by tasking α, β, and γ to AND, NOT, and XOR their respective inputs.

5. Permutation component. The β and γ RAMs each include a table of 64 preselected permutations of six bits. The β permutations operate across subwords while holding the bit position within subwords constant, and the γ permutations operate strictly within subwords. By composing from this limited "alphabet" of permutations, it's straightforward to compute any permutation of a 36-bit word in not more than five CPU cycles.

6. Ad hoc logic. Several ALU operations don't fall squarely into one of the above layer use patterns. For example, there is a DSL (Double Shift Left) operation that does the double-register shift operation `ABCDEF GHIJKL → BCDEFG`, where each letter represents six bits of a register. This operation is a basic step in 36-bit multiplication and is implemented via the α and β layers. There are many other ad hoc operations.

A full explanation of the ALU's internal operation, semantics, corner cases, and flags is too much detail for this proposal, but will appear in a paper. A preliminary description is available at [1], with design corrections from testing at [2]. Table 2 provides a partial list of operations with the number of instructions needed for each. In the architecture of this proposal, the execute stage of all instructions takes exactly one cycle, so counts of cycles and instructions are equivalent.

The proposed architecture does not include a full hardware multiplier, although [1] gives a detailed account of how and when to build a fast SRAM multiplier. Otherwise, full-word multiplication is done in software, using six specially designed ALU operations to speed multiplication routines. The 47 cycle worst case is not expected to apply to most instances of multiplication. Most often, multiplication is used to compute offsets of small array elements, and those multi-

Table 2. Principal ALU operations and timings

Operation	Number of cycles
bitwise Boolean logic	1
add, subtract	1
magnitude compare, maximum, minimum	1
shift/rotate by 0 to 63 bit positions	1
reverse bits of 36-bit word	1
36-bit linear feedback shift register	1
hash function for associative arrays, per word hashed	1
round function for 36-bit cipher	1
absolute value	2
leading or trailing bit manipulation (40 operations)	2
pseudorandom number, per word output	2
population count (Hamming weight)	3
unsigned multiply 36 bits × 6 bits	3
count leading or trailing zeros or ones	5
any permutation of 36 bits	≤ 5
36-bit multiply with 72-bit result (rivals Intel 80486)	47

plications only need three cycles due to their small multipliers and existing ALU support.

As mentioned on page 4, the ALU should not leave preventable hardware challenges on the table that can lead to exploitable defects. Over several decades, integer arithmetic wrapping has become a security concern, and treatment stopgaps have come at high cost. For instance in the C language, code that once read

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

now has to read something along the lines of

```
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" [42]. Not only is this solution burdensome on humans who write and maintain C code, but it also slows the compu-

tation about fivefold. The weight of these problems can be much lightened by changes to the architecture itself, changes that are simple in nature and implementation.

Just about every CPU and ALU knows when some kind of wrapping occurs. Usually there is a carry flag that should not be set for unsigned arithmetic, and an overflow flag that should not be set for signed arithmetic. This scheme has two drawbacks. First, the instruction set only distinguishes signed vs. unsigned operations by how their flags are interpreted. This is to say, the ALU operates in the absence of signedness information. The second defect is that abnormal flag outcomes are not retained, but overwritten on the next operation. The proposed architecture rectifies both problems, allowing code in low-level languages such as C and assembler to be written like this:

```
/* Register signedness transfers into machine code. */
unsigned int a, c, e, g;
signed int b, d, f, h;

/* Do as much arithmetic as desired. Don't test overflow. */
c = a - b;
h = d << e;
f = h + c;

/* Only test for exactly the knowledge needed. */
if (something_bad_ever_happened)
    deal_with_it_here;
else
    return f;
```

The ALU of Figure 3 has distinct operations for each combination of signedness. For example, there are eight operations that add 36-bit words. Four produce an unsigned result, and permit operands where both are signed, both are unsigned, the left only is signed, and the right only is signed. Another four produce a signed result with the same breakdown. What this means is that the ALU always knows whether or not the sum will fit in its destination register. If it does not fit, it sets the T flag.

The T flag indicates neither carry nor overflow, but that a “Temporal range” error has occurred. It works for any combination of operand and destination

signednesses; there aren't alternative flags to check depending on the types involved, as happens with traditional carry and overflow flags. T is more unified. It's called temporal, because the next instruction will overwrite the flag. Many programs will never need to inspect T, because if an instruction produces an invalid result, the needed information is not which specific instruction fails, but that the result is not correct. This is the purpose of the R flag.

The R flag indicates that a "Range" error has occurred for a past arithmetic instruction. It is set automatically when T is set, but it is not cleared by subsequent arithmetic. R is only cleared by an explicit CRF (clear range flag) ALU operation. So if a program has an integer expression or series of expressions to evaluate, the math itself can be separated from the error trapping. Once the result is obtained, R can be checked just once, and the result will be valid if R is clear.

A useful side effect of this range checking and signedness stratification of operations is that the programmer can be less thoughtful about variable types. In C and other languages, promotion rules, function prototypes, and implementation-dependent behavior make choosing variable signedness difficult. A programmer often knows that an integer variable should never be negative, but can still be pressured into declaring it signed in order to avoid breaking something. All this leads to confusing code, confused maintainers, and ultimately exploitable defects. For instance, C does not know how to add an unsigned operand to a signed operand. The ALU of this proposal does. In particular, range checking works correctly over the integers for:

- addition and subtraction for all signedness combinations,
- maximum and minimum for all signedness combinations,
- magnitude comparison for all signedness combinations,
- array boundary checking for any index signedness, and
- left and right arithmetic shifts of 0 to +63 bit positions for any signedness of the quantity to be shifted and any signedness of the destination register.

A minor limitation of this list is that shifting integers by considerably larger than the word size, or by a negative number of bit positions, is not supported. The reason can be seen in Figure 3: the number of positions to shift is conveyed

via the right operand, and must be distributed to all RAMs in the β and γ layers. Only six bits are available to encode the shift amount for each RAM, limit the number of encodings to 64. As the word size is 36 bits, 37 of those encodings represent shifts of 0 to 36 positions. For the 27 remaining encodings, it was decided that supporting shifts of 37 to 63 places would have better consistency and usefulness than enabling shifts of −1 to −27 places. Thus shifts in either direction are supported by using distinct left and right operations.

This ALU's shift semantics are a little different than on conventional processors. Shifts of unsigned words are traditionally called *logical shifts*, and signed word shifts are *arithmetic shifts*. On this ALU, both of these cases are arithmetic shifts, because their results will be range checked. Unchecked shifts by this ALU are termed logical shifts. Checking shift results is important, because the ALU has no multiply instruction, implying that an arithmetic shift is *the* way to multiply or divide by a constant power of two.[6] And as the R flag supports compositions, statements such as

```
kilo = kilo << 10 - kilo << 4 - kilo << 3;
```

can multiply `kilo` by 1000 in five CPU cycles without need to test for overrange during the computation.[7] The variable may be signed or unsigned.

Firmware for the ALU (that is, the contents of its 20 SRAM chips) has been generated and tested successfully in simulation, including the correct operation of all flags and intended detection of overrange conditions. The test vectors are randomly generated, with some skewed to occur near corner cases. Results are verified using code and algorithms that are independent of the ALU's firmware. For instance, the firmware adds by combining results of 6-bit subword additions, but the verification checks by adding 36-bit integers on 64-bit hardware. Overrange handling for shifts is verified in the simulator not by checking if results are out of range (as this is how the ALU does this), but by checking if the inverse shift yields the original word. 6-bit permutations are tested by composition into

6 The case of negative dividends does not work, as rounding is toward negative infinity.

7 The applicable domain for the multiplicand is about 2.5% less than a non-subtractive method would provide, but overrange is detected when the supported domain is exceeded.

randomly chosen 36-bit permutations. And so on. The outcomes of these tests give strong additional assurance that the ALU firmware is correct.

The simulation does not yet implement all operations that are planned for the ALU, but the operations where implementation presented technical challenges are complete. All that is implemented also has a working regression test. The current firmware includes all addition, subtraction, rotations, logical shifts, signed and unsigned arithmetic shifts, magnitude comparison, short unsigned multiplication, bitwise Boolean functions, bit permutations, array bound checking, S-box mixing for hashing/PRNG/ciphers, clearing the Range flag, and a pointer alignment operation. All of this code is available from [2]. A separate simulation demonstrates that a 47-instruction full unsigned multiplication routine works.

ALU firmware not yet written includes less frequently used, fairly straightforward operations such as absolute value, population count, leftmost zero isolation, and many others.

The work of this dissertation is to assess whether this promising ALU simulation can be expanded into a practical and working hardware platform. If this innovation proves to be feasible, it may emerge as the world's first "gold standard" for transparently functioning, fully auditable, and user-constructable computers for integrity- and confidentiality-critical missions.

6. Minicomputer and research plan

The objectives of this research are to establish the feasibility of building a practical solder-defined minicomputer, and to ensure that the outcome is reproducible. The proposed system will consist of a CPU, primary storage, an I/O mechanism, and necessary firmware. The computer will be constructed using maker-scale assembly tools, with the hope that when it is complete, it will be unlikely to contain backdoors or exploitable hardware irregularities that survive the assembly process. If the system proves to be feasible, its attack surface can be far smaller than that of a traditional computer:

1. The CPU and its firmware can be inspected and validated against publicly available references without reverse engineering any VLSI.

2. The CPU and its firmware can be produced on demand by the buyer or any of myriad, often local, circuit board assembly services.

3. Hazardous complexity can be eliminated, precluding exploits such as Spectre [22], Meltdown [23], and RowHammer [25].

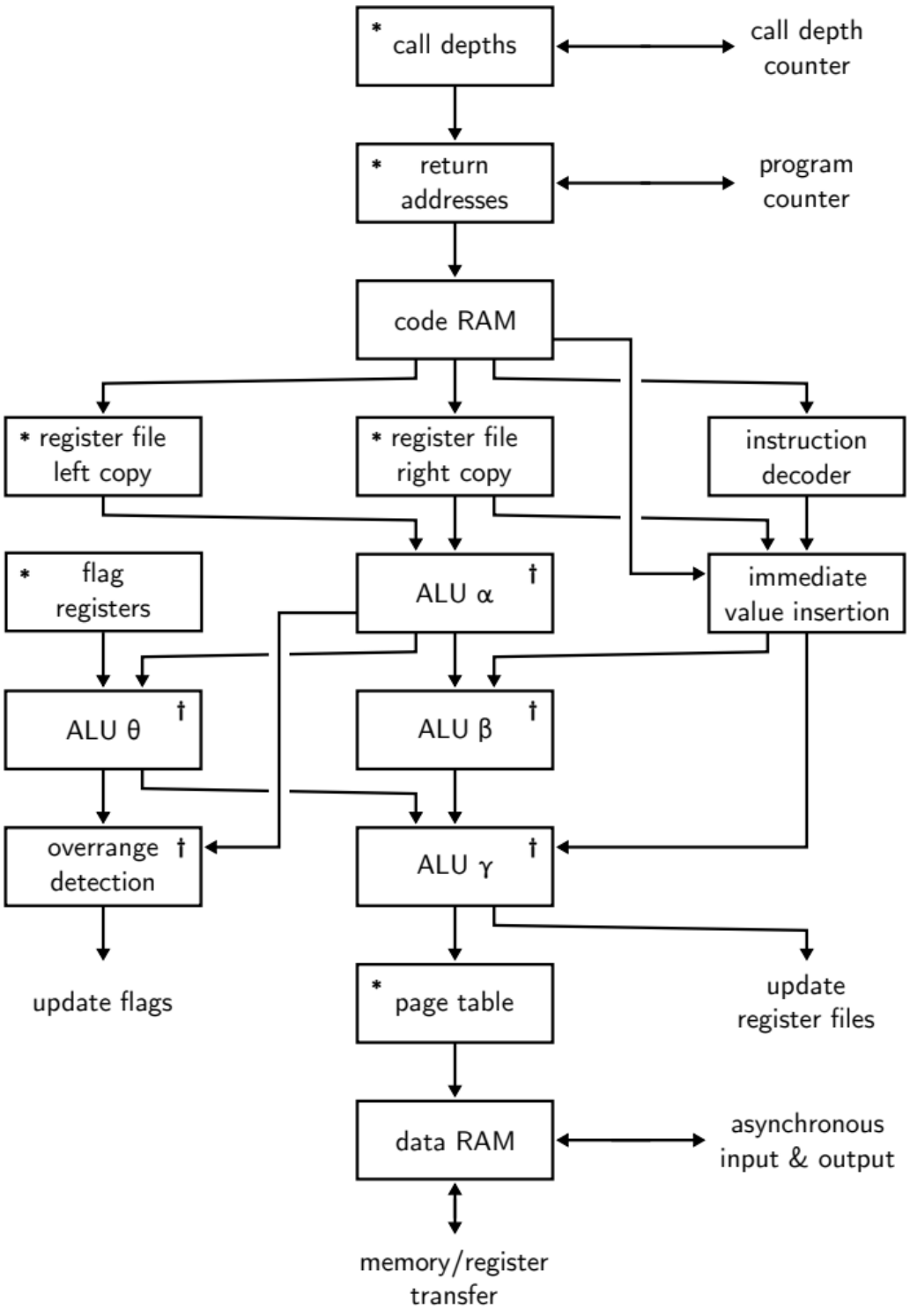
4. Instruction set semantics can assist in detecting and containing unexpected behavior, such as integer wraparound.

5. The I/O mechanism can be fortified to safeguard the I/O bus and bus-attached memory from attack by rogue peripherals.

6. Full documentation and executable models for simulation can be available to and challenged by anyone, without requiring a physical system.

7. The buyer can have exclusive control over availability, service life, manner of use, features, modifications, support, and access to repair.

Figure 4 presents a general map of how such a minicomputer may be architected. Everything drawn is subject to improvement and new ideas. Each box in the figure represents an SRAM module, either a single IC or a small number ganged in parallel. The arrows show information flow during a CPU cycle, not the complex interconnections that implement the flow. Only the first appearance of each module is drawn: the two register file copies appear in the fourth line from the top, but are accessed again in either the eighth or tenth line. There isn't



* Additional input specifies the program that is running.

† Additional input comes from the instruction decoder.

Figure 4. CPU cycle depiction with static RAM shown as boxes

room to draw all necessary arrows, so † and * are used to abbreviate two frequent sources.

This is a Harvard architecture computer, meaning that primary storage is divided into separate RAM ICs for code and data. These two memory banks can be accessed simultaneously. For simplicity, performance, and security reasons, only static RAM is used for primary storage. There is no DRAM. There is also no CPU cache, as no speed gap exists to improve on between chips for cache memory and chips for primary storage. In lieu of cache memory, a large number of general-purpose registers, at least 512 per program running, significantly reduce the number of instructions and register spills needed. As the two register file copies are at least 64k×36 each, at least 128 programs can be resident without need to save or restore any registers when switching.

This CPU does not have a stack frame and does not directly support recursive functions. This allows better allocation and reuse of general-purpose registers, while ruling out stack overflow and return address overwrites. The program counter is implemented as the top element of a stack of return addresses, clustered according to which program is running. If no suitable counter ICs are available for sale, linear feedback shift registers may be implemented instead, as was once done in [33] for other reasons.

Subroutine return addresses exist solely in physically segregated SRAM. All branches, including calls, are to fixed 27-bit addresses within their 36-bit instructions. The only possible subroutine return point is the instruction immediately following the call. The CPU does not directly support relocatable code or pointers to functions. This eliminates need for program memory management hardware, because the operating system can detect and preclude any unpermitted branches at program load time. It also simplifies calling library code, which works exactly like calling a regular subroutine, provided that the library routine's linkage is authorized at program load time.

Along the same lines, no hardware is needed to keep ordinary programs from executing restricted opcodes, such as opcodes that write to program memory or access hardware. The operating system can trivially scan for and disallow restricted opcodes as programs are loaded.

The code RAM is organized as 36-bit words. This is a readily available width for memory. Most instructions use nine bits each for the opcode, two source registers, and a destination register. The most promising location to divide the CPU cycle into a two-stage pipeline stages is immediately after instruction fetching. Stage two would begin with operand fetching and instruction decoding. The simplest way to preclude data hazards is to not further divide the stage two into smaller segments, so each instruction completes its register or memory write before another instruction fetches any data.

With a two-stage pipeline, a branch hazard will exist, because the first instruction after any branch will be unconditionally loaded and decoded. The simplest remedy is to document the hazard, and let the assembler, compiler, or programmer to generate code that accommodates this behavior. Much of this hazard's performance penalty can be optimized out.

The ALU has a benign bottleneck in that its three layers all receive the same right operand. More could be accomplished in one cycle if different layers could receive different right operands. This enhancement is labeled "immediate value insertion" in Figure 4, where right operands for β or γ can come from several sources. It also solves the problem as to how constants are loaded into registers or memory for the first time, as well as reduces register pressure.

If shortening the second pipeline stage becomes necessary, the page table and RAM access may be moved up parallel to the ALU, at the cost of losing use of the ALU for address arithmetic within the same CPU cycle. As the RAM does not require caching and takes only 10 ns, the most efficient place to introduce I/O may be via the main RAM. This approach would need the I/O controller to provide memory protection to protect the RAM from unauthorized reading or tampering by faulty or compromised peripherals.

The ALU omits fast hardware multiplication, because it is not a consistently cost-effective means of improving performance. For systems where the cost can be justified, [2] shows how to build a fast SRAM multiplier. The CPU design is "multiplication ready" in that a 72-bit result can be split between the two 36-bit register files using a single write, and the instruction set already has 16 opcodes for processing these "half registers" independently.

From an investigative research perspective, there are four key hurdles to overcome in as elegant a manner as possible, given the limited selection of available components:

1. SRAM data lines are bidirectional, and the CPU needs many interconnections, often 36 bits wide, between many segments. Capacitance and added drivers are both penalized, and their tradeoffs must be considered.

2. Many RAMs, such as the page table and code memory, need provision for occasional updates by the operating system.

3. All RAMs will need provision for loading firmware, using circuits that will need to drive every address and data line high or low as needed.

4. Timing and data flow relationships will be complex, call for a pipelined solution, and have a tight budget if a 10 MIPS goal is to be met.

This research will be successful if it manifests a minicomputer that is shown to do all of the following, first in simulation, and then with a physical prototype:

1. employ only simple, fungible components,
2. implement the operations stated in Table 2,
3. stay within the number of cycles in Table 2,
4. detect overrange conditions for any combination of signednesses,
5. latch the overrange CPU flag for inspection at a later point,
6. provide a no-wait-state memory subsystem using SRAM,
7. support some form of bus-safe I/O for at least 7 devices,
8. provide preemptive multitasking,
9. take no more than 5 CPU cycles to fully switch programs,
10. supply a resident monitor program for testing and diagnostics,
11. prevent unauthorized programs from accessing others' memory,
12. prevent unauthorized programs from using restricted instructions,
13. load its own firmware from a single flash memory IC, and
14. sustain 5 million instructions per second (the hope is 10 MIPS).

The research is anticipated to take one year. The tasks will be interleaved so as to make the best progress, accommodate supplier and assembly lead times,

and balance the tasks engaged in to assure freshness and diversity. These tasks will include:

1. an accepted journal article describing the ALU,
2. specification of the complete instruction set for the CPU,
3. validation and testing of the instruction set in a mocked-up system simulation (of the instructions, not of the hardware),
4. an assembler and resident monitor program written,
5. full component selection and electrical design of the CPU, memory subsystem, I/O subsystem, and initialization (firmware load) circuits,
6. access points added to the design to facilitate physical testing,
7. simulation testing of the complete electrical design and its access points, including verification of the 14 success criteria of the preceding list,
8. a relationship established with a circuit board assembly firm,
9. layout of one or more prototype circuit boards,
10. component procurement and circuit board assembly,
11. CPU testing, troubleshooting, and re-manufacture as needed, and
12. functional verification of the assembled minicomputer, based on the 14 success criteria.

7. Dissertation outline

Here is a sketch of the sections the dissertation may include.

1. Overview
   - 1.1 Categories of hardware behavior irregularities
   - 1.2 Opportunity to reduce hardware attack surface
   - 1.3 Practicality and economy of building CPUs
   - 1.4 Logic family selection
   - 1.5 SRAMs as electrical components
2. Notation and definitions
3. SRAM building blocks for complex logic
   - 3.1 Simple lookup elements
   - 3.2 Arbitrary geometry adders
   - 3.3 Carry-skip adders
   - 3.4 Swizzlers
   - 3.5 Logarithmic shifters
   - 3.6 Semi-swizzlers
   - 3.7 Substitution-permutation networks
   - 3.8 Fast multipliers
4. Glue logic for special challenges
   - 4.1 SRAM initialization
   - 4.2 Fast counters
   - 4.3 Latches, multiplexers, and tristate devices
5. 2-layer arithmetic logic unit designs
   - 5.1 A 2-layer ALU for 36-bit words
   - 5.2 A tiny ALU for 18-bit words
6. 3-layer arithmetic logic unit designs
   - 6.1 A 3-layer ALU for 36-bit words
   - 6.2 Additive operations
   - 6.3 Bitwise boolean operations
   - 6.4 Compare operations

8. Schedule

Members of the dissertation committee are encouraged to contact the author at any time with questions, suggestions, or for any other reason. The author will invite the committee members to meet with him at a mutually convenient date at the start of each semester, including summer. The author and chairperson will meet more frequently at appropriate times.

The author will email an appropriate report of his progress, as well as need for any assistance, to all committee members on or just before the 15th day of each month. These reports can increase in frequency if appropriate.

The target dates of Table 3 are products of estimates and simplified explanations. Two contingencies are attached:

1. The quality, completeness, and most importantly impact of the research take priority over publication dates and graduation date.

2. The author has applied for a National Science Foundation grant that overlaps in scope with this proposal. If funded, specific tasks may need to extend into the first quarter of 2022 to meet the NSF contract's obligations.

Table 3. Milestone goal dates

Task	Substantial completion
ALU firmware and features finalized	11 Sep 2020
ALU journal article submitted	2 Oct 2020
instruction set fully specified	18 Dec 2020
virtual machine works at instruction level	18 Dec 2020
assembler written	18 Dec 2020
resident monitor written	18 Dec 2020
return from break	4 Jan 2021
difficult circuit questions answered	29 Jan 2021
test fabrication of smaller circuits	26 Feb 2021
netlist specified	26 Mar 2021
virtual machine works at component level	26 Mar 2021
bill of materials finalized	26 Mar 2021
components on order	2 Apr 2021
Gerber and placement files ready	30 Apr 2021
prototype received	28 May 2021
prototype evaluated	11 Jun 2021
draft dissertation available to committee	30 Jul 2021
presentation materials ready	13 Aug 2021

9. References

1. Marc W. Abel. 2020. Elegant ALUs from Surface Mount SRAMs. https://talk.wakesecure.com/abel-alu-draft-1.pdf
2. Marc W. Abel. 2020. Untitled tarball of C and Python source code. https://wakesecure.com/alu-sim-0.1.tar.gz
3. 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.
4. 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).
5. 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
6. Robert G. Brown et al. 2020. Dieharder: A random number test suite. Retrieved from http://webhome.phy.duke.edu/~rgb/General/dieharder.php
7. Andrew Butterfield et al., eds. 2016. *A Dictionary of Computer Science* (7th ed.). Oxford University Press, Oxford, England.
8. CTS Labs. 2018. Severe Security Advisory on AMD Processors. CTS Labs, Tel Aviv, Israel.
9. Department of Computer Science and Engineering, Wright State University. 2019. *Program Guide and Policies*, approved April 26, 2019.
10. Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. At *Black Hat USA 2018*, (Las Vegas, NV), white paper.
11. Pavel Dovgalyuk. 2019. Relay computer computes 7 digits of pi. Video retrieved April 4, 2020 from https://www.youtube.com/watch?v=bOOCfx2EN10

12. 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
13. Jon Gambrell and Josef Federman. 2020. Fireworks, ammonium nitrate likely fueled Beirut explosion. Associated Press, August 5, 2020. Retrieved August 6, 2020 from https://apnews.com/cbeb3263d6fc30a63a0300f588e7207b
14. 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.
15. 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
16. IBM Security. 2020. *Cost of a Data Breach Report*, 2020. Retrieved August 6, 2020 from https://www.ibm.com/security/digital-assets/cost-data-breach-report/Cost of a Data Breach Report 2020.pdf
17. International Organization for Standardization. 2018. ISO/IEC 27000:2018(E). *Information technology – Security techniques – Information security management systems – Overview and vocabulary*. Retrieved July 14, 2020 from https://standards.iso.org/ittf/ PubliclyAvailableStandards/ c073906_ISO_IEC_27000_2018_E.zip
18. 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
19. Karl G. Jansky. 1933. Electrical disturbances apparently of extraterrestrial origin. *Proc. Inst. Radio Engineers* 21, 10 (Oct. 1933), 1387–1398. DOI: https://doi.org/10.1109/JRPROC.1933.227458
20. Dmitry Janushkevich. 2020. *The Fake Cisco: Hunting for Backdoors in Counterfeit Cisco Devices*. Version 1.0, July 2020. Retrieved July 19, 2020 from https://labs.f-secure.com/assets/BlogFiles/2020-07-the-fake-cisco.pdf

21. 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.

22. Paul Kocher et al. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, (San Francisco, CA), IEEE, 1–19.

23. 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.

24. The MOnSter 6502. Retrieved from https://monster6502.com/.

25. 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

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. Michael Pompeo. 2020. The tide is turning toward trusted 5G vendors. Press Statement by the Secretary of State. June 24, 2020, (Washington, DC). Retrieved July 8, 2020 from https://www.state.gov/the-tide-is-turning-toward-trusted-5g-vendors/.

28. 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/.

29. Joanna Rutkowska. 2015. Intel x86 considered harmful. (October 2015). Retrieved April 4, 2020 from https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf

30. 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

31. 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.

32. 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.

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

34. 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).

35. Neil H. E. Weste and David Money Harris. 2011. CMOS VLSI Design: A Circuits and Systems Perspective (4th. ed.). Addison-Wesley, New York, NY, 475–476.

This proposal uses colors from https://colorbrewer2.org by Cynthia A. Brewer, Geography, Pennsylvania State University.