Dauug|36

A Solder-Defined Computer Architecture for Backdoor and Malware Immunity

Marc W. Abel

***** DAUUG|36 SPECIFICATIONS *****

System classification	solder-defined minicomputer
Logic family	SRAM with 74AUC
Memory protection	paged virtual memory
Multitasking	cooperative or preemptive
Word size	36 bits
CPU speed	16–20 MIPS
Registers per program	512
Programs ready to run	256
I/O buses	SPI and I2C
Operating system	Osmin or owner-supplied
Design lifespan	30 years
Manufacturer	anyone

Dauug|36: An Open-Source, Solder-Defined Computer Architecture for Backdoor and Malware Immunity

Marc W. Abel
Wright State University

Abstract

VLSI complex logic in safety- or privacy-critical computers too often conceals Faustian ambushments. Systems are trusted and touted in early life as "secure," only to be condemned forever by discovery and disclosure of silicon-borne exploitable defects. Users of VLSI complex logic can neither inspect that logic, nor audit it, nor know its secure life expectancy, nor know if someone already found a vulnerability.

This paper evaluates the immediate potential and practicality of building embedded systems and/or communication endpoints without using VLSI complex logic at all. I call such machinery *solder-defined.* Rather than assemble proprietary logic via nanometer-scale photolithography, millimeter-scale surface-mount methods can produce open-source machines using maker-scale assembly tools. Finished computers can be inspected at the logic gate level using ordinary microscopes and in-circuit testing, with all design of their hardware, firmware, and software open for anyone to confirm, challenge, or adapt. But such end-user control and community validation, although a fine start, are not sufficient to ensure that hardware is free from exploitable defects. The other benefit to solder-defined systems is their designs can be radically simplified relative to prevalent VLSI CPUs in pursuit of a commensurate radical reduction in concealed mistakes.

This paper introduces Dauug|36, an open-source, solder-defined, 36-bit computer architecture with paged virtual memory and preemptive multitasking. The CPU and memory subsystem are fully specified, have been electrically simulated, and surpass 16 million instructions per second—all without a single microprocessor, FPGA, PLD, or ASIC in the design. A novel characteristic of Dauug|36 is that it uses read-only synchronous static RAM (SRAM) as its foundational logic block, enabling an astonishingly sophisticated instruction set for a logic family that can be hand-soldered. For example, a 36-bit word's population count (Hamming weight) can be computed in just two instructions, or eight clock cycles. This paper also introduces Osmin, the first operating system for Dauug|36, and describes the implementation status of the hardware, operating system, and associated tools.

This paper was submitted to an open-access journal on February 28, 2024.
Licensing information will appear in the published edition.

`https://dauug.com/`

Front cover: Lowercase letters first appeared in Digital's DECwriter line with the introduction of the LA36, the first DECwriter II model. Its seven-pin font could not support descenders. The other side of the cable was often a PDP-10 mainframe, which marked the twilight not only of 36-bit computing, but also of solder-defined computers generally. Fifty years later, the twilight is back.

Contents

Figures

Tables

Table 1: Categories of vulnerability-inducing hardware irregularities

	Category I	**Category II**	**Category III**
Origin	purposeful	unexpected	malicious
Example	arithmetic wrap	RowHammer	hidden backdoor
Software workaround?	yes	no	no
VLSI architect can fix?	yes	yes	no
Supply chain owner can fix?	yes	yes	yes

1 Introduction

1.1 Problem Statement

Irregularities in the behavior of computing hardware are a perennial source of software-related defects, including exploitable defects that are regarded as security vulnerabilities. I group these irregularities into three loose 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 [35], and increasing the separation between the instructions that programmers write and the silicon executing them [12].

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 [7, 11, 15, 16, 22, 23, 26, 32, 38, 41]. Table 2 lists a few well-known cases. 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 victim's hardware, operating system, or application is no longer supported by its originator. 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 II hardware irregularities are analogous to software defects, and I view the two as having the same two root causes. First, systems are too complex relative to the human scrutiny applied to their design and update. I like to mention this in my talks about cybersecurity, and I often project Figure 1 during my explanation. Second, others have pointed out that long practice in both software and hardware design has presumed use under controlled, non-adversarial conditions. In the 1980s, the famed network infiltrations by Marcus Hess and Robert Tappan

Table 2: Some Category II (unplanned and unexpected) hardware irregularities

When	Architecture	Name	Synopsis
1985	80386	multiply bug	arithmetic error
1994	Pentium	FDIV	arithmetic error
1998	Pentium	F00F	lockup
2003	Via C3	God mode	privilege escalation
2008	Intel AMT	Silent Bob	full control of everything
2015	DRAM	RowHammer	memory corruption
2017	x86	Spectre	read others' memory
2017	x86, POWER, ARM	Meltdown	read all memory
2020	Intel SGX	load value inj.	inject data values
2020	Intel CSME	[M. Ermolov]	broken authentication

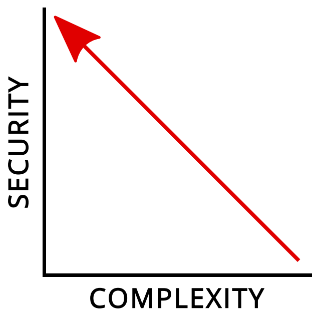


Figure 1: To increase security, reduce complexity.

Morris disproved this presumption [36, 37].

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 [6, 11, 14, 27, 30]. Table 3 lists some examples. Such vulnerabilities are of particular urgency when national boundaries are crossed [29]. Another model that falls under Category III is outright counterfeiting of critical equipment by unknown manufacturers, such as reported in July 2020 for network switches [21], or for many years for integrated circuits for aviation and military use [17]. Another Category III situation is when a manufacturer knowingly releases defective semiconductors for use in aerospace. One of my mentors was once fired for refusing to sign off on such shipments.

Table 3: Actual and rumored Category III (maliciously introduced) hardware irregularities

Who	**Architecture**	**Synopsis**
AMD	Platform Security Processor	hypothesized backdoor
Apple	iPhone 6 + iOS 10.2.1	sabotaged performance
Deere	8520T tractor	right to repair infringements
Huawei	5G cellular infrastructure	potential for China influence
Intel	Management Engine	hypothesized backdoor
Intel	RDRAND instruction	non-randomness suspicions
NSA	ANT Catalog	implantable surveillance products
VIA	C3 (x86 clone)	backdoors claimed by C. Domas
ZPMC	ship-to-shore cranes	undisclosed cellular modems
ZTE	5G cellular infrastructure	potential for China influence

Table 3 slips in an important point about *right to repair.* ISO 27000 [20] incorporates *availability* into the scope of *information security*, so when a manufacturer elects to make a deployed tractor unrepairable by its owner in order to sell a service call, a cyberattack has been committed via the supply chain. Likewise, the sale of digital devices that contain non-user-replaceable batteries constitutes a security affront, a brazen act of ecological terrorism, and an amplifier of human oppression in regions where conflict minerals are mined.

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 either via software or through hopes that a supplier will abstain from inserting them. Instead, the buyer must extend some form of control over the manufacturing process and/or the final assembled product.

1.2 Seeking a Solution

Various proposals have been made for safeguarding the VLSI supply chain against intentional defect introduction, including [19, 24, 29, 42] which are summarized in Table 4. The suggestions range from never trading with adversaries, to securing a multi-billion dollar supply chain across hundreds or perhaps thousands of vendors, to proving mathematically that designs are secure but giving no evidence that hardware actually sold follows those designs, to using one of the planet's most expensive machines to partially see inside ICs, to suggesting that complex ICs not be used for certain infrastructure. Not one of these proposals is practical, but the last of these, which I confess is my idea, is perhaps the *least impractical* and most immediately realizable for certain uses.

This paper's essential target is hardware irregularity Category III, the case where

Table 4: Proposed VLSI supply controls for Category III backdoors

Proponent	**Synopsis**
Michael Pompeo	geopolitical controls
Adam Waksman	lock down VLSI supply chain
Eric Love	add formal proofs of security to hardware IP
Mirko Holler	X-ray ptychographic inspection
this paper	all complex logic to be built by end user

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 his or her own complex logic. There is also a backup approach: have the buyer inspect his or her 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 paper is that it is possible to build an easy-to-reproduce 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 to make it impractical 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.

[1]This term is defined on page 22.

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.

These tamper resistance criteria, while useful for excluding Category III hardware irregularities that are maliciously introduced, don't address exploitable defects from Category I's unintended consequences or Category II's unforeseen behaviors. To address all three sources of exploitable vulnerabilities, we need to look at not just at computing machinery's tamper resistance, but also its behavioral characteristics. My minimum expectations include at least the following:

Behavioral expectations of computers and CPUs

1. Computers must not run operating systems. Instead, operating systems must run computers.[2]
2. A computer must not facilitate any exploit that can bypass any control of the operating system that is running it.
3. A computer must run any and all adversary-supplied code without possibility for privilege escalation,[3] never violating operating system permissions or resource limits.
4. Included or attached hardware, as well as buses, must not facilitate any exploit that can bypass any control of an operating system that is running a computer.
5. Included or attached hardware, as well as buses, must not exchange data with any other hardware, bus, or memory without authority from the operating system to do so.
6. Attaching a computer to a network must be no less secure than keeping the computer air-gapped.
7. A CPU must unconditionally protect all instruction pointers from tampering, including branch targets and subroutine return addresses.
8. A computer's security must not be fragile in the presence of malformed input.

[2]In other words, the unified semantics of the hardware and operating system must be as specified in the OS source code—neither buried in nor modified by undisclosed, proprietary, and/or non-inspectable electronics.

[3]A regrettable exception is needed for software that persuades a human to defeat security measures. According to Rice's theorem, such malware cannot dependably be identified by automated safeguards [31].

9. A CPU must mitigate unanticipated modular arithmetic wrapping without bloat, inconvenience, slowdown, or incorrect results.
10. A CPU must never give incorrect results merely due to unexpected signedness or unsignedness of an operand.
11. A CPU must support preemptive multitasking and memory protection, except for uses so simple that the application running and its operating system are one and the same.
12. A computer must offer hashing, pseudorandom number generation, and cryptography that are fast enough for its intended use.
13. A computer must not depend in any manner on microcode or firmware updates for its continued security or suitability for use.
14. A computer must be repairable by its owner, particularly with regard to on-site replacement of components or stored data that the owner might foreseeably outlive.
15. A computer must be identically replaceable for as long as it is needed.
16. A computer must be delivered with objective, verifiable evidence of conformity with these expectations.

Simultaneously satisfying the above expectations is comfortably within human intellect to accomplish: it's principally a discipline of scaling automation responsibly. Starting at the bottom, I own two sorobans that generally meet the parameters of this list, even though they don't use electricity to do arithmetic. Moving up the automation scale, I own two TRS-80s that also run fine. Although lacking, these mid-1980s computers conform closer to my expectations than the 64-bit machine that typeset this paper. So I started to think about what capabilities need to be added to these ancient, trustworthy computers to build usable systems today without making irresponsible security tradeoffs. I also contemplated what components could be used to build machines without purchased complex logic—that is, without any microprocessors, FPGAs, PLDs, or ASICs.

A special challenge with this research is that when it comes to remotely- or malware-exploitable hardware defects, an incremental improvement is not enough. The number of defects, unless zero, is immaterial. Likewise, the number of attacks that can succeed in a script, unless zero, is immaterial. Even the knowledge that a computer has no *known* exploitable defects is not sufficient. Once manufactured, hardware is unlikely to be patched, and once deployed, hardware is unlikely to be removed from service. The standard of care needs to be immunity—not resistance—to backdoors and malware.

No claim that any combination of hardware, firmware, and software is immune to hacking and malware is valid without proof. This is not the same as a supplier's assurance, adherence to so-called best practices, successful passage of a series of tests, exclusive use of open-source designs, or a certification. Any path to proof is likely to be lengthy and expensive; however, the system will never thereafter fall prey to a covered exploit, nor will it ever need a security update. A proof's high cost and and lead time can be beneficial in themselves because they promote configuration stability, discourage frequent releases, and abhor bloat.

My goal for the architecture of this paper is to soon finish its hardware design, firmware, real-time operating system (RTOS), cross assembler, and self-hosted assembler. Each of these is to be minimalist, succinct, open-source, and thoroughly documented. All except the cross assembler are to be likely free of exploitable defects.[4] A mathematical, semi-formal, or formal proof that the hardware, firmware, and kernel are immune to hacking and malware would follow at a later time.

1.3 Alternatives to VLSI Complex Logic

Almost all CPUs made out of anything except silicon wafers lately have been somebody's avocation, such as the many homebrew CPUs cataloged in [40]. But the research of this paper 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 more. 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 are some logic families that might be usable for building transparently-functioning computers. Neither practicality nor seriousness was a requirement to appear on this list because every choice here has important drawbacks. I found it better to start with an overly 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. One benefit of using relays is resistance to electrostatic discharge.

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 may 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 [34].

Nanoscale vacuum-channel transistors may someday work like vacuum tubes without filaments, but at present are only theoretical.

Transistors in individual packages may be within scale if extremely small packages such as VML0806 (0.8×0.6 mm footprint) are used. An advantage of discrete transistors is that no component sees more than one bit position, so slipping a

[4]The cross assembler's dependence on a brutishly large C compiler precludes any ability to guarantee its safety. Its principal use is to assemble, with human verification of every instruction, the self-hosted assembler for the first time.

hardware backdoor into the CPU unnoticed would be particularly difficult.[5] Finding transistors with desirable characteristics for CPUs might not be possible now. For example, the MOnSter 6502 is an 8-bit CPU containing 3 218 transistors, but it can only operate to 50 kHz due to component constraints [25].

7400-series and other **glue logic** have largely been discontinued. NAND gates and inverters aren't a problem to find, but the famed 74181 four-bit arithmetic logic unit is gone, the 74150 sixteen-to-one multiplexer is gone, etc. Most remaining chips have slow specifications, obsolete supply voltages, limited temperature ranges, through-hole packages, and/or single sources. Four-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. But a small number of 7400 parts are offered with very fast speeds, such as the SN74AUC series.

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 as of early 2020 when I needed to commit to a logic family. 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 present 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.

EPROM with a parallel interface apparently comes from only one company as of 2020. 45 ns access time is available, but requires a 5V supply. Data retention was 10 years in vendor advertisements, but omitted from datasheets. [4] and [5] describe a CPU that uses EPROM as its principal logic family.

EEPROM as fast as 70 ns is available with a parallel interface. Data retention is typically 10 years, but I have seen 100 years claimed for some pieces.

NOR flash with a parallel interface is suitable for combinational logic, offering speeds as fast as 55 ns. Storage density is not as extraordinary as NAND flash, but 128Mi $\times$ 8 configurations are well represented by two manufacturers as of early 2020. Data retention is typically 10 to 20 years, so these devices must be periodically refreshed by means of additional components or temporary removal. 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. Even so, NOR flash is usually considered reliable enough that external error-correcting code is not employed.

Although NOR flash's access time is much slower than SRAM's, its storage density can be much greater. For applications that can use large lookup tables directly instead of multiple-step algorithms, NOR flash may be faster than SRAM for uses like finding sines or logs of single-precision floats when high throughput is needed.

[5]One possible backdoor would be to install several RF retro-reflectors like NSA's RAGEMASTER [27] in parallel, or a single retro-reflector in combination with a software driver.

NOR flash may also be useful for building finite-state machines such as peripheral controllers or boot logic for loading firmware. Here again, a servicing mechanism would need to exist at the point of use on account of NOR flash's limited retention time.

NOR flash with a serial interface can be used as secondary storage for system firmware. At power-up, hardwired logic would load the CPU's firmware from the NOR flash. The firmware would need reflashing about every ten years to prevent bit rot.

NAND flash is available with parallel interfaces, but data and address lines are shared. These devices aren't directly usable as combinational logic. Also, NAND flash has a high enough error rate that external error correction is considered mandatory.

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 [26], as well as to cosmic ray and package decay soft errors. The 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 simple parallel interface we need for combinational logic. We can substitute SRAM logic for ROM logic, except SRAM will forget its logic tables when powered off. To use SRAM for logic, an additional circuit called a *firmware loader* must be present to load any needed tables when power is applied. In addition to increasing the number of components, the firmware loader's presence increases capacitance at the SRAM pins it needs to access. This added capacitance may require a compensating reduction in CPU speed.

As a logic family, SRAM's main selling point is its speed, due in part to SRAM's high computational expressivity. Even a tiny SRAM has at least 16 address pins for inputs, allowing much more computation per part than trivial glue logic gates. An access time of 10 ns is typical for asynchronous SRAM, with 8 and 7 ns obtainable at modest price increases. For synchronous SRAM, 5.5 ns access time is typical. Price is roughly 600 times that of DRAM as of 2020, around $1.50/Mibit. Of the sequential logic families that do not employ VLSI complex logic, SRAM offers the best combination of cost and computation speed available. As main memory, SRAM's decisive selling point is natural immunity from RowHammer and other shenanigans. Moreover, SRAM's simple parallel interface, separate connections for addresses and data, and predictable timing make it easily adaptable for use as logic.

It may seem contradictory to allow use of SRAM even as all other VLSI ICs (microprocessors, FPGAs, PLDs, ASICs, etc.) are disallowed. The distinction is one of complexity: SRAM is a perfectly regular array of addressable D flip-flops. It is not *complex logic* per my definition in Section 2. SRAM is fungible, generic, and standardized by JEDEC. SRAM's operation is simple and predictable, as evidenced by the difference between an SRAM datasheet (15–24 pages, depending on interface logic) and even a basic 8-bit microcontroller (hundreds of pages).

Compared to microprocessors, FPGAs, PLDs, and ASICs, it's probably very difficult to insert a supply-chain backdoor into an SRAM IC. The adversary would need to know what the SRAM's end use will be, what tables will be loaded into it, and where it will be placed within a circuit. The SRAM would need to touch enough identifiable data in context to be useful to an adversary—perhaps feasible for primary storage or registers, but likely infeasible inside the arithmetic logic unit.[6] Even if a backdoor could be inserted, it would need to add no more than a couple nanoseconds to the SRAM's operation, or the whole computer will fail. And assuring delivery of an SRAM backdoor to a single victim may well require altering the product worldwide.

Because any backdoor logic for an SRAM would be considerably more complex than the internal logic ordinarily present, it would be straightforward to detect via scanning electron microscopy of delayered ICs. And because the capacity of SRAM ICs is capped in order to manufacture them in older foundries (e.g. 40 nm), the best and latest imaging equipment would not be needed. Some forms of screening, such as an electrical noise analysis prior to soldering, may be effective without imaging due to SRAM's simplicity and external clocking.

There is one other motivation for permitting SRAM as a logic family: its use is already mandatory for primary storage. To build a computer with just one million bits of primary storage, we either need one million parts (more or less), or we will need some kind of VLSI storage. There is a security distinction between using SRAM for storage, where an IC would see a lot of data at rest, and using SRAM for computation, where an IC would see only a tiny amount of data in flight. So if it turns out that a malicious vendor can introduce an exploitable defect to an SRAM IC, it's much more likely to be a problem for primary storage SRAM than for SRAM that only implements logic.

Programmable logic devices, or **PLDs**, and **field programmable gate arrays**, or **FPGAs**, can implement CPUs, but they are not inspectable, not auditable, not fungible, ship with undocumented firmware 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 may also be the ultimate delivery vehicle for supply chain backdoors. They are easily reconfigured without leaving visible evidence of tampering. I would resist using PLDs and FPGAs in security-critical systems.

1.4 Dauug|36 as a Collection of Soldered Components

Dauug|36 is an open-source, 36-bit architecture for owner-built CPUs, controllers, and minicomputers. One of its purposes is to test whether the behavioral expectations and supply chain tamper resistance criteria in Section 1.2 can be achieved in a useful computer. Only maker-scale assembly tools are necessary, so this architecture can be implemented even in regions with minimal infrastructure and capital. No access is needed to a VLSI foundry. All that a builder will need is a bare circuit

[6] The bit-sliced ALU for the architecture of this paper only exposes six bits of data at a time to each IC.

board, a few hundred components, and some practice soldering. Because Dauug|36 does not contain a microprocessor (whether hard core or soft core), it isn't technically a *microcomputer*, so instead I call it a *minicomputer*. These two terms are differentiated in Section 2.

Even though Dauug|36 doesn't have a microprocessor, it offers 36-bit computing with paged virtual memory, preemptive multitasking, and a rich instruction set with more than 180 opcodes. Its physical logic family is synchronous static RAM containing read-only lookup tables, augmented with fast 7400-series logic. This union is tricky and somewhat fragile because these components were not designed for use together. The SRAM employed allows either 2.5 or 3.3 V power, while the 7400-series parts are designed for 1.8 V use. Fortunately, the glue logic datasheets permit up to 2.7 V, so by using these parts near their highest rated voltage, and the SRAM parts at their lowest, it's possible to build the machine. Only ICs rated for at least industrial use (−40 °C to +85 °C) and that can be hand-soldered appear in the architecture.

Another discontinuity between the 7400-series ICs and SRAM is how each interacts with the system clock. The SRAM ICs have registered inputs that can be synchronized by the system clock. The D flip-flops in the 7400 series have clock inputs, but their clock semantics are not consistent with SRAM's. The RAMs have *clock enable* inputs that the CPU's control decoder can use to tell each RAM what to do—or not to do anything—on the next rising edge. These clock enable inputs, in combination with other control inputs, also provide a clocked output enable capability for the RAMs. This is vital because many nodes in the circuit must be driven by different components at different times. In contrast, the stand-alone flip-flops, including the 16-bit D flip-flops that coordinate divergent data paths through the CPU, not only don't have clock enable inputs, but have unregistered output enable pins that aren't clocked at all. I would need many more words to explain all the reasons why this is bad, but the upshot is that SRAM and 7400 logic are difficult to use alongside each other in clocked circuits.[7]

Of the more than 50 logic series named with a 74- prefix, the most appealing for this architecture is 74AUC-, which stands for **A**dvanced **U**ltra-Low Voltage **C**MOS. I only know of one manufacturer for this series, and only a few component types are offered. Of these, my architecture only uses D flip-flops and trivial logic gates. My reason for selecting the 74AUC series is that no other has a propagation delay that can keep pace with prevalent synchronous SRAM ICs. Most SRAMs in the design offer clock-to-output-valid times of 5.5 ns and can compute any deterministic function of 18 input bits in that time. In contrast, the surrounding circuits built from glue logic will generally need more than one gate each, wherein the longest path should have delay comparable to one SRAM. The 74AUC trivial logic (AND, OR, NOT, NAND, NOR, XOR, and buffer) propagation delay is between 1.0 and 1.3 ns, so its speed mixes well with synchronous SRAM. Propagation delay for 74AUC flip-flops is 1.1 to 2.2 ns, so they are able to keep pace alongside 5.5 ns

[7]Note to fabs: Introducing 1- and 16-bit flip-flops with clock enable and clocked output enable to the 74AUC family would solve this problem. When you do, remember to offer more than ball grid array packages for those who solder by hand.

SRAM.

Figure 2 is a conceptual schematic of Dauug|36's data paths within the CPU and primary storage. Most paths in the drawing, including nodes 1–5, are 36 bits wide.[8] Rectangles indicate SRAMs, and triangles indicate parallel-wired D flip-flops in the same number as the number of bits. Wires that touch in the drawing intersect; the only non-intersecting crossover in the figure is where node 1 is drawn in front of node 2. To first order, "normal" information flow is through SRAMs, with the flip-flops offering alternative paths for special-use instructions.

Here is an example of "normal" information flow. Most instructions (more than 130) are arithmetic logic unit (ALU) instructions, where operands are fetched from the register file, then acted on by a three-layer SRAM ALU, then the computed result is returned to the register file. Although this circular path appears to take five clock cycles (fetch operands + 3 ALU + store result), the register file operation is interleaved such that the same clock edge that accepts write data from nodes 2 and 5 also initiates the next instruction's fetch to those nodes, while simultaneously disconnecting the γ RAMs from nodes 2 and 5. Thus for ALU instructions, the instruction cycle is always four clock cycles. For control simplicity, all other instructions also always take four clock cycles.

Another "normal" instruction flow is a load from or store to data memory, which is located in parallel with portions of the ALU. The privileged instructions `RDM` and `WDM` (read and write data memory) pass all address bits unmodified through the α RAMs, thereby bypassing the page table. These instructions can access data memory at any address. The non-privileged `LD` and `STO` (load and store) instructions that ordinary user programs would run pass the highest bits (the 4096-word page) of each address through the page table, while using two of the α RAMs to pass the offset within each page unmodified. These instructions can only access those pages that the operating system has mapped for the currently-executing program.

Most flip-flops in Figure 2 are for special-use instructions where "normal" information flow needs to be diverted. An example is the `WCM` (write code memory) instruction. The address to write at needs to move from the right operand register at node 5 to the code RAM's address pins at node 0. This is the purpose of 27-bit flip-flop "a" (address for code reads and writes) in the drawing. The data to be written comes from the left operand register at node 2, and passes through 36-bit flip-flop "w" (write code) to reach the code RAM's data pins at node 1.

The architecture divides the register file into identical copies kept in separate SRAM ICs named "left" and "right." This allows both operands of an instruction to be fetched on the same rising clock edge. When results are stored back to the registers, each bit is received from two separate pins in the γ RAMs in order to keep nodes 2 and 5 electrically distinct. The cost of this separation is that twice as many bits of storage are needed for the γ RAMs—a negligible drawback because the smallest RAMs on the market are already wide enough to drive both nodes.

The register file, page table, and return address stack are stratified by *user*, an eight-bit, system-global flip-flop indicating which of up to 256 programs is run-

[8]Node 0 and the unmarked nodes above it are 27 bits to match the architecture's 27-bit code address space.

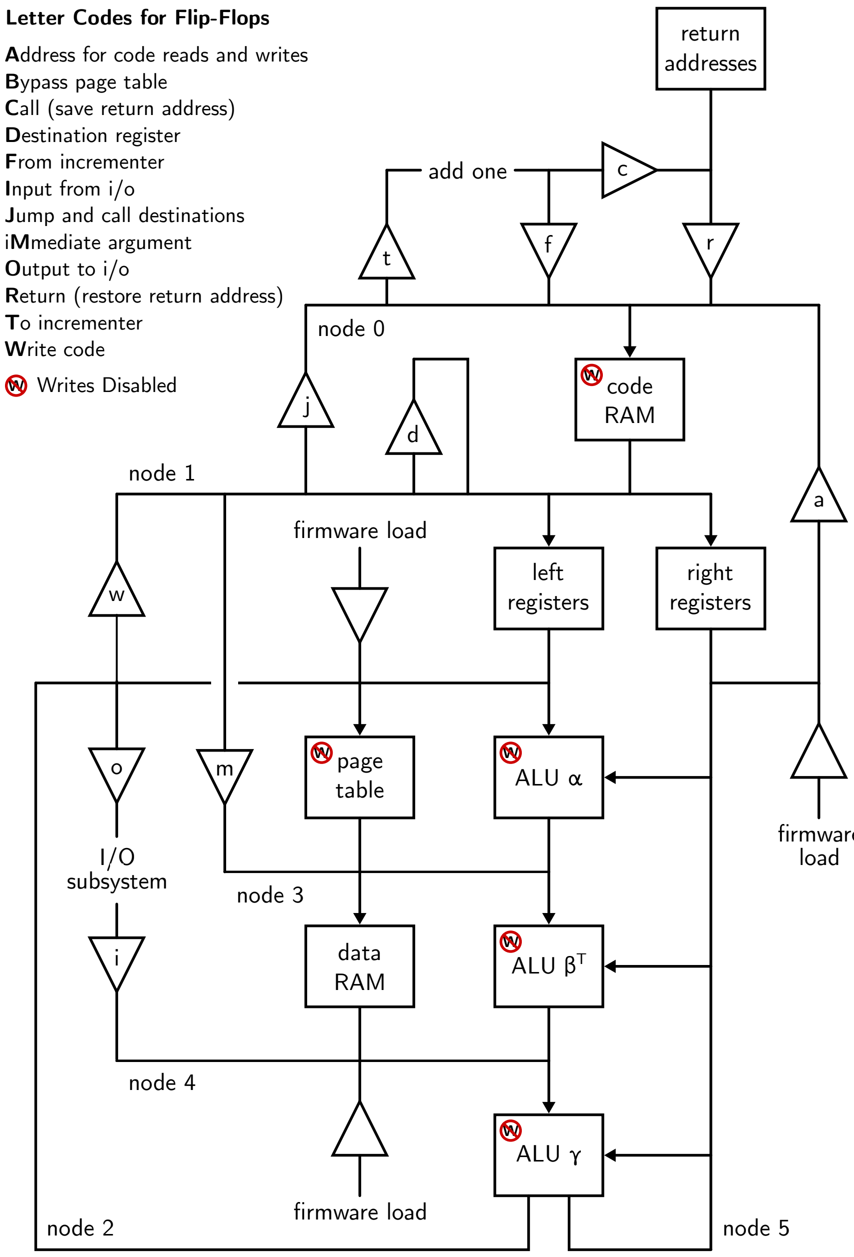


Figure 2: Principal Data Paths of the CPU and Memory Subsystem. All components are clocked and have output enables. Triangles and rectangles show D flip-flops and SRAM respectively. Most paths are 36 bits. SRAM address inputs are shown with arrows; the bottom SRAM connections exchange bidirectional data.

ning. User 0 is considered the *superuser*, the identity used by the operating system kernel. The remaining 255 numbers are available for use by user programs, although it's likely that a small number of these will be interruptible kernel subtasks. One example of a kernel subtask would be a routine that loads a program into code memory after other user programs have already started. The program loader needs to be interruptible so that real-time operating system (RTOS) constraints of already-running programs are not violated by the kernel pausing to load a large, new program all at once.

The 74AUC logic family does not offer counters or shift registers, and the corresponding parts in other 7400 series are too slow for Dauug|36. But the architecture needs both, so 74AUC trivial glue logic and 16-bit D flip-flops are used to build up circuits that do well enough. The program counter incrementer, which adds either 0 or 1 to a 27-bit code memory address, is built from 36 eight-pin packages, all of which are dual two-input AND or XOR gates. A 16-bit D flip-flop alongside five XOR gates supplies an 8-bit "up/down counter" that is actually a bidirectional Galois linear feedback shift register (LFSR). That's the stack pointer: the `CALL` and `RETURN` instructions respectively invoke its successor and predecessor operations. Another 16-bit D flip-flop implements a presettable "instruction counter," also an LFSR, that periodically interrupts the CPU for preemptive multitasking. This multitasking timer's preset value determines its interval and is stored in a shift register hacked from another 16-bit D flip-flop. Timer expiry is represented as 16 binary ones, which require 15 AND gates to detect.

In addition to SRAM and 7400-series logic, a handful of other components affect the architecture's security. The first is that a reservoir is needed for about 100 Mbit of firmware when the power is off. I opted for a NOR flash IC because it's the most transparent means I know that can store firmware at this scale. Punched paper or Mylar tape would increase transparency of operation, but at 2.54 mm per byte stored (the prevailing standard when punched tape was last popular), the tape would be 33 km long. Of the alternatives, I was most confident loading firmware from a solitary NOR flash IC that is electrically isolated such that the CPU cannot modify it.

Although programmable frequency synthesizer ICs containing micro-electromechanical system (MEMS) oscillators have largely replaced crystals for system clocks in new designs, Dauug|36 uses a crystal oscillator for two reasons. First, MEMS oscillators would be a perfect vector for introducing time-related logic bombs into an electronics supply chain. If I wanted to compromise a nation's critical infrastructure or military electronics and evade detection, tampering with the MEMS device supply chain would be my first idea. One manufacturer's white paper disavows culpability here, using the inaccurate claim that *producing secure code is beyond the ability of all semiconductor manufacturers*, and then suggests suing attackers under the Digital Millennium Copyright Act as a defense strategy for vulnerable silicon [18]. Second, I have found no data retention information for MEMS devices. How many years will a circuit run before a clock forgets what frequency to output? Dauug|36 plays it safe and uses a regular crystal oscillator. Likewise, Dauug|36's clock driver eschews recently-marketed clock distribution ICs that are extremely sophisticated and could conceal associated vulnerabilities.

Another class of security-problematic components is electrolytic capacitors. Because ISO 27000 [20] includes *availability* within its definition of *information security*, wear-out failure of electrolytic capacitors is not an option. What should be used instead of electrolytics is a field of active study. I intend to use multi-layer ceramic capacitors (MLCCs) in early Dauug|36 prototypes. I am less concerned about electrolytic capacitors in commodity power supplies that can be replaced easily, subject to their environment's sensitivity to outages and servicing.

2 Definitions and Abbreviations

The Dauug|36 architecture employs terms defined in [20], plus the following definitions for concepts that do not have well-known terms. Key abbreviations are also spelled out.

ALU. Arithmetic logic unit.

Arithmetic shift. Multiplication or division by a power of two, rounding towards $-\infty$ in the case of division. (This is *not* the customary definition.)

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

Clock cycle. The span of time between two consecutive rising edges of the system clock oscillator.

Code RAM. A primary storage SRAM that contains instructions that are fetched and executed by the CPU.

Complex logic. Digital electronic parts that, because of their complexity, may contain unseen exploitable defects.

CPU cycle. The amortized span of clock cycles required to execute a CPU instruction. In Dauug|36 systems, a CPU cycle is four clock cycles.

Data RAM. One or more primary storage SRAMs that are read or written by load or store instructions.

Discounted logic. Digital electronic parts that are unlikely to contain exploitable defects, as evidenced in a written assessment or other approved measure.

Firmware loader. Circuit that cold-boots a minicomputer, including logic that copies firmware from nonvolatile storage to SRAM logic elements and code memory.

FSM. Finite state machine.

Instruction. A 36-bit word in code memory consisting of an opcode with zero, one, two, or three operands.

Internal firewall. A boundary that isolates a portion of circuitry that is not solder-defined, such that exploits of defects within that portion cannot escape.

LFSR. Linear feedback shift register in Galois configuration.

Logical shift. A binary shift without any intent to multiply or divide. Unlike arithmetic shift, no overflow check is made.

Maker-scale assembly tools. Capital equipment for electronics assembly that can be made available to most technically knowledgeable builders.

MEMS oscillator. A frequency synthesizer IC referenced to an on-die microelectromechanical system (MEMS) resonator. Their displacement of crystal oscillators raises security concerns.

Mibit. $2^{20} = 1\,048\,576$ bits. In contrast, 1 Mbit is $1\,000\,000$ bits.

Microcomputer. According to custom and [10], a "computer system that utilizes a microprocessor as its central control and arithmetic element."

Microprocessor. A die that contains at least one complete CPU.

Minicomputer. A computer wherein all hardware logic that may contain exploitable defects is solder-defined, and all firmware is open-source.

Net. An electrically contiguous set of component pins. A net may communicate one bit at a time among electrical components.

Node. An intentional grouping of related nets. A node may communicate many bits, such as a word, at a time among electrical components.

Opcode. A nine-bit field in a CPU instruction that the control decoder uses to determine the necessary control signals to execute the instruction.

Opcode family. A group of opcodes that have the same purpose but need different control signals due to minor variations.

Operand. A field within an instruction containing a 9-bit register number, 18-bit integer constant, or 27-bit code address.

Overrange. A convenient synonym for out-of-range. In this paper, this word does not distinguish between overflow and underflow.

Primary storage. Non-cache memory that is accessible to or contains individual CPU instructions. Ordinarily termed "RAM" outside this paper.

PRNG. Pseudorandom number generator.

Program loader. Operating system code that copies a program to code memory, excludes forbidden privileged instructions, and completes link editing.

RAM. Within this paper, an informal abbreviation for SRAM.

SPN. Substitution-permutation network.

SRAM. Static RAM. Most SRAM ICs in this paper implements logic using read-only lookup tables. A few SRAMs provide read-write storage.

Solder-defined behavior. Intentional operational characteristics of solder-defined hardware when used with exclusively open-source firmware.

Solder-defined hardware. Digital electronics needing only maker-scale assembly tools to build, in which all complex logic components are discounted.

Tribble. A six-bit subword of a 36-bit word. This word envisions a "tri-nibble," a nibble that has been enlarged to the next multiple of three.

Word. A bit vector of a CPU architecture's natural size. Dauug|36's natural size is 36 bits for both code and data.

3 Dauug|36 as a Programmable Machine

Having considered the physical makeup of Dauug|36 in Section 1.4, I here describe the architecture's operational makeup. Ordinarily a description in this much detail would be excessive for peer literature, but in this case the information presented gives an approximation of the capabilities, performance, and cost that a solder-defined architecture can reach. This information is new to the literature and is not available elsewhere. Also from these details can be inferred a set of strategies for optimal use of the SRAM-with-74AUC logic family, as well as a benchmark for the best implementation to date.

3.1 State

The end deliverable in software development is a machine-executable set of rule-based state transitions. This section outlines what state is available to modify, thereby giving an outline of how software functions on Dauug|36 machines. This section's brevity advances an important point: the architecture is very simple in comparison with other contemporary architectures. This simplicity not only enables an SRAM-with-7400 architecture to exist, but allows the architecture to be readily understood even by a single person.

3.1.1 State for User Programs

From a non-privileged program's perspective, the internal state of the architecture is as follows.

Registers. Each program running has exclusive use of 512 registers. Every register is 36 bits wide.[9] All registers are considered to contain integers; there is no native floating-point representation. As a convenience, registers are declared with variable names and signages in assembly language, much like other languages declare signed and unsigned integers.

`N`(egative) flag. This one-bit flag can be set by the ALU to indicate that a result is negative. This flag's semantics are stratified by opcode, so the meaning of "negative" varies depending on what is appropriate. The `N`(egative) flag does not always match bit 35 of a numeric result because the ALU considers signage and overflow.

`R`(ange) flag. This one-bit flag can be set by the ALU to indicate that the true value of a numeric result does not fit in its destination register. Setting the `R`(ange) flag is intended to indicate an error condition; therefore, it is not cleared by subsequent ALU instructions, even if they succeed. Only a `CRF` (clear range flag) or `REVERT` (return from subroutine with flags unchanged) instruction can clear the `R`(ange) flag.

`T`(emporal) flag. This one-bit flag is a non-sticky counterpart of the `R`(ange) flag. The ALU sets this flag to indicate that the true result of the most recent arithmetic instruction does not fit in its destination register. Unlike the `R`(ange) flag, the `T`(emporal) flag must be tested immediately in order to be useful.

`Z`(ero) flag. This one-bit flag can be set by the ALU to indicate that the true value of a numeric result is zero. The `Z`(ero) flag is not always set when all 36 bits of a numeric result are zero because the ALU considers overflow and underflow.

[9]Simplification. To speed operand fetching, every register has a left copy and a right copy, each of which are 36 bits. In fancy circumstances, these "copies" can contain different values.

Data memory. A non-privileged program can address 8Mi words of virtual data memory. Here again, the word size is 36 bits. Data memory is always addressed with word granularity; it is not byte-addressable. Non-privileged programs cannot access physical memory directly; they are constrained by the page table.

Data memory edge cases. The following phenomena matter little to most user programs, but can be observed. Physical and virtual memory are managed in 4096-word pages. The architecture has no concept of page faults. Instead, all virtual addresses are guaranteed to be backed by physical memory. More than one virtual page can be mapped to any physical page. Virtual pages can be write-protected. Typically, an operating system will allocate and maintain a single all-zero physical page for the entire system, and map all unneeded virtual pages to it with write protection.

Stack. The stack is a circular buffer with 255 entries. Each entry consists of a 27-bit subroutine return address, alongside the four CPU flags (N, R, T, Z) at the time of the corresponding `CALL` instruction. The `RETURN` instruction branches to the return address at the "top" of the stack and discards the saved CPU flags. In this situation, the caller sees any flag modifications made by the subroutine. A `REVERT` instruction works like return, except the original flags are restored exactly as the caller had them.

What the stack is not. The only access to stack memory is via `CALL`, `RETURN`, and their variants. The `LD` (load) and `STO` (store) instructions and their variants have no access to the stack. The stack is stored on its own dedicated SRAM IC, and there is no way to obtain a pointer to a stack location. There is no random access to stack locations. The only information contained in the stack are return addresses and CPU flags; program data such as local variables are never on the stack.[10] Dauug|36 has no support for stack-based recursion—a small price to pay for completely preventing stack corruption and overflow.

Stack pointer. The stack pointer is not directly accessible to programs, but is adjusted by ± 1 by `CALL`, `RETURN`, and their variants.[11] If the pointer overflows its 255 possible states, the user program is likely to get lost, but there won't be a security concern. If the pointer underflows, the user program will `RETURN` to operating system code that, because the user program underflowed its stack, will safely terminate the user program.

Code memory. The behavior of user programs is determined by their representation in code memory; however, user programs have no access to this representation other than by fetching the next instruction to immediately execute. User programs cannot read or write code memory.

Instruction pointer. The instruction pointer is the address of the next instruction to be executed in code memory. The instruction pointer is write-only. It can be modified by variants of `CALL`, `RETURN`, and `JUMP`, but there is no electrical path that can enable programs to inspect it.

CPU possession. Although a program isn't conscious of when it's not running,

[10]Small to medium programs should keep most or all variables in their 512 available registers.

[11]Simplification. The stack pointer is a reversible 8-bit linear feedback shift register, not an up/down counter.

a program can surrender the remainder of its timeslice via the `YIELD` instruction. This is a state change.

3.1.2 State for Privileged Programs

It is recommended that the only Dauug|36 privileged programs be the operating system kernel that runs as user 0, and its interruptible subtasks that run as non-zero users. From the kernel's perspective, the internal state of the architecture is as stated for non-privileged programs, plus the following privileged state:

Code memory. Up to 4Mi words of code memory can be installed. Privileged programs can read from and write to all of it. The electrical path to read from code memory is too long to fit in four clock cycles, so a contiguous sequence of two instructions is needed to read. Information loss will occur if the `RCM1` and `RCM2` (read code memory 1 and 2) instructions are not contiguous. Writing a word to code memory takes one instruction.

Data memory. Privileged programs can read from and write to all physical data memory, bypassing the page table. Up to 8Mi words of physical data memory can be installed.

User. The *user* is an 8-bit flip-flop that indicates which of up to 256 programs is running. User 0 is reserved for the operating system kernel, also called the "superuser." User 0 is electrically immune to preemption by the multitasking timer.

The user number electrically stratifies the registers, page table, and stack into separate partitions for each program. For example, every program can access 512 registers (9 address bits), and there can be 256 programs (8 address bits). Thus the architecture has 131 072 registers, which are chosen via 17 address bits. Likewise, each program has 2048 page table entries, for a total page table size of 512Ki words, and 256 stack entries, for a total stack size of 64Ki words.[12]

Page table. The page table, which can be read and written by privileged programs, maps virtual pages to physical pages in data memory. Section 3.4 offers more specifics.

Privilege mode. The contents of the user register can be selectively masked with all zeros, enabling the kernel to do housekeeping on other programs' registers, stacks, and page tables. Table 6 (p. 39) shows this relationship. The `PRIV` instruction dynamically substitutes user 0 for whatever the user register contains, essentially entering a "superuser mode." The `NPRIV` instruction restores normal access to the registers, stacks, and page table, essentially entering a "normal mode." The `SETUP` instruction is a hybrid of `PRIV` and `NPRIV`, allowing the kernel to exchange data between its own registers and a different program's page table and stack.

Timer setpoint. Preemptive multitasking is achieved by forcibly interrupting the CPU after a user program has executed a certain number of instructions. This number of user instructions is effectively a multitasking timer interval, which can vary between 2 and 65 535. Supporting a larger setpoint in hardware is not necessary because the round-trip time from user to kernel to user is just 20 clock cycles—the

[12]Because of LFSR limitations, only stack locations 1–255 are available to each user.

time required for 5 CPU instructions. Thus the overhead incurred to extend a user's CPU allocation beyond one time slice is less than 0.01%.

The timer itself is implemented as a 16-bit LFSR, so the timer setpoint is essentially a starting point within the sequence. It is not a linear count, but must be calculated. For example, the setpoint for the maximum interval of 65 535 instructions turns out to be 1010 1111 1111 0111.

Timer progress. The multitasking timer LFSR's present 16-bit value is not observable; however, control of the CPU will return to the kernel when the LFSR value reaches 65 535.

3.2 Firmware

Other than D flip-flops, the architecture has 200-or-so logic gates comprising inverters and two-input AND, NAND, OR, NOR, and XOR. All are used for control; no basic gates appear in the data path. Instead, all Dauug|36 computation is achieved by lookup tables that are stored in every SRAM that does computation. These lookup tables are called *firmware*, and must be copied from nonvolatile storage (serial NOR flash) to the SRAMs at power-up. Once the firmware is in place, all writes to these SRAMs are disabled until power is removed. Importantly for security, the CPU is electrically unable to alter the contents of either the nonvolatile storage or any SRAM used as logic.

When discussing Dauug|36 as a programmable machine, firmware almost doesn't have to be mentioned because usually firmware can be abstracted away. Instead, one can say that the `LD` (load) instruction does such and such, `TIMER` does something else, etc. All these behaviors can be implemented with basic gates instead of SRAM, although size, cost, power, and execution time would increase intolerably. Writers of user programs or even OS kernels can look at the CPU's semantics and behavior as merely these and ignore firmware behind the scenes. Yet firmware is central to implementing the architecture and evaluating its security.

Firmware is also the language in which the instruction set is extended. For example as of early 2024, there is no library code for integer division, and the person responsible for writing division routines will, by extending the firmware, create new CPU instructions that harness the ALU's potential to accelerate division.

Table 5 inventories the 23 SRAMs that firmware is loaded into. Of these, all but one are exclusively for firmware. After the firmware loader's work is done and before the operating system is loaded, further writes to these RAMs are disabled. The exception is the code RAM. The firmware loader's final task is to transfer a short program called the *boot loader* to the code RAM and start it. It is the boot loader's job to load the operating system from external media and start that. Because the boot loader is maintained in the same nonvolatile storage as the firmware and is brought into the system by the firmware loader, the boot loader is deemed to be firmware.

Two 128Ki $\times$ 36 SRAMs are wired in tandem for 128Ki $\times$ 72 and form the *instruction decoder*. Most of this memory is unused, but what is used accepts a nine-bit opcode, a two-bit cycle indicator (there are four clock cycles in an instruction),

Table 5: SRAMs that contain firmware. The code RAM only contains firmware until the operating system has been loaded. The other SRAMs are used solely as logic and cannot be modified after firmware is loaded. The address space supports 128Mi words of code RAM, but present SRAM ICs are much smaller.

Name	Qty.	Organization	Purpose
C	1	up to 4Mi × 36	boot loader in code RAM
D	2	128Ki × 36	instruction decoder
α	6	256Ki × 18	α (first) layer of ALU
β	6	256Ki × 18	β (second) layer of ALU
γ	6	256Ki × 18	γ (third) layer of ALU
θ	1	256Ki × 18	subword look-across and carry propagation
ζ	1	256Ki × 18	CPU flag generation

and two “hijack” bits that indicate when a multitasking context switch is occurring. The output is 44 control bits that, as they change during the four clock cycles of an instruction, tell the circuit board how to execute the instruction.

The 20 remaining SRAMs that contain firmware implement the arithmetic logic unit. The main section of the ALU uses 18 RAMs, arranged in three layers of six RAMs that operate in parallel as will be shown in Figure 12 (p. 53). The ALU uses a bit-slicing scheme, where each 36-bit operand is split into six-bit subwords, also called *tribbles*. The ALU uses six RAMs at a time on these six-bit subwords to advance a 36-bit word through one clock cycle of computation. The operation done by each RAM is very small, such as add two tribbles, logical AND of two tribbles, etc. Because the ALU has three layers, it takes three clock cycles for a calculation to proceed fully through the ALU. The layers are named α (alpha), β (beta), and γ (gamma) after their sequential order.

The ALU’s six-bit slices are logically connected for carry propagation and other needs by an SRAM named θ (theta). Another SRAM, ζ (zeta), combines many inputs to produce the four CPU flags `N`(egative), `R`(ange), `T`(emporal), and `Z`(ero).

3.3 Instruction Format

Every Dauug|36 instruction is encoded as a 36-bit word. There are four possible formats; these are shown in Figure 3 and are as similar to each other as practical. Each format is named after the number of fields it contains; for example, Format III has three fields.

Format IV is the most common and is used for ALU instructions. This format provides a 9-bit destination register, left operand register, and right operand register. Except for the always-leftmost opcode, the field order purposely conforms to infix arithmetic in the form `c = a + b`.

Format III is for instructions that move immediate values to registers, such

Format I	opcode	ignored		
Format II	opcode	branch target in code memory		
Format III	opcode	dest. register	immediate value	
Format IV	opcode	dest. register	left register	right register
	bits 35–27	bits 26–18	bits 17–9	bits 8–0

Figure 3: Dauug|36 CPU instruction formats. Every field is a multiple of nine bits.

reserved	virtual page	offset
bits 35–23	bits 22–12	bits 11–00

Figure 4: Virtual address format for data memory. The virtual page field's width matches Figure 5, and can grow in the future from the reserved field.

as `IMN` (immediate negative). Eighteen bits can be specified, which cover many frequently-used integer constants. Constants that don't fit in 18 bits require three instructions to load, typically `IMH` (immediate high), `IMP` (immediate positive), and `OR` (logical OR).

Format II is for branch instructions such as `JUMP` and `CALL`. All Dauug|36 branch destinations that aren't from the stack (like `RETURN`) are absolute addresses. This allows the operating system kernel to police all access to code memory, even though the architecture includes no code memory protection hardware.

Format I is the simplest. The CPU is guaranteed to ignore all bits except for the opcode, which represents an instruction like `CRF` (clear range flag), `NOP` (no operation), `RETURN`, etc.

3.4 Address Formats, Memory Protection, and Memory Capacity

Figures 4, 5, and 6 show the virtual and physical address formats for data memory and their relationship. In brief, the page table is indexed by virtual page numbers to look up physical page numbers by direct substitution. Each page contains 4096 words. Separation between programs is achieved by using eight bits of each virtual page number to identify the owning program. Ownership only applies to virtual pages; all physical pages are "owned" by the kernel. To be secure, this mechanism requires *every* page table entry to be valid, which can be achieved by filling unneeded entries with a single write-protected "zero page" for the entire system.

Physical pages may not all be contiguous. The present netlist permits one or

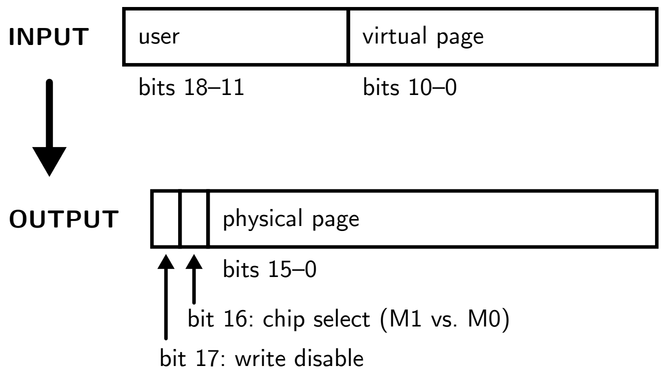


Figure 5: Page table RAM's input and output bit assignments

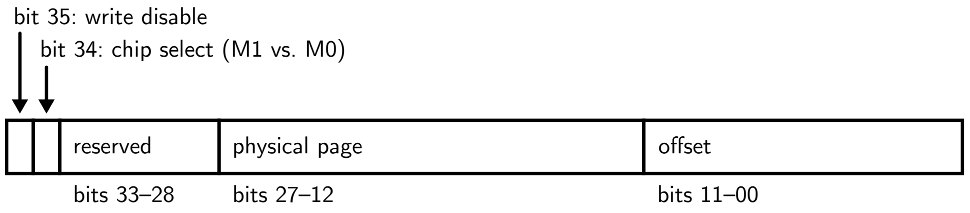


Figure 6: Physical address format for data memory. The reserved field can be reallocated if SRAM densities improve significantly to expand the number of physical pages.

two data memory SRAM ICs using bit 34 as a chip select bit. Bit 35 is a write disable flag, effectively causing every memory location to have a writable and non-writable representation. This limits the physical address space to 35 bits at present, although it would be easy to semantically attach write protection to virtual memory instead, thereby increasing the physical data memory address space to 36 bits.

On actual machines, the amount of data memory will be much smaller. Although 288 Mibit SRAMs are sold organized as 8Mi $\times$ 36, and the architecture supports two, these parts are only available in ball grid array packages. These are unrealistic to place or solder by hand. Makers with modest capabilities will be limited to 144 Mibit (4Mi $\times$ 36) ICs with linear pins. Parts larger than 288 Mibit are not sold as of early 2024. Although DRAM lore attributes this to SRAM's high comparative transistor count per cell, that isn't why. Process technology for considerably larger SRAMs does exist, yet part sizes have not increased since 2015. SRAMs are also more expensive than DRAMs, again due primarily to market factors. Not only do SRAMs use more transistors per bit than DRAM, but SRAM prices are around 100 times more *per transistor*.

To some degree, foundries are ready to scale up if demand emerges for larger SRAM ICs, but challenges have arisen shrinking SRAM cells for nonplanar processes [33]. SRAM cells as small as 0.02 μm^2 can be made at scale, suggesting that a 4608 Mibit, 128Mi $\times$ 36 SRAM IC can be built on a 1 cm^2 die. But its static power consumption and self-heating will be high, and expensive GPUs for training AI models will compete with the RAM for manufacturing capacity.

3.5 Logic

The firmware implements one opcode for each of the 16 bitwise boolean functions of two words:

- all zeros, all ones
- AND, OR, NAND, NOR, XOR, XNOR
- exactly left, NOT left, exactly right, NOT right
- left AND NOT right, right AND NOT left, left OR NOT right, right OR NOT left

Some of these are not helpful, but AND NOT in particular is useful to have as one instruction.

The architecture implements left and right logical shifts and rotations, each as one instruction. An encoding must be used to represent the shift or rotation displacement because this information is needed within each of the ALU's six-bit slices. For left shifts and rotations, the encoding is $e = 010101010101_8 \times n$, where e is the encoding placed in the right operand, and $0 \leq n < 36$ is the number of positions to shift or rotate. In other words, the number of bits to shift is replicated to all six tribbles of the encoding. For right rotations or shifts, use encoding $e = 010101010101_8 \times (36-n)$, where $1 \leq n < 36$. In other words, the equivalent number of bits to rotate *left* is replicated to each tribble. To shift or rotate right by zero bits, use $e = 0$.

If an encoded displacement is not in a register when it is needed for a shift or rotation, it must be prepared on the fly. This would cause the rotation or shift to require two instructions instead of one. Although the described encodings suggest multiplication, that would take a while. Special opcodes exist to encode the displacement in one instruction.

3.6 Arithmetic

3.6.1 Single-Instruction Arithmetic

The basic operations add, subtract, arithmetic shift left, and arithmetic shift right are available for 36-bit signed and unsigned integers. These use one instruction each, assuming that shift and rotation displacements are already encoded per Section 3.5. I have improved these instructions' semantics relative to prevalent CPUs.

For most CPUs, arithmetic instructions only offer homogeneous signage. Either two unsigned integers produce an unsigned result, or two signed integers produce a signed result. This covers only two of eight possible situations. Dauug|36 addition and subtraction support heterogeneous signage, so all of the following automatically work correctly:

- unsigned = unsigned + unsigned
- unsigned = unsigned + signed
- unsigned = signed + unsigned
- unsigned = signed + signed
- signed = unsigned + unsigned
- signed = unsigned + signed
- signed = signed + unsigned
- signed = signed + signed

The above stratification by signedness is achieved by implementing eight opcodes for addition. The assembler selects the correct opcode based on whether each register is declared as unsigned or signed. This frees the programmer from needing to think hard about signedness or worry about conflicts because all that matters is that the result fits in the destination register and is interpreted with the correct signage.

Most computers detect out-of-range additive operations using a scheme that was modern 60 years ago that checks a *carry flag* or *overflow flag* for fully-unsigned and fully-signed operations respectively [13]. There are a few problems with these scheme, starting with the issue that neither flag is decisive by itself even for homogeneous-signage addition and subtraction. They *do not work* for heterogeneous addition and subtraction,[13] nor do they know anything about left or right arithmetic shifts. On top of this, many programmers—and programming languages—recklessly don't even bother to check for out-of-range results.

Dauug|36 presses for a higher standard. The architecture doesn't have a carry or overflow flag because they seek the wrong information. The right question is, "is the stored result of a recent calculation incorrect?" Dauug|36 supplies the answer via a CPU flag named `R`(ange). This flag is turned on whenever the result of an addition or subtraction does not fit in its destination register. The `R`(ange) flag is also turned on when the result of an arithmetic shift is not exactly equivalent to multiplying or dividing by the corresponding power of two.[14] Further calculations do not turn `R`(ange) off; the flag is only cleared by the `CRF` (clear range flag) and `REVERT` (subroutine return with original flags) instructions.

When testing the `R`(ange) flag, the programmer doesn't need to know anything about operand or destination signage. The `R`(ange) flag is turned on if and only if it is not already on and a result does not fit in the destination register. In addition to working for all eight signage variations for addition and all eight for subtraction, the flag works for all four variants of arithmetic shift left (shift signed/unsigned register into signed/unsigned result) and all four for arithmetic shift right.

When an out-of-range condition needs to be detected for the last instruction only, Dauug|36 supplies a `T`(emporal) flag. Any computation that may turn `R`(ange) on

[13]For example, whether a 4-bit sum $1000_2 + 1000_2$ carries, overflows, or is zero depends on the operands' signages.

[14]Right arithmetic shift quotients are rounded toward negative infinity.

will turn the `T`(emporal) flag on or off to indicate whether the destination is able to hold the full result.

3.6.2 Short Multiplication

Dauug|36 does not include a hardware multiplier. In the absence of heavy number crunching, asymmetric cryptography, or artificial intelligence, most computer multiplication is for the purpose of computing memory addresses of array elements. The hardware requirements to do this efficiently are quite minimal.

Dauug|36 defines *short multiplication* as unsigned multiplication where one factor is known to be less than 64, and the result is known to fit in 36 bits. The small factor will often be size of an array element in words, and is ordinarily known when the program is assembled. This factor is replicated across the six tribbles of a word, e.g. by multiplying by 010101010101_8. So if we're multiplying by $19 = 23_8$ for example, a control word of 232323232323_8 is used.

With the large factor and encoded small factor in hand, the architecture's `ML` (multiply low) and `MH` (multiply high) instructions are invoked. Both multiply each factor across the bit slices of the ALU, with six bits taken from the large factor, and six from the expanded small factor. The reason for using two different instructions is that when we multiply six bits by six bits, the result will be twelve bits, but each bit slice only fits six bits. `ML` and `MH` multiply the same numbers, except `ML` returns the six least-significant bits of the product, and `MH` returns the six most-significant.

Something else needs to happen before the two "partial products" are added: the most-significant bits need to move six bit positions leftward to obtain their correct place values. This isn't exactly what happens. Instead, `MH` *rotates* its result six bits left. This result will be incorrect if the six wrapped bits are anything but all zeros. `MH` knows about this and will set the `R`(ange) and `T`(emporal) flags if this error occurs. But considering that the architecture won't have more than 2^{36} words of data memory under any circumstance, we already know that for computing array offsets, the results of `MH` and `ML` added together will always fit in their destination. Thus the architecture offers short multiplication in just three instructions (`MH`, `ML`, `A`), or 12 clock cycles, with no special hardware.

3.6.3 Long Multiplication

By long multiplication, I mean unsigned multiplication of two words to produce a 72-bit result. This has to be done in software and takes 35 instructions (140 clock cycles), plus a probable `CALL` and `RETURN`. Many embedded systems rarely need to do long multiplication. Getting this as fast as 35 instructions requires elaborate ALU firmware tricks to create several oddball instructions. A full explanation is beyond the scope of this paper and is available in the system documentation, so here are illustrative excerpts for two of the instructions used:

`DSL` (double shift left)

```
c = a dsl b
```

`DSL` adds the `T`(emporal) flag with wrapping to `b`, and then shifts the sum left six bits. The six vacated bits are filled using the six leftmost bits of `a`. The result is written to `c`.

`MHL4` (multiply high and low, tribble 4)
`c = a mhl4 b`
`MHL4` replicates tribble 4 (bits 24–29) of `a` across all six subwords, and then multiplies them pairwise with the tribbles of `b`. In order to fit the six 12-bit results into `c`, the six most significant bits of each product are written to the left copy of `c`, and the six least significant bits of each product are written to the right copy of `c`. No flags are changed.

3.6.4 Adding a Hardware Multiplier to the Architecture

In most cases, there is no performance advantage in adding a fast SRAM multiplier to Dauug|36. Such a multiplier would more than double the size of the whole computer for a seldom-used instruction. Having said that, I present an elaborate explanation of how to multiply quickly using SRAM as logic in chapter 10 of [2]. I also wrote a Rust program [1] to generate and test SRAM multipliers for any combination of factor signages, including an “either” option that allows a factor to be declared unsigned or signed at the time of multiplication. Factor lengths need not be identical and can be several thousand bits.

Although there are no plans to add a hardware multiplier to Dauug|36, its cost and performance have been calculated. A 36-bit, either-signage multiplier would require 49 SRAMs, all of which are 256Ki $\times$ 18. Dozens of 16-bit D flip-flops would be needed to load 91 164 672 bits of firmware into these SRAMs. A 36-bit multiplication with a 72-bit result, with either factor independently signed or unsigned, would take five clock cycles.[15] Instruction decoding, operand fetching, and storing the result must also happen, so a hardware-multiplier instruction will require as much time as two regular instructions.

3.6.5 Division

Division is not implemented yet, although assembly subroutines that quickly divide by 24 and 60 have been tested. A triage scheme to leverage Dauug|36 for unsigned integer division appears in [3]. The method stratifies division into six cases, then uses the ALU’s ability to (sometimes) do more than one task per instruction to find the quotient and remainder.

- If the numerator is less than 64, the ALU directly looks up the quotient.
- If the denominator is a power of two, the process becomes a right arithmetic shift.

[15]The multiplier can complete a 36-bit multiplication every clock cycle, but its path length is five clock cycles.

- If the denominator is less than 64 and is not one of (29, 37, 53, 58, 59, 61), a short series of shifts and adds completes the division.
- If the denominator is less than 4096, its 36-bit reciprocal is recalled from the firmware, and the division is completed by long multiplication.
- If the denominator is larger, binary long division with non-restoring subtraction is done.[16]
- If the denominator is zero, the numerator is copied to the quotient and remainder.[17]

3.7 Advanced ALU Instructions

Because Dauug|36 computers are for building at millimeter scale using off-the-shelf components, they face a performance ceiling that would be difficult to raise. Moreover, the process technology used in today's SRAM ICs is already fifteen years old, making SRAM capacity for code memory lackluster. These missed capabilities are partially restored by making use of the ALU's distributed construction: because the ALU has 20 SRAM ICs, some of these ICs can be used for different tasks within the same instruction cycle.

"Advanced" instructions that speed up the architecture have two general flavors. Some instructions are effectively catenations of instructions that use different parts of the ALU. A trivial example is `NAND`, which does a bitwise AND in the ALU's α layer and a bitwise NOT in the β layer. Other instructions exploit the ALU's high interconnectedness to do tasks in a fixed number of clock cycles that other architectures use a loop for. For instance, a hash function for associative arrays uses only one instruction per input word because Dauug|36's ALU is a special case of a substitution-permutation network. Ordinary ALUs would be envious of the `MIX` and `XIM` instructions.

This section introduces some noteworthy ALU instructions that speed up computation and reduce code size. For operational details, the reader may consult [2] and [3].

3.7.1 Reverse Subtraction

Because subtraction is not commutative, operand order makes a difference. On ordinary CPUs, the desired sign is obtained by swapping operands if necessary. Swapping is not always possible on Dauug|36 because occasionally the contents of a register will depend on whether it is fetched as the left operand or as the right operand. This is the case, for example, after Section 3.6.3's `MHL4` instruction, or if a hardware multiplier is added that atomically stores a 72-bit product. For this reason, all `S` (subtract) opcodes are offered with `RS` (reverse subtract) variants.

[16]The architecture's ability to count leading zeros without using loops considerably speeds long division.

[17]Dividing by zero is not wrong, but there are infinitely many correct quotients and only one correct remainder.

3.7.2 Minimum and Maximum

Although `MIN` and `MAX` are not advanced ideas, most architectures employ more than one instruction, including a branch, to implement them. Dauug|36 boasts eight signage-stratified `MIN` instructions with full range checking, and eight matching `MAX` instructions.

3.7.3 Nudge Integer to Offset from Power of Two

Sometimes one needs to discard the rightmost bits of an integer and replace them with a fixed substitute. This is equivalent to adjusting n such that $n \pmod{p} = r$ without changing $\lfloor n \div p \rfloor$, where n, p, and r are non-negative integers, p is a power of two, and $r < p$. Although this sounds esoteric, it's useful for hopping a pointer among the fields of a power-of-two-aligned structure, and Dauug|36 can do it in a single instruction. When presented n as a left operand and $p + r$ as a right operand, the `NUDGE` instruction replaces the rightmost $\log_2 p$ bits of n with the bits of r and returns the result.

3.7.4 Population Count (Hamming Weight)

The number of 1 bits in a 36-bit word can be counted in a sequence of two instructions: a variant of `STUN` (stacked unary) that will eventually be available as an assembler macro named `HAM1` (Hamming weight 1), and an opcode named `HAM2`.

3.7.5 Tribble Swizzling

A set of commonly-occurring rearrangements of a register's 6-bit subwords is accessible via the `SWIZ` instruction. The most-often used of these is rearrangement 0, which copies tribble 0 (the six least-significant bits) of a word to the other five. Thus the expression `x SWIZ 0` is equivalent to multiplying the six least-significant bits of `x` by 010101010101_8.

3.7.6 Transposing XOR

As data passes through the bit-sliced ALU, the slices are not kept as-is. The six α RAMs transmit one bit to each of the six β RAMs, and the six β RAMs transmit one bit each to the γ RAMs. (See Figure 13, p. 54.) The firmware makes extensive use of these wiring transpositions. The `TXOR` (transposing XOR) instruction lets the programmer access the transpositions directly.

In the instruction `c = a txor b`, the bits of `b` are transposed to compute $\mathrm{b}^\top$ by relocating the ith bit of tribble j to the jth bit of tribble i for all $i, j \in \{0...5\}$. The bitwise XOR of `a` with $\mathrm{b}^\top$ is then written to destination `c`.[18]

`TXOR` can be used to obtain the transpose of a register. To do this, use zero for `a`.

[18]Simplification. The netlist implements $(\mathtt{a}^\top \oplus \mathtt{b})^\top$ instead of $\mathtt{a} \oplus \mathtt{b}^\top$.

3.7.7 Bit Permutations

Because the SRAMs in the bit-sliced ALU operate on six-bit subwords, these RAMs can be used to look up permutations of six-bit tribbles taken from the left operand. Which permutation to apply comes from the corresponding right operand tribble, and because there are only 64 possible values, the architecture can only implement 64 of the $6! = 720$ permutations of six bits. The 64 implemented were chosen for their likelihood of programmer interest and for being a complete set such that each of the 720 permutations can be reached as a catenation of two implemented permutations.

The full story about bit permutations is lengthy, but the outcome for programmers is that some of the simpler word permutations, such as reversing the order of all 36 bits, can be done in one CPU instruction. It is also known that *any* needed permutation of 36 bits can be achieved in five or fewer CPU instructions. These instructions are named `PIT` (permute inside tribbles, which is done in the ALU's γ layer), `PAT` (permute across tribbles, within the transposed β layer), and `PAIT` (permute across and inside tribbles, in the transposed β layer followed by γ).

3.7.8 Substitution-Permutation Network Instructions

The full interconnectedness between the RAMs in the ALU's α and β layers, and subsequently the β and γ layers, allow the ALU to function as a substitution-permutation network. At each of the three layers, tribbles from the left operand are replaced by S-boxes selected by tribbles from the right operand. The instruction that does this is `MIX`, and its inverse is `XIM`. So for any 36-bit plaintext x and 36-bit key k:

$$x = (x \texttt{ MIX } k) \texttt{ XIM } k = (x \texttt{ XIM } k) \texttt{ MIX } k$$

The S-boxes for `MIX` and `XIM` derive algorithmically from nothing-up-my-sleeve number $\sqrt{2}$. Uses for `MIX` and `XIM` may include the following:

- As a hash function for associative arrays. A running hash beginning with an undisclosed random seed is used as the left operand for either `MIX` or `XIM`. The right operand is the next word to be included in the hash. This scheme uses one instruction per word hashed.

- As the second instruction of a two-instruction pseudorandom number generator (PRNG). The first instruction is a `STUN` (stacked unary) configured as a linear feedback shift register starting from a non-zero random seed. The output of `STUN` is fed back to itself to produce successive keys that become the right operand of either `MIX` or `XIM`. The first `MIX` or `XIM` takes its left operand from another random seed, and its output is fed back as the PRNG output and next left operand. Although this PRNG is not cryptographically secure, it has superb statistical properties and passes all PRNG tests in [9].

- As a round function for a 36-bit block cipher. Among this idea's problems are an anemic block size and S-boxes that were never optimized against differential cryptanalysis.

- As an element of a round function for a cipher with larger blocks, such as 144 bits. Such a cipher is likely to be slow, but not as slow as if `MIX` and `XIM` were not present. It would also be incompatible with non-Dauug|36 hardware. Because of `MIX` and `XIM`'s heavy S-box use and word transpositions, these instructions execute faster on Dauug|36 machines than they can be simulated on multi-GHz microprocessors. Any new cipher designed specifically for Dauug|36 will require public vetting by expert cryptographers.

3.7.9 Stacked Unary Instructions

As shown in Figure 12 (p. 53), the ALU copies the right operand of each instruction to all three ALU layers unmodified, while the left operand progresses through the three layers with modifications. By using the left operand as an input to a unary function, and the right operand to identify one of 64 possible unary functions, a bank of unary functions can be designed that use all three layers of the ALU as a "stacked" unit. These are available via the `STUN` (stacked unary) opcode family. Notable implemented unary functions include:

- Compute the absolute value of a number in two instructions without branching.
- Compute the absolute value of a number with magnitude $< 2^{30}$ in one instruction.
- Signum: -1, 0, or $+1$ based on $x < 0$, $x = 0$, or $x > 0$, in one instruction.
- Range-checked conversions of a shift or rotate displacement to a shift control word.
- Population count (Hamming weight): the number of ones in a word, in two instructions.
- Identify leading or trailing zeros or ones in one instruction.
- Determine the parity of a word in one instruction.
- Increment or decrement a bit-reversed integer modulo 2^{36} in one instruction. Carry propagation is from left to right. This makes canonical trailing-bit manipulation that other CPUs can do available as *leading-bit* operations on Dauug|36. There are also other uses.
- Implement a 36-bit linear feedback shift register in one instruction.
- Efficiently implement 72-, 108-, and 144-bit linear feedback shift registers.
- Shift or rotate left or right by one bit through the `T`(emporal) flag.

3.7.10 Unusual Memory Instructions

The data memory and page table are electrically within the arithmetic logic unit. Certain instructions use these as a combined circuit. Three applications are worthy of mention. For each use, a non-privileged version for virtual memory and a privileged version for physical memory exists.

- `LDSTO` (load and store) and its privileged counterpart `RWDM` (read and write data memory) atomically read from and write to a single memory address in one instruction. They are useful for implementing semaphores in shared memory.

- `STO2` (store twice) and its privileged counterpart `WDM2` (write data memory twice) can store one value in two consecutive memory locations using one instruction. They are useful for zeroing large blocks of memory quickly.

- `ADDLD` (add then load) and its privileged counterpart `ADDRDM` (add then read data memory) approximate a one-instruction "base + offset" scheme for reading from memory. This means that a separate `A` instruction is not necessary to fetch memory at some offset from a pointer. The limitation is that carry propagation across tribbles isn't available before the memory is accessed, so certain criteria about the base and offset must be satisfied for these instructions to work correctly.

3.8 Multitasking

Dauug|36 multitasking is designed to have simple semantics, very fast context switches, and few components. Both cooperative and preemptive multitasking are supported. Their context switch process is identical because the control signal asserted by the `YIELD` instruction for cooperative multitasking feeds the same OR gate as the preemption timer's output.[19]

Because nearly all CPU state is stratified by user (Section 3.1.2), only 39 bits within the CPU must change to transition from one program to another. These are the:

- instruction pointer (27 bits)
- CPU flags N, R, T, Z (4 bits)
- D flip-flop "u," set by the `USER` instruction (8 bits)

Changing the instruction pointer is straightforward because every program already has a stack initialized for subroutine return addresses. Firmware to save the instruction pointer when a program is interrupted is almost identical to the `CALL` opcode, except no branch needs to occur—there is no "`CALL` to" address. Similarly, firmware that restores the instruction pointer when a program is resumed is

[19]The context switch is not immediate: two instructions beyond `YIELD` will execute before the CPU is relinquished. If this is a concern, these two instructions can be `NOP`s.

Table 6: CPU privilege modes. The active partition of stack, page table, and register memory is set by the `USER` instruction; however, a handful of AND gates can override the current partition with user 0, the superuser. Instructions `NPRIV`, `SETUP`, and `PRIV` switch between the privilege modes.

Privilege Mode	**Active Stack**	**Active Page Table**	**Active Registers**	**Multitasking Timer**
`NPRIV`	current `USER`	current `USER`	current `USER`	active
`SETUP`	current `USER`	current `USER`	superuser	inactive
`PRIV`	superuser	superuser	superuser	inactive

almost identical to `RETURN`. No hardware is added beyond what is already present to support `CALL` and `RETURN`.

Changing the CPU flags is almost free because the 27-bit instruction pointer is pushed onto a 36-bit-wide return address stack. For no extra hardware, the four CPU flags are also pushed on the stack, leaving five bits of each stack entry unused. The `CALL` opcode always pushes the flags with the instruction pointer, even though `RETURN` never restores them. A tiny amount of hardware must be added to restore the flags, specifically, half of a 16-bit D flip-flip. Firmware for an additional opcode `REVERT` is a near-copy of `RETURN`, except that the flags revert to their as-saved condition. The firmware that restores the instruction pointer is actually more like `REVERT` than `RETURN` because `REVERT` unlike `RETURN` restores the flags with the instruction pointer.

All that remains is to change the eight *user* bits that identify which program is running to the call stack, page table, and register file. The current user is subordinate to one of three "privilege modes" appearing in Table 6. Only `NPRIV` and `PRIV` modes apply to this discussion.[20] `NPRIV` mode, named after the `NPRIV` instruction, is when a user program is executing. During this time, the CPU sees the call stack, page table, and register file specified by the most recent `USER` instruction executed by the kernel. In `PRIV` mode, the CPU sees the stack, page table, and registers belonging to user 0, also called the superuser or kernel. This means the effect of the `USER` instruction is not immediate while in `PRIV` mode, but is deferred until the next `NPRIV` instruction.

The CPU does not transition directly from one user program to the next. Instead, it enters `PRIV` mode in between so that the kernel's scheduler can direct which program should run next. The transition from user program to kernel takes the duration of two instructions, which is eight clock cycles. The first four clock cycles work like `CALL`, saving the instruction pointer and flags on the user program's stack, and right at the end placing the CPU in `PRIV` mode. The second four clock cycles work like `REVERT`, loading the instruction pointer and flags from the kernel's stack.

[20]`SETUP` mode allows the kernel to initialize or sanitize a user program's stack before or after running. It is also used to effect page table updates when virtual memory is requested or released.

These eight cycles of work need to happen when the multitasking timer expires, but how does the CPU know what to do? In other words, what forcibly takes the CPU away from a running user program without creating havoc?

Not counting the preemption timer of Section 3.1.2, the circuit that captures the CPU from user programs only adds two NOR gates and two 1-bit D flip-flops, known as the *control decoder hijack counter*, to the architecture. But it quadruples the tiny amount of firmware present in the instruction decoder. What happens is this: the hijack counter is ordinarily zero, but when a `YIELD` instruction is executed (for cooperative multitasking) or the preemption timer expires (for preemptive multitasking), its flip-flops count out the next three instruction cycles, and then return to and remain zero.[21]

The instruction decoder RAMs produce the control signals each opcode needs during its four clock cycles. This isn't much memory: 4 cycles $\times$ 512 instructions possible = 2048 doublewords. But these RAMs contain four variants of each instruction's firmware, for a total of 8192 doublewords. These variants are keyed by the hijack counter. Variant 0 is the instruction as it ordinarily exists. Variants 1, 2, and 3 are recalled when a context switch from running a user program to running the kernel occurs.

When the hijack counter begins its sequence, variant 0 of an opcode's firmware won't be run. Instead, variant 1 is substituted. This might be any opcode a user program is allowed to execute; perhaps `S` (subtract), `JUMP`, or whatever. Variant 1 executes the instruction normally. Its firmware is identical to variant 0, except the control signal that allows the instruction pointer to increment is suppressed. The incrementer has a long path length, so it's too late to prevent I, the address of the instruction executing as variant 1, from becoming $I + 1$. But we need to prevent $I + 1$, which is the correct resume address for the next CPU timeslice, from becoming $I + 2$, which would skip an instruction.

The next instruction will be decoded as variant 2. It doesn't matter what the instruction is or what address it was fetched from because variant 2's firmware is identical for all 512 opcodes. Variant 2 does two things. First, $I + 1$ is pushed on the user program's return address stack. This is the location of the next instruction when the user resumes. Second, the CPU is switched to `PRIV` mode, thereby switching to the superuser's registers, stack, and page table.

The next instruction will be decoded as variant 3. Its firmware is also identical for all 512 opcodes, so it doesn't matter what the instruction is or what code address it's in. The firmware is bit-identical to `REVERT`, causing the instruction pointer and flags to be popped from the kernel's call stack. The next instruction fetch will be where the kernel takes over, decoded in normal fashion as variant 0.

Not counting latency to catch the preemption timer expiration and finish the user instruction immediately prior to the context switch, a complete transition from user program to kernel takes eight clock cycles.

The opposite transition from kernel to user program is simpler because the kernel is non-preemptable. Rather than respond to an expiring timer, the kernel directly

[21]Simplification. The actual count sequence is Gray code 0, 1, 3, 2 to conserve components. The sequence advances once for each instruction cycle, which is four clock cycles.

issues the needed instructions to effect the context switch. Only three instructions (12 clock cycles) are needed:

First, a `USER` instruction executes to direct which program will run next. This instruction can be skipped if the kernel elects to continue the program that was just interrupted. Because the preemption timer has a maximum setting of 65 535 instructions, which isn't very many, returning to the just-interrupted program may be the norm at times.

Second, an `NPCALL` (call as nonprivileged program) instruction calls a short pseudo-subroutine. It's immaterial that a subroutine is being called because the prime objective is to place the kernel's address to continue at on the stack. This instruction also switches the CPU to `NPRIV` mode, making the call stack, page table, and registers for the continuing user program available.

Third, a `REVERT` instruction executes, thereby popping the user program's saved instruction pointer and flags from its own stack. The next instruction fetch will continue the user program.

Although these three steps are the whole story of kernel-to-user context switches, complexity is hidden in its briefness. The target of `NPCALL` never returns. A casual reading indicates that it `REVERT`s, which is a special case of returning, but that isn't what happens either. The `REVERT`, which executes as user code, is not paired with the `NPCALL`, which executes as kernel code. The other condition to meet is that `REVERT` can't follow `NPCALL` as the next instruction because the `REVERT` must happen *before* the user program's timeslice, and the instruction that follows `NPCALL` continues the kernel *after* the user program's timeslice. The `REVERT` is written as if it's a separate subroutine.

3.9 Instruction Set Tables

Tables 7–17 provide a short index of non-privileged Dauug|36 instructions. Rather than describe semantics for each instruction, they state a brief overview of the instructions, how they are named, and where some appear in this paper. The privileged instructions follow in Tables 18–20.

Privileged instructions must not be executed by user programs, because they can escalate privilege as well as compromise separation between programs. The enforcement mechanism is that the OS kernel's program loader knows which instructions are privileged and refuses to copy them to code memory without authorization. To make this distinction easy, nonprivileged instructions are assigned opcodes 0–447, and privileged instructions are assigned opcodes 448–511. A program cannot circumvent the opcode restriction by branching to another program's code, because the program loader checks all branch instructions to ensure they do not cross text segments. The opcode restriction and branch restriction can be overcome by the `XANY` (execute any instruction) and `JANY` (jump anywhere) instructions respectively, but they are privileged.

Table 7: Additive instructions (sect. 3.6.1). Half are stratified by signage.

Mnemonic	# of	Description
`A`	8	add
`AC`	8	add with carry
`AW`	1	add with wrap
`AWC`	1	add with wrap and carry
`RS`	8	reverse subtract (sect. 3.7.1)
`RSC`	8	reverse subtract with carry
`RSW`	1	reverse subtract with wrap
`RSWC`	1	reverse subtract with wrap and carry
`S`	8	subtract
`SC`	8	subtract with carry
`SW`	1	subtract with wrap
`SWC`	1	subtract with wrap and carry

Table 8: Bitwise boolean instructions (sect. 3.5)

Mnemonic	Description
`AND`	AND
`IGF`	ignorant false
`IGT`	ignorant true
`LANR`	left AND NOT right
`LONR`	left OR NOT right
`NAND`	NAND
`NL`	NOT left
`NOR`	NOR
`NR`	NOT right
`OR`	OR
`RANL`	right AND NOT left
`RONL`	right OR NOT left
`XL`	exactly left
`XNOR`	exclusive NOR
`XOR`	exclusive OR
`XR`	exactly right

Table 9: Immediate instructions (sect. 3.3) move constants to the low or high half of registers.

Mnemonic	**Description**
`IMB`	immediate both
`IMH`	immediate high
`IMN`	immediate negative
`IMP`	immediate positive

Table 10: Shift and rotate instructions. Arithmetic shifts are stratified by signage.

Mnemonic	**# of**	**Description**
`ASL`	4	arithmetic shift left (sect. 3.6.1)
`ASR`	4	arithmetic shift right (sect. 3.6.1)
`LSL`	1	logical shift left
`LSR`	1	logical shift right
`ROL`	1	rotate left

Table 11: Multiplication instructions (sect. 3.6.2–3.6.3)

Mnemonic	**Description**
`DSL`	double shift left
`MH`	multiply high
`MHL`	multiply high and low
`MHL0`	multiply high and low, tribble 0
`MHL1`	multiply high and low, tribble 1
`MHL2`	multiply high and low, tribble 2
`MHL3`	multiply high and low, tribble 3
`MHL4`	multiply high and low, tribble 4
`MHL5`	multiply high and low, tribble 5
`MHNS`	multiply high no shift
`ML`	multiply low

Table 12: Bit-rearranging instructions

Mnemonic	Description
`PAIT`	permute across and inside tribbles (sect. 3.7.7)
`PAT`	permute across tribbles (sect. 3.7.7)
`PIT`	permute inside tribbles (sect. 3.7.7)
`SWIZ`	swizzle (sect. 4.3)
`TXOR`	transposing XOR (sect. 3.7.6)

Table 13: Substitution-permutation network instructions (sect. 3.7.8)

Mnemonic	Description
`MIX`	mix
`XIM`	unmix

Table 14: Miscellaneous arithmetic instructions. Most are stratified by signage.

Mnemonic	# of	Description
`CMP`	4	compare
`CRF`	1	clear R(ange) flag
`MAX`	8	maximum (sect. 3.7.2)
`MIN`	8	minimum (sect. 3.7.2)
`NUDGE`	1	nudge (sect. 3.7.3)

Table 15: Mixed arithmetic-with-logic instructions

Mnemonic	**Description**
`HAM2`	Hamming weight part 2 (sect. 3.7.4)
`NOLIST`	`NOP`, but not shown in listings
`NOP`	no operation
`STUN`	stacked unary (sect. 3.7.9)
`UN.A`	single-RAM unary operations, alpha layer
`UN.B`	single-RAM unary operations, beta layer
`UN.G`	single-RAM unary operations, gamma layer

Table 16: Branch instructions. `JUMP` comes with flag-conditioned variants.

Mnemonic	**# of**	**Description**
`CALL`	1	call
`JUMP`	11	jump
`RETURN`	1	return
`REVERT`	1	return with original CPU flags
`YIELD`	1	relinquish CPU (sect. 3.8)

Table 17: Memory instructions. Data memory addresses are virtual.

Mnemonic	**Description**
`ADDLD`	add then load (sect. 3.7.10)
`LD`	load
`LDSTO`	load and store (sect. 3.7.10)
`STO`	store
`STO2`	store twice (sect. 3.7.10)

Table 18: Identity-modifying instructions (privileged)

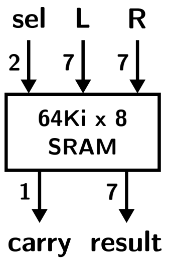
Mnemonic	Description
NPCALL	call subroutine and become nonprivileged user
NPRIV	become nonprivileged user
PCALL	call subroutine and become superuser
PEEK	copy user register to superuser
POKE	copy superuser register to user
PRIV	become superuser
SETUP	access user's stack and user's page table as superuser
USER	specify current nonprivileged user

Table 19: Program initialization instructions (privileged)

Mnemonic	Description
CALI	call stack initialize
JANY	jump to any code memory address
TIMER	set preemptive multitasking timer interval
XANY	fetch and execute instruction from register

Table 20: Unrestricted memory instructions (privileged). Data memory addresses are physical.

Mnemonic	Description
ADDRDM	add then read data memory (sect. 3.7.10)
RCM1	read code memory part 1 (sect. 3.1.2)
RCM2	read code memory part 2 (sect. 3.1.2)
RDM	read data memory
RPT	read page table
RWDM	read and write data memory (sect. 3.7.10)
WCM	write code memory
WDM	write data memory
WDM2	write data memory twice (sect. 3.7.10)
WPT	write page table



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

Figure 7: A small RAM used as a simple lookup element. Arrow labels show number of wires.

4 Arithmetic Logic Unit Theory of Operation

Chapters 4–7 of [2] present a theory of SRAM arithmetic logic units that can be tailored to a range of CPU objectives. Dauug|36 tries to minimize the number of clock cycles required for a particular task, while offering a reasonably-large word size of 36 bits. Applications that can tolerate smaller word sizes and/or more clock cycles to accomplish a task can use a considerably different—and proportionately smaller—ALU based on techniques from these chapters that Dauug|36 does not use or need. This section that you are now reading focuses on Dauug|36's ALU; for other use cases, [2] may be a more complete reference.

4.1 Simple Lookup Elements

Simple lookup elements are the fundamental building block of all SRAM logic. Note that the use of RAM has nothing to do with its mutability: we actually want ROM, but RAM is much faster and does not require custom fabrication. 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 typically one or two subwords and sometimes a carry bit.

Figure 7 shows a whimsical lookup element that offers two logical and two arithmetic functions. The element accepts two 7-bit inputs, here named `L`(eft) and `R`(ight), and a 2-bit input `sel` that specifies the operation to perform. The 8 output bits are partitioned into a 7-bit `result` for the four operations, and a 1-bit `carry` that outputs a carry bit for addition, borrow bit for subtraction, or 0 for NAND and XOR.

4.2 Three-Layer Carry-Skip Adders

Although Figure 7 can add small words directly, it can't naively scale to add larger words. A one-RAM adder for even 16-bit words requires $(16 + 1) \times 2^{16+16}$ bits of storage—more than 240 times that of today's largest SRAM ICs. Nor can addition be trivially parallelized by bit slicing. Although we can break two 16-bit words into 4-bit subwords and use 4 RAMs to add the subwords pairwise, these bit slices are not independent of one another due to carries that sometimes occur from one bit slice to the next. So the four bit slices will produce 16 result bits and 4 carry bits, but the output we need is 16 result bits and 1 carry bit.

If each RAM is given its own carry input such that carries can chain from right to left (least- to most-significant), four small RAMs can easily compute a 16-bit sum, but there is a time penalty proportional to the number of bit slices, which thus far is equal to the number of RAMs. The rightmost slice, slice 0, must finish its work before slice 1 can start, and so on. This is called *ripple carry*, and it doesn't scale to add large words quickly. An adder where all RAMs that do similar work can operate simultaneously would add faster.

Figure 8 is a four-slice, 16-bit adder with no ripple. It's called a *carry-skip adder*, and it uses nine RAMs. Without loss of generality, these RAMs are assumed to be synchronous (have clocked inputs), so their sequence can be described in terms of clock cycles, sometimes regarded as *layers*. The dependencies are such that three clock cycles are required to add two numbers. Dauug|36's carry-skip adder is much like this figure and also has three layers, except it has two more slices and two more bits per slice (a total of thirteen RAMs) to accommodate 36-bit words.

In a three-layer carry-skip adder, the left operand, right operand, and output are the same width within a given slice.[22] The first layer of each slice simultaneously adds subwords locally, producing a local sum `s` that may wrap around, a local carry output `c`, and a local *propagate* output `p` that indicates when the local sum `s` is all ones, at which time the local carry output `c` will be zero. The third layer simultaneously but selectively increments the local sum of each slice with wrapping, depending on whether the slice's *carry decision* indicates a carry into the slice.

A slice's carry decision is true if and only if either of the following is true:

- The slice's local carry output is true.
- The slice's local propagate output is true, a neighboring, less-significant slice exists,[23] and said neighboring slice's carry decision is true.

The slice carry decisions are therefore recursively defined. In non-SRAM carry-skip adders, this recursion inserts one AND gate's delay between each slice, resulting in a right-to-left ripple that is tiny compared to that of a ripple-carry adder. In contrast, SRAM carry-skip adders dispense with *all* ripple,[24] because the second layer's single RAM looks up all carry decisions simultaneously based on propagate and carry signals that arrive all at once from the first layer.

Carry-skip adders can be built with two layers instead of three, subject to drawbacks [2].

[22]In SRAM ALUs all bit slices are typically the same width, although this is not so in fast SRAM multipliers.

[23]When chaining additions to support multiple-word operands, this slice may be in a neighboring word.

[24]The internal operation of the second layer's SRAM is not considered here.

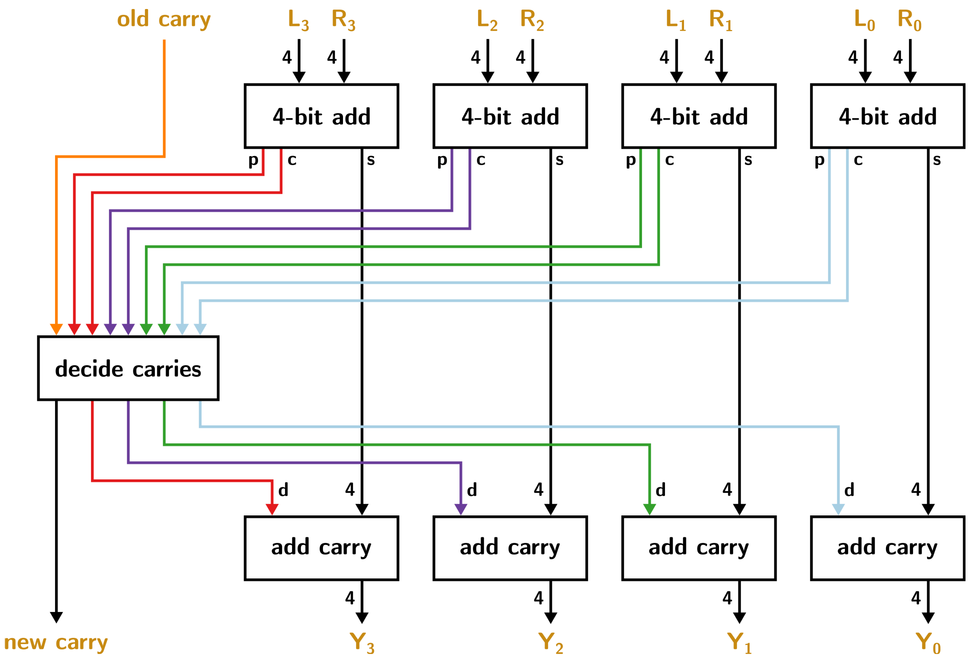


Figure 8: A four-slice, 16-bit carry-skip adder. The nine rectangles depict SRAMs. Paths labeled 4 convey four bits; other paths convey one bit. `L`, `R`, and `Y` indicate left operand, right operand, and result subwords. `p`, `c`, and `s` indicate each subword's propagate bit, carry bit, and partial sum intermediate. `d` indicates each subword's decision whether to add 1 due to a carry. The `old carry` and `new carry` wires facilitate iteration for multiple-word operands.

4.3 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 outputs one data bit to each subword. Figure 9 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 a subword-local place value of 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.

Like all SRAMs in Section 4, a swizzler's RAMs contain read-only lookup tables. For this reason, the firmware designer must determine in advance what functions a swizzler will implement. Dauug|36's `SWIZ` instruction, which operates on six slices of six bits, currently supports copying each of the six subwords to all of the others,

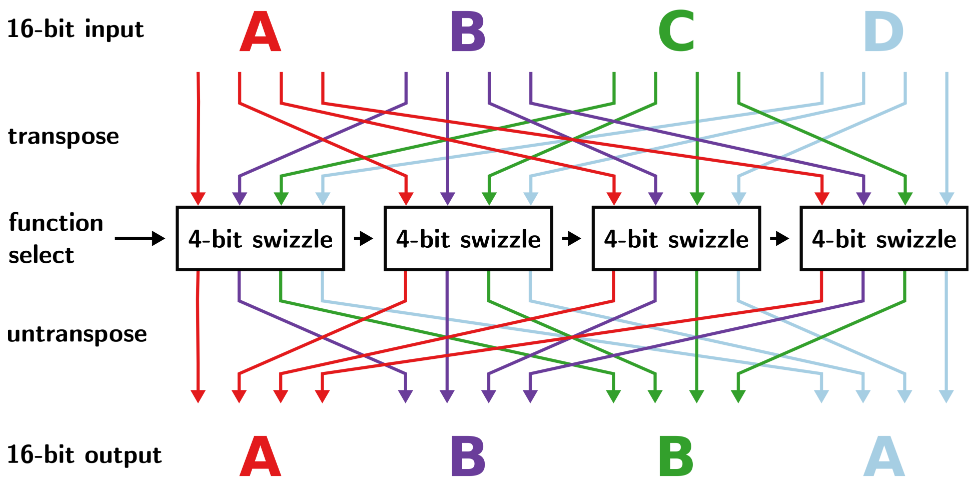


Figure 9: A four-slice, 16-bit swizzler acts like a multi-way switch. The four rectangles depict SRAMs. Identical, self-inverse wire transpositions appear at the input and output, where the i^{th} bit of the j^{th} subword is moved to the j^{th} bit of the i^{th} subword. By simultaneously executing the same operation, the four SRAMs can direct any subword of the input to any subword of the output.

leaving space for another 58 swizzle operations that have not yet been chosen.[25]

4.4 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 RAMs that can operate *within* subwords rather than *across* them. Figure 10 shows this combination, which permits rotation of any number of bits in a single pass. To shift instead of rotate, appropriate masking is easily added to the RAM contents of either layer.

Two important requirements pertain to logarithmic shifters. First, the RAMs within a layer need to all process the same number of bits. Second, the bits leaving the RAMs of layer one must be evenly distributed to the RAMs of layer two. Thus

[25]There are $6^6 = 46\,656$ "strict" swizzles, where each output bit is a copy of one of six input bits, that these 58 can be chosen from. But these RAMs could in principle implement any of the 64^{64} functions that map six bits onto six bits.

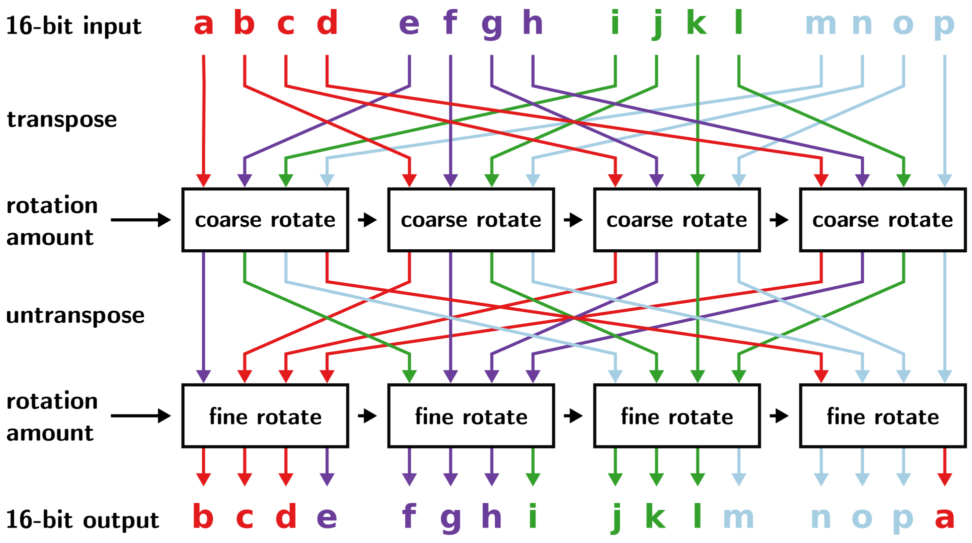


Figure 10: A four-slice, 16-bit logarithmic shifter is assembled from a swizzler followed by a layer of simple lookup elements. The eight rectangles depict SRAMs. The transposition wiring is identical to Figure 9, although color is used here to indicate a one-bit left rotation.

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.

Logarithmic shifters can be built with half the number of layers and transpositions (one and one, respectively), but each operation would require two passes to complete [2].

4.5 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. Table 21 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. Figure 11 shows a substitution-permutation network that uses the S-box

Table 21: 4-bit S-box

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

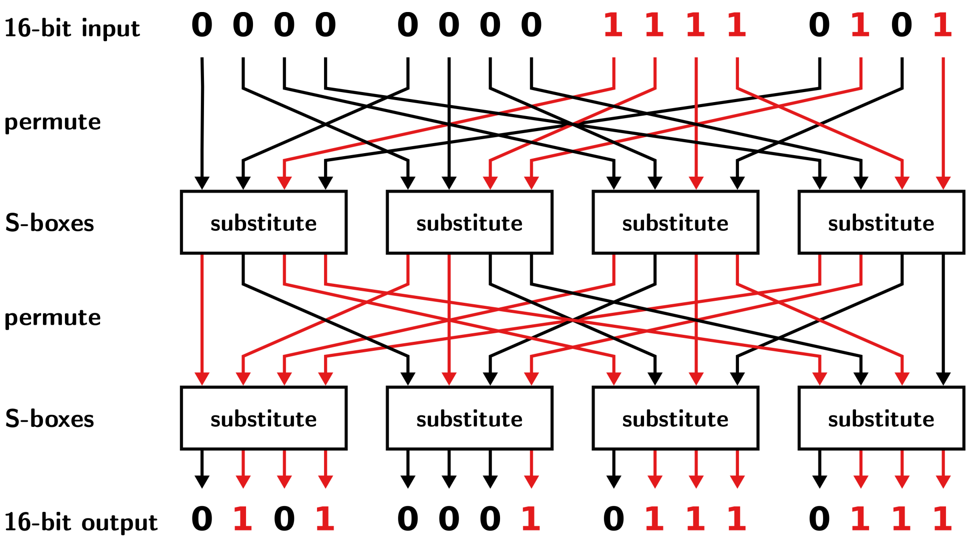


Figure 11: A four-slice, 16-bit substitution-permutation network. Except for the firmware in its eight SRAMs, this is the same circuit as the logarithmic shifter of Figure 10.

of Table 21 in every RAM for ease of understanding, but in practice, S-boxes may vary between RAMs and/or be key-dependent.

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 pseudorandom number generators (PRNGs). Suitability for cryptography is considered in Section 3.7.8.

4.6 Three-Layer Arithmetic Logic Units

The SRAM logic blocks of Sections 4.2–4.5 and Figures 8–11 complement each other like pieces in a puzzle, as if meant to assemble into still-more-capable circuits. Most strikingly, the three-layer carry-skip adder of Figure 8 shows a void in its second layer while and where carry decisions are made. There is space to insert four more

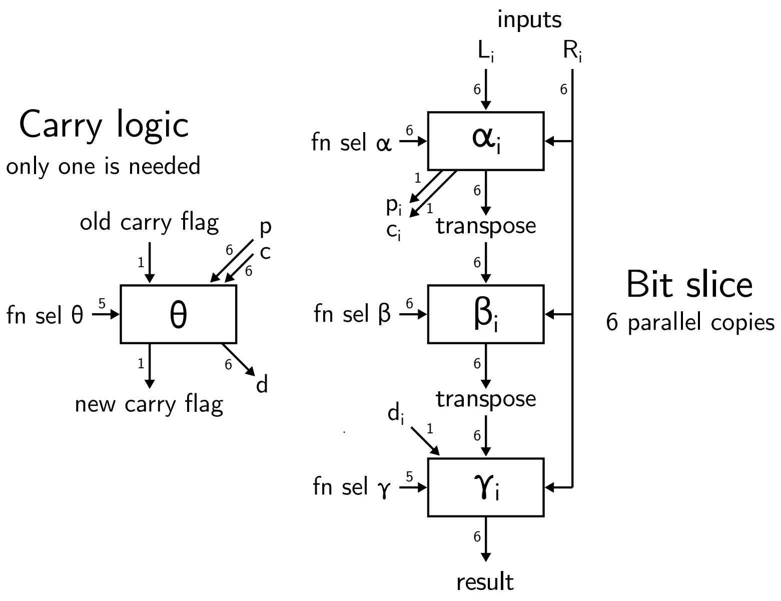


Figure 12: A six-slice, 36-bit arithmetic logic unit. Digit labels indicate number of wires. Subscript i designates bit slices numbered from 0 through 5. The three layers are named α, β, and γ solely for their order of occurrence. Carry decision RAM θ is so-named because it's "off at an angle" relative to the six β RAMs it's parallel to.

RAMs within the sum datapaths, with each processing four bits. The swizzler of Figure 9 matches this description exactly. The resulting circuit would be a mess to draw on a plane without using numerous overlays, but general idea is that Figure 9, including both wire transpositions, would be stuffed into the center of Figure 8.

Figure 12 extends the concept to 36 bits and shows the general flow of the arithmetic logic unit used by Dauug|36. To keep the central ideas prominent, certain flag-handling details are glossed over.[26] Only one of the six bit slices is written out, and the α–β and β–γ transpositions across slices are simply marked "transpose" instead of drawn with 72 wires in six colors. A square-matrix explanation of these transpositions appears in Figure 13.

The ALU of Figure 12 is a quadruple superposition of the circuits earlier considered. Layers α and γ, with assistance from RAM θ, form a carry-skip adder. Layer β, the bits of which are transposed relative to the others, form a swizzler. Layers β and γ, with transpositions, form a logarithmic shifter. And although layers β and γ form a substitution-permutation network, it turns out that layers α, β, and γ form

[26]One of several details not discussed is that RAM α_5 must tell ζ whether a sum or difference will fit in its destination register, contingent on whether the register is unsigned or signed, and whether or not θ detects a carry into tribble 5.

$$\begin{array}{cccccc} \multicolumn{6}{c}{\mathtt{w}} \\ \mathtt{z} & \mathtt{y} & \mathtt{x} & \mathtt{w} & \mathtt{v} & \mathtt{u} \\ \mathtt{t} & \mathtt{s} & \mathtt{r} & \mathtt{q} & \mathtt{p} & \mathtt{o} \\ \mathtt{n} & \mathtt{m} & \mathtt{l} & \mathtt{k} & \mathtt{j} & \mathtt{i} \\ \mathtt{h} & \mathtt{g} & \mathtt{f} & \mathtt{e} & \mathtt{d} & \mathtt{c} \\ \mathtt{b} & \mathtt{a} & \mathtt{9} & \mathtt{8} & \mathtt{7} & \mathtt{6} \\ \mathtt{5} & \mathtt{4} & \mathtt{3} & \mathtt{2} & \mathtt{1} & \mathtt{0} \end{array} \qquad \begin{array}{cccccc} \multicolumn{6}{c}{\mathtt{w}^\top} \\ \mathtt{z} & \mathtt{t} & \mathtt{n} & \mathtt{h} & \mathtt{b} & \mathtt{5} \\ \mathtt{y} & \mathtt{s} & \mathtt{m} & \mathtt{g} & \mathtt{a} & \mathtt{4} \\ \mathtt{x} & \mathtt{r} & \mathtt{l} & \mathtt{f} & \mathtt{9} & \mathtt{3} \\ \mathtt{w} & \mathtt{q} & \mathtt{k} & \mathtt{e} & \mathtt{8} & \mathtt{2} \\ \mathtt{v} & \mathtt{p} & \mathtt{j} & \mathtt{d} & \mathtt{7} & \mathtt{1} \\ \mathtt{u} & \mathtt{o} & \mathtt{i} & \mathtt{c} & \mathtt{6} & \mathtt{0} \end{array}$$

Figure 13: Bit transposition as a square matrix reflection. The bit positions of a 36-bit word `w` can be written as a 6×6 square matrix using base 36. The α–β and β–γ wire transpositions in the Dauug|36 ALU are simply reflections through the main diagonal, and are therefore self-inverse. The ALU's α and γ layers operate on the rows of `w`, while the ALU's β layer operates on the rows of $\mathtt{w}^\top$, which are the *columns* of `w`.

a bigger one, so the architecture uses that.

The ALU's RAMs are soldered in place, so physical reconfiguration is not possible when switching from addition to rotation, or swizzling to encryption. But there is no operational conflict so long as the RAMs are large enough to contain firmware for all of these operations. The α, β, and γ RAMs divide their 18 inputs equally between a six-bit left operand, six-bit right operand, and six-bit "function select" that says what needs to happen. For the γ RAMs only, one of the function select bits is taken from θ's carry decision pronouncement. The θ RAM must also support many functions, as must ζ, an unpictured RAM that manages transitions for the `N`(egative), `R`(ange), `T`(emporal overrange), and `Z`(ero) flags. θ and ζ have five function select bits each. The total firmware is rather complex: among the α, β, γ, θ, and ζ RAMs, 113 operations are now defined for the function select inputs, and some operations for α, β, or γ require their six RAMs to use different tables. Three of these 113 operations are the "identity" operations that α, β, and γ use to pass their left operand unchanged when the layer in question does not contribute to a calculation.

The arithmetic logic unit does not treat its left and right operands equivalently. The left operand is replaced by each layer with a revised operand, but the original right operand is supplied unchanged to every RAM. There is also no transposition of the right operand coming into the β layer, so its six RAMs compute on transposed left subwords with non-transposed right subwords.

A much-expanded description of the Dauug|36 ALU and firmware is in chapters 5 and 7 of [2]. Post-2022 changes can be found in the implementation source code. Chapter 5 also contrasts 36-bit words with other plausible architecture widths. Chapter 6 shows how to build a half-decent SRAM ALU that uses two layers instead of three, although it will have drawbacks.

5 Implementation

5.1 Portions That Are Substantially Complete

The logical design and dataflow for the central processing unit and memory subsystem have been stable and testable since April 2023. The remainder of this section describes several derivative products that are to a large degree operational.

5.1.1 Netlist Generation

Although open-source tools for designing and simulating CMOS ICs exist, I did not find an open-source tool that is suitable for designing and simulating *circuit boards.* Efforts to use a leading open-source electronic design automation suite were catastrophically unable to cope with the 244 parts and 6948 pins that comprise Dauug|36 as of early 2024. Failing to identify an existing tool, I designed a macro language for describing components and their interconnections, another language for assigning components to 21.1 × 27.1 mm "tiles" and placing the tiles on a grid, and a Python script to transform the two languages into a machine-readable netlist. The same script estimates pin-to-pin delays based on estimated track lengths and capacitances extracted from component positions. It also checks for obvious mistakes such as shorting an output to ground.

5.1.2 Electrical Simulation

I wrote an electrical simulation of every type of component used in the architecture, including timing and capacitance information from datasheets, in the C language, along with a discrete-event simulator. This simulator models the CPU and memory subsystem exactly as the netlist describes them. Its purpose is not only to seek out wiring and logic errors, but also to detect timing conflicts and overcurrent conditions. The model includes the most-recent Dauug|36 firmware, so the model can run various regression tests and assembly-language programs. In one, the electrical simulation causes the modeled CPU to run a short bootloader. The bootloader, in turn, then loads and starts Osmin, the first operating system for Dauug|36, and then Osmin loads several user programs and executes them simultaneously. These tests support my opinion that Dauug|36's preemptive multitasking and memory protection work correctly.

One of the electrical simulator's strengths is also its principal limitation, specifically, its modeling of signals between 6948 IC pins with picosecond granularity. If an SRAM has an 18-bit address input, the 18 bits arrive at separate times. In fact, they arrive more than once each because many of their connected output pins will transition to an uncertain state prior to reaching a settled state. For each arriving transition, the simulation must check for conflicts from other components that may have recently driven the same input pin. My workstation takes 23 hours to electrically simulate one second of the early-2024 Dauug|36 netlist, and the simulation does not parallelize.[27]

[27]For illustration only, the workstation CPU is an Intel Core i5-6200U running at 2.30 GHz.

Dauug|36's targeted speed is 20 MIPS (million instructions per second), based on an 80 MHz clock and a surface-mount board that can be hand-soldered. This speed has not been reached in simulation as yet, but 16 MIPS from a 64 MHz clock runs cleanly. The emulated architecture fits on a 20 × 20 cm board. Separation between components is at least 4.1 mm in all cases, which is probably sufficient in light of the much-tighter 0.5 mm pitch for pins on certain components.

5.1.3 Virtual Machine

While developing the arithmetic logic unit, I wrote a virtual machine that could quickly emulate the ALU without considering the electrical implementation. Soon, the `JUMP` variants and a few other instructions were emulated also. The VM's ability to run hundreds of thousands of Dauug|36 instructions per second is ideal for ALU firmware regression tests, including long multiplication verification and many needed `T`(emporal) and `R`(ange) flag checks.

The virtual machine's "wiring" of the ALU is implemented in the C language, not in the macro language that defines the circuit board. Although having a VM that can quickly emulate the remainder of the architecture would greatly enhance regression tests for operating system correctness, a significant risk is that through an oversight, the VM's logical model may not stay synchronized with the electrical model. The best solution would be to rewrite the virtual machine to accept a netlist as input, and to redesign the netlist macro language and software so that a common specification produces the circuit board, electrical simulation netlist, and virtual machine model. This would also be a good point to adopt a subset of an already-standardized language such as VHDL or Verilog for specification. But as of early 2024, no fast emulation of the full instruction set exists.

5.1.4 Assembly Language

Dauug|36's assembly language seeks a balance between ease of use and ease of implementing the assembler. As a language illustration, Figure 14 presents a two-page working solution of the familiar Tower of Hanoi puzzle, where a "tower" of progressively-smaller disks must be moved between two spindles one disk at a time, using a third spindle as a spare to rest disks. The invariant that no disk may be placed atop a smaller disk must be maintained.

This sample program encodes the three spindles as the integers [0, 1, 2], which are stored in registers `from` and `to` when planning a disk motion. No register indicates the `spare` spindle because it is computable as `spare` = 3 − `from` − `to`. Because the architecture does not support recursive function calls (Section 3.1.1, "What the stack is not"), a stack grows upward from virtual address 0 for iterative subproblem queuing. Each word on the stack encodes one subproblem, using 2 bits each for the "from" and "to" spindles and an overkill 32 bits for the number of disks. Because zero is not a valid subproblem encoding, it is used to indicate the bottom of the stack. Program output is a series of lines in the format `2 -> 0` (move a disk from spindle 2 to spindle 0).

The sample program contains four defects. First, the stack data should use 12 bits for each field, and rotations instead of shifts, to reduce the number of integer constant registers. Second, the stack will overflow if too many disks are requested. Third, this example uses the privileged `WPT` (write page table) instruction to seize page 0 of physical data memory. It should instead obtain memory through an operating system request. Fourth, the program should terminate via a `RETURN` to the operating system, because although the simulator has a `HALT` instruction, real machines won't.

Most language features of Figure 14 can be understood with little or no coaching. The operators `+`, `-`, `+=`, `-=`, `&`, and `|` are borrowed from C and have their familiar meanings. From any semicolon rightward is a comment, as is everything in parentheses. Although comments don't nest, they can be subordinated by using a different number of parentheses, e.g.:

```
(((This comment takes three (3), not four)))), parentheses to close.)))
```

Registers are not numbered, but declared using `signed` and `unsigned` keywords, with shortcuts `s.` and `u.` offered. Register names are local within callable *scopes*, which are introduced with double colons. Jump labels, introduced with single colons, are also local within scopes. A program begins at the first scope in the file, which is automatically named `main::`. Because most instructions have no room for immediate operands, the assembler generates and calls a subroutine that pre-loads all necessary constants into registers.

Registers are dead when a scope is not active, allowing future assemblers to coalesce registers using liveness analysis. The exception is when a register is "kept" using the `keep` keyword, in which case the register is always live and can be accessed from any scope via a double colon. A scope need not contain instructions, but may simply introduce a register namespace, as in the example's `g::` scope.

Numeric suffixes `‘u`, `‘b`, `‘o`, `‘d`, `‘h`, and `‘t` indicate radices 1, 2, 8, 10, 16, and 64 respectively, and a decimal suffix ≤ 64 indicates its radix, such as `‘21` for radix 21. Periods and apostrophes (`’`, not `‘`) in names behave as if they are letters.

Because the I/O subsystem for Dauug|36 is not yet implemented, the electrical simulation emulates a simple I/O subsystem containing a real-time clock and limited filesystem. It is accessed via a `PVIO` (paravirtualized I/O) instruction and has its share of quirks.

The assembly language does not yet support record structures or initialized data.

5.1.5 Cross Assembler

Multi-million line compilers such as GCC, Clang, and LLVM, as well as their associated tools, are impractical to assess for exploitable defects or protect against tampering. It was shown 40 years ago that backdoors can survive toolchain self-compilations, even without malicious content in the source code [39]. Perhaps worse, a proof that static code analysis cannot ever detect all malware, even if artificial intelligence is used, has been known for 70 years [31]. But both problems can be escaped for a specific toolchain if it is small enough for a human to audit its source

```
; ==================================================
; Tower of Hanoi puzzle (invented by E. Lucas, 1883)
; ==================================================
    wpt (write page table) 0 = 0    ; map virtual page 0 onto phys. page 0
    g::tos = 0                      ; stack grows upward from virt. addr 0
    sto (store) g::tos = 0          ; place 0 as bottom-of-stack guard

    g::from = 0                     ; tower originally on spindle #0
    g::to = 2                       ; tower finishes on spindle #2
    g::n.disks = 8                  ; 8 disks will result in 255 motions
    call push.work                  ; to stack: move 8 disks from #0 to #2

    u. f(ilename) ig(nored)         ; declare two unsigned registers
    f = pts`t(etrasexagesimal)      ; base-64 filename is "000pts"
    ig = f pvio out8`t              ; 000pts points to /dev/pts/0 (stdout)

loop:                               ; top of main problem-solving loop
    call pop.work                   ; pop from, to, n.disks from stack
    jump == solved                  ; Z(ero) flag means problem is solved

    cmp g::n.disks - 0              ; test for n.disks == 0, which should
    jump == loop                    ; never occur, but would be a no-op

    cmp g::n.disks - 1              ; simple case: just one disk to move
    jump != multi.disk
    call print.move                 ; just print out the move
    jump loop

(   The multi.disk case will move more than one disk. We must move:
        n-1 disks:  'from'  to 'spare'
        1 disks:    'from'  to 'to'
        n-1 disks:  'spare' to 'to'
    Because these steps go on a stack, we push them in reverse order.   )

multi.disk:                         ; we get here to move more than one disk
    u. save                         ; declare unsigned register
    save = g::n.disks               ; will need prev. value of n.disks

    g::from = 3 - g::from           ; THIRD step that will happen:
    g::from -= g::to                ; use current 'spare' spindle as 'from',
    g::n.disks = save - 1           ; move n-1 disks
    call push.work
```

Figure 14: Dauug|36 assembly language demonstration: Tower of Hanoi puzzle solution (part 1 of 3).

```
    g::from = 3 - g::from          ; SECOND step that will happen:
    g::from -= g::to               ; use current 'from' spindle as 'from',
    g::n.disks = 1                 ; move 1 disk
    call push.work

    g::to = 3 - g::to              ; FIRST step that will happen:
    g::to -= g::from               ; use current 'spare' spindle as 'to',
    g::n.disks = save - 1          ; move n-1 disks
    call push.work
    jump loop                      ; multi.disk case is now pushed on stack

solved:                            ; we get here when solution is complete
    ig = 10 pvio write`t           ; write final newline
    ig = f pvio close`t            ; close the terminal output
    halt                           ; return from simulation

; ---------------------------------------------------------------------------
push.work::                        ; subroutine: push sub-problem on stack
    u. work                        ; bit fields: from, to, n.disks
    work = g::from
    work = work asl 020202020202`o ; left shift 2 bit positions; `o = octal
    work = work | g::to
    work = work asl 404040404040`o ; left shift 32 bit positions
    work = work | g::n.disks
    g::tos += 1
    sto (store) g::tos = work
    return

; ---------------------------------------------------------------------------
pop.work::                         ; subroutine: pop sub-problem from stack
    u. work                        ; bit fields: from, to, n.disks
    work = ld (load) g::tos
    jump == done                   ; if loaded 0, then stack is empty
    g::tos -= 1
    g::n.disks = work & 037777777777`o  ; bits 31--0 is 'n.disks'
    work = work asr 040404040404`o ; right shift 32 bit positions
    g::to = work & 3`o             ; bits 33--32 is 'to' spindle
    work = work asr 424242424242`o ; right shift 2 bit positions
    g::from = work & 3`o           ; bits 35--34 is 'from' spindle
    work = 1                       ; ensure Z(ero) flag is not set
done:
    return
```

Figure 14: Dauug|36 assembly language demonstration: Tower of Hanoi puzzle solution (part 2 of 3).

```
; ----------------------------------------------------------------------------
print.move::                        ; subroutine: called when n.disks == 1
    u. ig(nored)                    ; declare unsigned register
    ig = g::from pvio dec‘t         ; print g::from in decimal
    ig = 32 pvio write‘t            ; print " -> "
    ig = 45 pvio write‘t
    ig = 62 pvio write‘t
    ig = 32 pvio write‘t
    ig = g::to pvio dec‘t           ; print g::to in decimal
    ig = 10 pvio write‘t            ; print newline
    return

; ----------------------------------------------------------------------------
g(lobal register namespace)::       ; invariant: from + to + spare == 3
    u. from                         ; spindle to move from (0, 1, or 2)
    u. to                           ; spindle to move to (0, 1, or 2)
    u. n.disks                      ; number of disks to move
    u. tos (top of stack)           ; stack grows up, zero guard at bottom
    keep from to n.disks tos        ; allow global access to these registers
```

Figure 14: Dauug|36 assembly language demonstration: Tower of Hanoi puzzle solution (part 3 of 3).

code *and executable code.* The toolchain's "root of trust" would be a *self-hosted assembler*, a short program that, when given its own source code as input, produces its own executable as output. But as Ken Thompson demonstrated, this is not sufficient. The "trust" part of a self-hosted assembler becomes final when a human audits every word of the executable code and attests to its perfect transcription from the source code.

A self-hosted assembler for Dauug|36 is not implemented as of early 2024, but something very close is. The Dauug|36 *cross assembler*, which runs on a wide range of conventional CPUs, is written not in a familiar high-level language, but in a terse, human-written bytecode which a small interpreter executes. Notwithstanding its alien specification, this is the assembler that builds all present Dauug|36 software, including the Figure 14 example and the Osmin kernel. To become a self-hosted assembler, only the bytecode *interpreter* needs rewritten in assembly language—the bytecode itself will port unmodified to any host architecture. Needed work to finish a self-hosted assembler is minimal because the bytecode interpreter is well under 2500 lines of C and is already tested.

5.1.6 Firmware

The distinctive characteristic of Dauug|36 is that of being *solder-defined* (Section 2), giving the end user decisive control and final say over all aspects of its electrical and firmware design. The firmware is a not a "program" in the customary sense that it "runs." Rather, the firmware is a collection of random-access tables, in which each table determines the output of a single SRAM for a single clock cycle based on that SRAM's input. The firmware might be said to be written in C, but this is not true. The firmware is "written in" a list of integers that are computed by a program that is written in C. In other words, a C compiler does not compile the firmware, but compiles a program that outputs the firmware.[28]

The present firmware is by far the most stable part of the implementation. In-use computers are too busy, too inaccessible, too numerous, and/or too support-constrained to rationally or ethically require firmware updates. Moreover, a temptation to introduce breaking changes to firmware may be too strong to resist. This is why so much attention has been given to the architecture's instruction set, semantics, CPU flags, extraordinary situations, and regression tests. No firmware release for a real-world minicomputer should ever contain an exploitable defect.

Notwithstanding the present firmware's comparative maturity, version 1 still feels faraway. Flag handling wasn't considered when the `IMx` (immediate both/high/negative/positive) opcodes were conceived. The stacked unary opcodes and their right operands are not yet stable. There are no opcodes to accelerate division, nor opcodes to aid signed multiplication. A planned, firmware-affected assembler macro scheme is not ready. Floating-point arithmetic isn't available even in single precision, nor have any I/O instructions been determined.

5.1.7 Documentation

By far the largest task implementing the Dauug|36 architecture has been its high documentation workload. An architecture no one knows how to use or maintain may as well not exist. Resources [2] and [3] alone exceed 200 000 words, and several further writeups exist.

5.1.8 Operating System

It is crucial that the first release of the hardware and firmware include not only a toolchain for programming, but also a preliminary operating system. Supporting adoption is not the only reason. A functioning OS is indispensable for developing and testing memory management, multitasking, the I/O subsystem, and device drivers. Writing an OS also helps identify capabilities that should be added to the firmware before its first release. For example, an ability to reverse a word's bit order in one instruction can be useful in CPU schedulers. Another reason to release an operating system is knowledge transfer, because the hardware is treacherously subtle. The operating system and its documentation will offer future architects and maintainers an executable specification as to how the architecture is intended to work.

In the near term, the scope of a Dauug|36 operating system is very minimal. Conformity with external specifications such as POSIX is not in scope because of

[28]To shield the firmware from corruption by a buggy or compromised C toolchain, the firmware generator may someday be rewritten in Dauug|36 assembly language. This project is not on my radar yet.

their breadth and unmet preconditions, such as existence of a C compiler. Instead, the OS's purpose is to enable programs to share the CPU and memory without interfering with one another, and that's pretty much it. Any other service can in principle be left for user programs to implement themselves, but a higher power—a *kernel*—must apportion memory and CPU time among multiple programs.

This kernel is already written and is named Osmin, after the comic villain of Mozart. Even with its discardable bootloader and single-use initialization routines that run at startup, Osmin weighs in at a scant 1339 instructions.[29] Implemented functionality as of early 2024 includes:

- Fetch the kernel from external storage, check for several possible errors, and start it.
- Quantify the host system's installed code, data, and page table memory.
- Wipe all primary storage at system startup.
- Wipe all primary storage when a shutdown is requested.
- Allocate, deallocate, and wipe pages of physical memory.
- Manage reference counts for physical memory pages.
- Obtain physically-backed virtual memory for the kernel.
- Create, manage, and retire hashed process ids.
- Load programs from external storage.
- Segment programs to run in fragmented code memory with negligible overhead.
- Update branch instructions to follow code relocation.
- Maintain reference counts for programs that may run multiple copies.
- Recover code memory from programs no longer running.
- Leave marked-to-remain programs in code memory even if not running.
- Reject nonprivileged programs that contain privileged instructions.
- Reject nonprivileged requests to manage privileged programs.
- Reject programs that can branch outside of their address space.
- Reject malformed executable files.
- Issue meaningful error codes for declined operations and kernel panics.

[29]For length comparisons with kernels for other architectures, each Dauug|36 instruction is $4\frac{1}{2}$ bytes.

- Start and terminate programs present in code memory.
- Terminate programs that underflow their call stack.
- Manage the multitasking preemption timer.
- Use round-robin scheduling to switch programs at timeslice boundaries.
- Manage internal kernel data structures and collections.
- Respond to API requests.
- Prevent nonprivileged access to unauthorized pages of data memory.
- Prevent writes to read-only data memory.
- Annotate and print any of several kernel data structures.

All the above already work and fit in a total of 1339 instructions. Osmin truly is OS MIN. The specificity, brevity, and simplicity of assembly language, combined with Dauug|36's expressive instruction set, make kernel programming straightforward and enjoyable. Much harder is writing the documentation needed to explain Osmin to a human.

Osmin suffers from some missing features, principally because I froze its development to catch up on documentation. Although the OS has an API, only two functions are implemented: terminate the calling program, and shut down the operating system. The latter offers no permission checking and was only put in as a stub to demonstrate that more than one API function is working. The next API calls to implement will request and relinquish virtual memory, which amount to minimal effort because all the physical memory management is already complete. Next after those are calls for interprocess memory sharing; the largest task here will be identifying how discovery and authorization semantics should work.

Although today's kernel already identifies certain operations as requiring privileges, I need to think through how permissions should work. For many embedded uses, an immutable table that lists what programs to start and what calls each may make would be optimal because of its simplicity and low susceptibility to mischief. But this mechanism can't handle a diverse set of programs that begin and end at various times. Similar choices lie ahead in determining if and how resource limits will be applied. Generally these limits will be maximums, but some programs will also need guaranteed resource *minimums* such as CPU availability.

Osmin is intended to be a real-time operating system (RTOS) that can meet hard scheduling deadlines. This isn't the case yet, because certain length-dependent kernel operations are done without interruption. These operations involve validating programs and moving them into code memory, allocating multiple pages of data memory, wiping memory that has been deallocated, etc. They become a problem, for example, when program X misses a deadline while the kernel is serving a large malloc to program Y. The solution will be to move such tasks out of the scheduler loop to a queue that is serviced by an interruptible kernel subprogram. After this is done, Osmin will reliably meet programs' hard deadlines irrespective of other users' system calls. Until then, Osmin can run as an RTOS for some uses by (i) completing all variable-time kernel operations such as memory allocation during startup, and (ii) disabling any problematic API calls thereafter.

Because Dauug|36 does not have any I/O yet, neither can Osmin. The "external storage" for the purpose of booting the OS, loading programs, and I/O within programs, is via the electrical simulation's `PVIO` (paravirtualized I/O) instruction. This does no good for real-world machines, but it provides usable scaffolding for implementing Osmin, which in turn provides scaffolding to test the ensuing I/O subsystem and device drivers.

Regression testing for Osmin is messy on account of the electrical simulation's low instruction throughput. Wiping a million-word data memory at startup would take more than two hours, so the instruction set and simulation offer a `MEMSET` pseudo-instruction that can fill contiguous memory in a moment. But no tests presently monitor the kernel's internal data structures for symptomless bugs. Such tests should be written in parallel with kernel development. To speed testing and maximize independence from possibly-defective kernel source, these checks should be written in C to run as part of the simulator itself instead of run slowly within a simulation.

5.2 Portions That Are Substantially Missing

For three reasons, I have not undertaken to build a physical computer yet. First, real costs are incurred constructing partial machines. Second, SRAM supply strain during the pandemic threatened component availability for more than two computers. Third, the maker community is quick to clone open-source CPUs, and I do not want to penalize early adopters with the shortcomings of a premature release. So although a partial, proof-of-concept implementation has been possible for a while, I intend to finish a fully-standalone, fully-solder-defined specification before I build machine zero. The rest of this section describes this prerequisite work.

5.2.1 Clock Distribution

The largest risk to early prototypes is that their clock skew may be too large for the design to work. The problem is one of precision, not speed, so reducing the clock frequency will not help. There is only a 1.5 ns tolerance in clock arrival time between adjacent RAMs. This corresponds to about nine inches of track length, which is easy to balance, but there are other concerns. Even if rising edges arrive simultaneously at two RAMs, how will the parts themselves interpret these arrivals? What will the effects of noise be? There are good reasons why most 7400-series ICs *intentionally* are not as fast as the 74AUC family. Thus work remains to address clock skew and clock distribution.

5.2.2 I/O Subsystem

One of computer history's long-running memes is the practice of attaching crippling, afterthought I/O to otherwise-promising CPUs. At least we have an example of what not to do—fritter a 36-bit CPU to bang I/O one bit at a time. So instead, much-smaller, dedicated controllers will transfer data between I/O RAM and serially-attached peripherals. Each I/O controller will be very simple, consisting primarily of an SRAM IC that is configured as a finite state machine (FSM), and another SRAM IC for I/O buffer memory. This design frees the CPU to run other code while the controller handles I/O transfers. Support is planned for the Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) bus specifications, to which most modern hardware can attach directly or via commonly-available adapters.

The I/O subsystem is the *internal firewall* (Section 2) between the solder-defined, user-auditable minicomputer and the opaque, proprietary, uncertain world of commodity peripherals. The only thing a peripheral is electrically able to do to the minicomputer is drive *one wire* high or low in reply to an I/O bus strobe. All interpretation of what high or low means is deferred until the bit safely reaches the CPU. Although SPI and I^2C support bus sharing, solder-defined hygiene demands that all peripherals use a dedicated bus to prevent them from interfering with or eavesdropping on each other. Because most minicomputers should support at least eight connected peripherals, serial buses' comparatively small component and footprint demand will be advantageous.

An I/O controller may service several buses. Its two RAMs will be segmented by address input bits according to bus number, so no electrical path will exist from any peripheral to another peripheral's buffer memory. Moreover, if there is an error, corruption, or crash caused by defective firmware for one peripheral, the controller's other peripherals can recover.[30] Most Dauug|36 computers should have at least two I/O controllers, so that transfers between two peripherals can keep both busy.

5.2.3 Device Drivers

At least two small device drivers will be written and tested before a physical computer is built, to validate the I/O subsystem's ability to transfer serial data. One driver may support a real-time clock such as Maxim's DS3231, and the other may support interactive testing via Ethernet packets or an RS-232 device. Device drivers will be implemented in three layers:

I/O device layer. The highest (closest to user code) of the three driver layers, the I/O device layer will be written specifically for the peripheral it controls. Its job is to produce and interpret serial packets that are exchanged with the peripheral.

I/O kernel layer. The middle of the three driver layers, the I/O kernel layer will exchange packets between the CPU's data memory and an I/O controller's separate buffer memory. Less frequently, the kernel layer will schedule the controller's work,

[30]The CPU may need to shut down an infinite FSM loop before another peripheral can use the controller.

respond to controller requests, patch controller firmware to specify timing, packet size, and other configuration, and download firmware to the controller. Generic across all devices, this layer will be a permanent part of the kernel.

I/O bus layer. The lowest (closest to peripheral) of the three driver layers, the I/O bus layer will bit bang data exchanges between an I/O controller's buffer memory and a serially-connected peripheral. Except for on-the-fly patching to select among SPI- and I^{2}C-supported configurations, this layer is generic to its bus protocol, not customized to a device.

5.2.4 Firmware Loader

When the system powers up, about 100 million bits of firmware must be retrieved from their serial NOR flash memory to the arithmetic logic unit's 20 SRAM ICs, 2 control decoder RAMs, and the code memory RAM. This firmware is pushed to the RAMs via D flip-flops that afterward "vanish" by disabling their outputs. These flip-flops are already present in the design in order to account for their drag on board capacitance, and most are visible in Figure 2. But the control logic that drives these flip-flops and the SRAM write signals remains to be designed.

5.2.5 Proof of Separation Between Programs

Although a mathematical, semi-formal, or formal proof that the hardware, firmware, and kernel are immune to hacking and malware is not anticipated prior to the first physical prototype, the usefulness of having such a proof must not be forgotten. Such proof must specify and derive from a set of assumptions and expectations, including how immunity is defined for the architecture.[31]

5.2.6 Circuit Board Finalization, Routing, and Fabrication

Once the Dauug|36 specification is able to self-load and run from power-up when simulated, bypass capacitors will be added and placed, and regulated power will be supplied and distributed.[32] Connectors and logic will be added for testing the board and flashing firmware. By design, the minicomputer cannot alter its firmware without onsite assistance. JTAG won't help for circuit testing or firmware loading, because JTAG and hand-solderability are rarely offered in the same IC package.

Substantial extra work may be needed to find or write software to route circuit board tracks. Although open-source hardware should not necessitate use of closed-source tools to design and maintain, I have not found an open-source, off-the-shelf electronic design automation tool that can tackle a project of this size.

After these tasks, machine zero will be ready to assemble and test.

[31] The proof should cover all time when power is applied. One assumption that may be necessary is that the kernel is retrieved from the firmware loader's serial NOR flash memory instead of from an attached device outside the security perimeter. Otherwise, the proof may have to extend over filesystems, device drivers, and non-solder-defined hardware.

[32] Power consumption is hard to estimate from datasheets and simulation, so it will be measured from a prototype.

6 Implications and Conclusions

6.1 Security Advantages

6.1.1 Open Hardware and Open firmware for Running Open Software

I have read many forum posts complaining that a purchased device's firmware is closed-source, inaccessible to the owner, and/or encrypted. All Dauug|36 firmware is fully open-source, accessible to those who install it, and nothing is encrypted. Even the S-boxes for the ALU's substitution-permutation networks are fully derived via a transparent algorithm. The system owner has absolute and final authority over every *bit* of the firmware, and the purpose of each bit is explained in the documentation.

The openness of the architecture's firmware extends also to its electrical design and implementation. Its only secrets are the die-level design of discounted logic ICs—synchronous static RAMs and basic glue logic—with clear interface definitions and publicly available datasheets. There are no secret functionalities in the architecture, no vendor lock-in, no encrypted or closed-source firmware, no license fees to build, use, or modify, no purpose of use limitations, no patents on any technology originating from me, and no infringements on the owner's right to repair.

All hardware is visually and electrically inspectable after manufacture and purchase. Firmware is easily accessible for inspection and modification via appropriate in-person tools.

6.1.2 Security Perimeter for Solder-Defined Logic

A security perimeter surrounds the CPU, memory subsystem, firmware loader, and I/O subsystem, inside which there is no purchased complex logic such as microprocessors, FPGAs, PLDs, or ASICs. The only perimeter crossing points are the serial buses between the I/O controllers and peripherals, with no serial bus attaching to more than one peripheral. The serial bus boundaries form an internal firewall, such that any peripheral's defects that might be exploitable cannot propagate into other parts of the system.

Some peripherals, such as mass storage, are not available as solder-defined subsystems. The operating system should use encryption mechanisms within the CPU that are designed to prevent, for example, a rogue disk controller from reading or modifying programs contained on its disk. This protection can be at low computational cost on account of the `MIX` and `XIM` opcodes. Encryption keys would be installed by the firmware loader from the serial flash memory. These keys need only be kept secret from the peripherals they defend against.

6.1.3 Memory Hygiene for Hardware

The architecture is intrinsically immune to certain memory exploits. A complete absence of DRAM eliminates the RowHammer class of leaky-capacitor exploits, and may also reduce susceptibility to radiation upsets. Absence of cache memory and speculative execution also rules out Spectre- and Meltdown-type attacks and reduces the range of side channel attacks that may be possible.

The firmware as written has no opcode that can result in data exchange between the stack and registers. Figure 2 does not suggest firmware modifications that could read data from the stack, short of having to pass through the code RAM. There are two electrical routes to write to stack memory, which would require complicit firmware using an elaborate control decoder scheme. The shorter route passes through via flip-flops “a,” “t,” and “c” in Figure 2.

In the presence of well-behaved firmware, the only access to stack memory is via the `CALL`, `RETURN`, and their variants. No privilege escalation can result from stack overflow, and there is no possibility of harmful stack underflow if the kernel adheres to certain rules.

It is not possible to branch to locations in code memory that are not already present in a branch instruction in code memory. This allows exclusive ownership of portions of code memory by various users, along with arbitrary sharing of code memory as may be supported and permitted by the operating system.

Unprivileged users are subject to paged virtual memory for data segregation.

The I/O controller’s buffer memory and finite-state machine memory will be electrically segregated on a per-serial-bus (that is, per-peripheral) basis.

For *software*, the architecture contributes little if anything new to memory hygiene.

6.1.4 Control of the CPU

All nonprivileged opcodes are such that they cannot cause privilege escalation on their own. They would need a complicit human, operating system, or control decoder RAM to do this. Of these three routes, the control decoder RAM is easiest to defend against—its firmware is tiny—and a complicit human is the most difficult to defend against.

The minicomputer contains no persistent state within the security perimeter except for the firmware’s serial flash memory, which the CPU does not have write access to.

6.1.5 Arithmetic

The arithmetic logic unit has effective means for detecting out-of-range conditions for addition, subtraction, multiplication, arithmetic shifts, and absolute value. These means can look back to the last time the `R`(ange) flag was cleared, therefore permitting long computation sequences to run without overhead to check for out-of-range conditions. These arithmetic improvements can help redeem out-of-favor programming languages such as C and assembler, which current arithmetic hygiene expectations for traditional architectures have made unsuitable for secure programming.

6.1.6 Why Tamper Resistance Is Out of Scope

Some vendors and some cybersecurity practitioners may dismiss this architecture over concern that it is not tamper-resistant. This would be a valid complaint for electronics that safeguard a physical asset against on-site compromises, such as locks to prevent a handgun from firing, a missile from detonating, or a bank card from exposing a private key. The complaint may also be valid for small personal platforms that could be stolen, such as smartphones. But for fixed assets such as desktop computers in homes and offices, routers in network closets, industrial controllers, and farm machinery, the presence of on-location, technically-sophisticated adversaries is possibly not today's top threat. More likely, the adversaries will be equipment manufacturers, part suppliers, governments of interested jurisdictions, or international criminals.

Buyers who require tamper resistance for Dauug|36 are at liberty to add it, subject to their weight, size, and cost budgets, using physical barriers and surveillance that are suitable for their needs. But technological security controls have both active and passive failures, and in the case of tamper resistance, an active failure usually infringes on the buyer's rights.

6.2 Performance and Applications

Dauug|36 pursues the world's first "gold standard" for transparently functioning, fully auditable, user-constructable controllers, CPUs, and minicomputers for integrity- and confidentiality-critical missions. This 36-bit architecture executes 16 MIPS in simulation and is free of wait states, and has an eventual goal of 20 MIPS, despite a total absence of purchased complex logic such as microprocessors, FPGAs, PLDs, and ASICs within its security perimeter. Relative to today's off-the-shelf alternatives, the architecture's drawbacks are added size, cost, and power consumption, inability to run existing software, and a ceiling on installable memory.[33] Summary specifications appear in Table 22.

Table 23 lists potentially compatible and incompatible uses for Dauug|36. The minicomputer will 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, container cranes, chemical plants, engines, and turbines. It will 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 will also permit desktop use such as 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. Dauug|36 is not small or fast enough for smartphones or video.

For servers, Dauug|36's applicability will depend on workload and surrounding components, especially software. A web platform intended for an eight-core CPU

[33]The present board and available ICs support 8Mi $\times$ 36 and 4Mi $\times$ 36 bits of data and code memory respectively.

Table 22: Specifications

System classification	solder-defined minicomputer
Logic family	SRAM with 74AUC
Memory protection	paged virtual memory
Multitasking	cooperative or preemptive
Word size	36 bits
CPU speed	16–20 MIPS
Maximum data RAM	8Mi $\times$ 36 bits
Maximum code RAM	4Mi $\times$ 36 bits
Registers per program	512
Programs ready to run	256
I/O buses	SPI and I^2C
Hardware license	CC BY 4.0 Intl.
Firmware license	CC BY 4.0 Intl.
Operating system	Osmin or owner-supplied
Manufacturer	anyone

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 emergent architecture will run fine. For more than 75 years, engineers have proven stunningly adept at making systems fit within computing hardware constraints when sufficient motivation and talent are present.

Acknowledgments

This work has been partially sponsored through the Assured Digital Microelectronics Education & Training Ecosystem (ADMETE) grant awarded through Air Force Research Laboratory (AFRL).

Thank you to Glenn Alexandra Ochsner of Miami University for feedback and suggestions related to the architecture's operation, terminology, drawings, and understandability since its inception.

Figure colors were selected with guidance from [8].

References

[1] Marc W. Abel. 2022. Parallel Multiplier Synthesis Software. Harvard Dataverse. (Sep. 4, 2022). doi: 10.7910/DVN/DABIBJ

Table 23: Dauug|36 potential applications

Fast enough for	**Too slow for**
process controls	contemporary Web surfing
controlling objects that move	AI model training
peripheral and device controllers	image recognition
light- to moderate-use servers	fast raster or vector graphics
modest Ethernet switches	fast symmetric cryptography
telephony	fast asymmetric cryptography
hardened desktop applications	video compression
electronic mail	computational biology

[2] Marc W. Abel. 2022. *A Solder-Defined Computer Architecture for Backdoor and Malware Resistance.* Ph.D. Dissertation. Wright State University, Fairborn, OH, USA. `http://rave.ohiolink.edu/etdc/view?acc_num=wright167489700770166`

[3] Marc W. Abel. 2023. *The Dauug House: Dauug|36 Minicomputer Documentation.* `https://dauug.cs.wright.edu/`

[4] Olivier Bailleux. 2016. A CPU made of ROMs. Watched Apr. 4, 2020 from `https://www.youtube.com/watch?v=J-pyCxMg-xg`

[5] Olivier Bailleux. 2016. The Gray-1, a homebrew CPU exclusively composed of memory. Retrieved Apr. 4, 2020 from `https://bailleux.net/pub/ob-project-gray1.pdf`

[6] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. 2013. Stealthy Dopant-Level Hardware Trojans. In *Cryptogr. Hardw. Embed. Syst. – CHES 2013*, August 18–23, 2013, Santa Barbara, CA, USA. *Lect. Notes Comput. Sci.* 8086 (July 30, 2013), Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-40349-1_12

[7] Sergey Bratus, Travis Goodspeed, Peter C. Johnson, Sean W. Smith, and Ryan Speers. 2012. Perimeter-crossing buses: A new attack surface for embedded systems. In *Proc. 7th Workshop Embed. Syst. Secur. (WESS 2012)*, (Tampere, Finland).

[8] Cynthia A. Brewer, ed. 2013. ColorBrewer 2.0: Color Advice for Cartography. Geography, Pennsylvania State University. University Park, PA, USA. `https://colorbrewer2.org`

[9] Robert G. Brown, Dirk Eddelbuettel, and David Bauer. 2020. Dieharder: A random number test suite. Retrieved Sep. 6, 2022 from `http://webhome.phy.duke.edu/~rgb/General/dieharder.php`

[10] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr, eds. 2016. *A Dictionary of Computer Science* (7th ed.). Oxford University Press, Oxford, England.

[11] CTS Labs. 2018. Severe Security Advisory on AMD Processors. CTS Labs, Tel Aviv, Israel.

[12] Roger Dannenberg, Will Dormann, David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, Timothy Wilson. 2010. *As-if Infinitely Ranged Integer Model* (2nd ed.). Technical Note CMU/SEI-2010-TN-008. Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA. doi: 10.1184/R1/6572048.v1

[13] Digital Equipment Corporation. 1964. *Programmed Data Processor-6 Handbook*, p. 17. Retrievevd Feb. 13, 2024 from `https://bitsavers.org/pdf/dec/pdp6/F-65_PDP-6_Handbook_Aug64.pdf`

[14] Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. At *Black Hat USA 2018*, (Las Vegas, NV), white paper.

[15] 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.

[16] Mark Ermolov. 2020. Intel x86 root of trust: loss of trust. (Mar. 2020). Retrieved Apr. 4, 2020 from `https://blog.ptsecurity.com/2020/03/intelx86-root-of-trust-loss-of-trust.html`

[17] Larry Geenemeier. 2017. The Pentagon's seek and destroy mission for counterfeit electronics. (Apr. 28, 2017). `https://www.scientificamerican.com/article/the-pentagon-rsquo-s-seek-and-destroy-mission-for-counterfeit-electronics/`

[18] Maurizio Gavardoni. 2016. Microchip AN2340: Immunity of MEMS oscillators to mechanical stresses. (Nov. 7, 2016). Retrieved Feb. 27, 2024 from `https://ww1.microchip.com/downloads/en/Appnotes/00002340A.pdf`

[19] Mirko Holler, Manuel Guizar-Sicairos, Esther H. R. Tsai, Roberto Dinapoli, Elisabeth Müller, Oliver Bunk, Jörg Raabe, and Gabriel Aeppli. 2017. High-resolution non-destructive three-dimensional imaging of integrated circuits. *Nature* 543 (Mar. 16, 2017), 402–417. doi: 10.1038/nature21698

[20] International Organization for Standardization. 2018. ISO/IEC 27000:2018(E). *Information technology – Security techniques – Information security management systems – Overview and vocabulary.* Retrieved Jul. 14, 2020 from `https://standards.iso.org/ittf/PubliclyAvailableStandards/c073906_ISO_IEC_27000_2018_E.zip`

[21] Dmitry Janushkevich. 2020. *The Fake Cisco: Hunting for Backdoors in Counterfeit Cisco Devices*. Version 1.0, Jul. 2020. Retrieved Jul. 19, 2020 from `https://labs.f-secure.com/assets/BlogFiles/2020-07-the-fake-cisco.pdf`

[22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symp. Secur. Priv.*, (San Francisco, CA), IEEE, 1–19. doi: 10.1109/SP.2019.00002

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. 27th USENIX Secur. Symp.*, (Baltimore, MD), USENIX Association, 973–990. doi: 10.1145/3357033

[24] Eric Love, Yier Jin, and Yiorgos Makris. 2011. Enhancing security via provably trustworthy hardware intellectual property. In *2011 IEEE Int. Symp. Hardw.-Oriented Secur. Trust*, June 5–6, 2011, San Diego, CA, USA. IEEE, New York, NY, USA, 12–17. doi: 10.1109/HST.2011.5954988

[25] Eric Schlaepfer. 2016. The MOnSter 6502. Retrieved Sep. 6, 2022 from `https://monster6502.com`

[26] Onur Mutlu and Jeremie S. Kim. 2019. RowHammer: A retrospective. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*. doi: 10.1109/TCAD.2019.2915318

[27] National Security Agency Advanced Network Technology Division. 2008. NSA ANT catalog. Retrieved Apr. 4, 2020 from `https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf`

[28] National Security Agency. 2022. Software memory security. (Nov. 10, 2022). Retrieved Nov. 14, 2022 from `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`

[29] Michael Pompeo. 2020. The tide is turning toward trusted 5G vendors. Press Statement by the Secretary of State. Jun. 24, 2020, (Washington, DC). Retrieved Sep. 5, 2022 from `https://2017-2021.state.gov/the-tide-is-turning-toward-trusted-5g-vendors/index.html`

[30] 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 Apr. 4, 2020 from `https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it`

[31] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.* 74, 2 (Mar. 1953), 358–366. doi: 10.1090/s0002-9947-1953-0053041-6

[32] Joanna Rutkowska. 2015. Intel x86 considered harmful. (Oct. 2015). Retrieved Apr. 4, 2020 from `https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf`

`https://blog.invisiblethings.org/papers/2015/state_harmful.pdf`

[33] David Schor. 2022. IEDM 2022: Did We Just Witness The Death Of SRAM? *WikiChip Fuse: Chips & Semi News*, (Dec. 14, 2022). Retrieved Feb. 27, 2024 from `https://fuse.wikichip.org/news/7343/iedm-2022-did-we-just-witness-the-death-of-sram/`

[34] Mischa Schwartz and Jeremiah Hayes. 2008. A history of transatlantic cables. *IEEE Commun. Mag.* 46, 9 (Sep. 12, 2008), 42–48. doi: 10.1109/MCOM.2008.4623705

[35] 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, USA, 112–118, 126–135.

[36] Eugene Howard Spafford. 1989. The Internet worm: Crisis and aftermath. *Commun. ACM* 32, 6 (June 1989), 678–687. doi: 10.1145/63526.63527

[37] Clifford Paul Stoll. 1988. Stalking the wily hacker. *Commun. ACM* 31, 5 (May 1988), 484–497. doi: 10.1145/42411.42412

[38] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annu. Tech. Conf.*, (Boston, MA), USENIX Association, 213–225.

[39] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (Aug. 19984), 761–763. doi: 10.1145/358198.358210

[40] Warren Toomey (Ed.). 2017. Homebrew computers web-ring. Retrieved Sep. 6, 2022 from `https://www.homebrewcpuring.org`

[41] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. *41st IEEE Symp. Secur. and Priv. (S&P 20)*, (pandemic all-digital conference). doi: 10.1109/SP40000.2020.00089

[42] Adam Waksman. 2014. *Producing Trustworthy Hardware Using Untrusted Components, Personnel and Resources.* Ph.D. Dissertation. Columbia University, New York, NY, USA. doi: 10.7916/D8N014PX

2 370001 851351

Dauug|36 is designed from scratch to preclude exploitable hardware defects that often arise from longstanding custom (e.g. arithmetic wraparound), undue complexity (e.g. Spectre, RowHammer, Meltdown), or intentional backdoors (e.g. Clipper).

Dauug|36 does not rely for trustworthiness on foreign countries or semiconductor companies—regardless of where located—because the system owner's own soldering and firmware determines the computer's logical connectivity and operation.

There isn't a microprocessor or anything like one (FPGA, PLD, or ASIC) **anywhere in the design.**

Compare the following Dauug|36 characteristics to any other computer architecture on the planet, and decide for yourself.

- Sticky, consistent overrange flag for arithmetic
- Stratified opcodes for heterogeneous register signedness
- No privilege escalation via stack
- No access to stack except via CALL and RETURN variants
- Code and stack memory inaccessible via LD and STO opcodes
- No branch to addresses not hardcoded in CALL or JUMP
- Faultless paged virtual memory without overcommit
- No privilege escalation via CPU
- No DRAM or DRAM-associated vulnerabilities
- No VLSI complex logic except in attached peripherals
- Every peripheral isolated to its own bus and buffer memory
- No CPU persistent state except for one firmware IC
- No MEMS oscillator for age- and frequency-selected attacks
- No firmware modification without physical access
- No parts that can't be hand-soldered and probed afterward
- No secret functionality
- No unexplainable S-box constants
- No vendor lock-in
- No encrypted or closed-source firmware
- No license fees to build, use, or modify
- No purpose-of-use limitations
- No planned obsolescence
- No right-to-repair infringements