

Wright State University

CORE Scholar

---

[Browse all Theses and Dissertations](#)

[Theses and Dissertations](#)

---

2022

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

Marc W. Abel

*Wright State University*

Follow this and additional works at: [https://corescholar.libraries.wright.edu/etd\\_all](https://corescholar.libraries.wright.edu/etd_all)



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

---

### Repository Citation

Abel, Marc W., "A Solder-Defined Computer Architecture for Backdoor and Malware Resistance" (2022).  
*Browse all Theses and Dissertations*. 2662.

[https://corescholar.libraries.wright.edu/etd\\_all/2662](https://corescholar.libraries.wright.edu/etd_all/2662)

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact [library-corescholar@wright.edu](mailto:library-corescholar@wright.edu).

A SOLDER-DEFINED COMPUTER ARCHITECTURE FOR  
BACKDOOR AND MALWARE RESISTANCE

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

By

MARC W. ABEL

B.S., California Institute of Technology, 1991

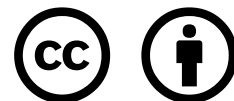
---

2022

Wright State University

Copyright © 2020–2022 Marc W. Abel.

This work is licensed under a Creative Commons  
Attribution 4.0 International License. For more information,  
see <https://creativecommons.org/licenses/by/4.0/>



Wright State University  
GRADUATE SCHOOL

November 30, 2022

I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Marc W. Abel ENTITLED A Solder-Defined Computer Architecture for Backdoor and Malware Resistance BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

---

Travis E. Doom, Ph.D.  
Dissertation Director

---

Thomas Wischgoll, Ph.D.  
Director, Computer Science and Engineering Ph.D. Program

---

Shu Schiller, Ph.D.  
Interim Dean of the Graduate School

Committee on  
Final Examination

---

Travis E. Doom, Ph.D.

---

Jack S. N. Jean, Ph.D.

---

Michael L. Raymer, Ph.D.

---

Krishnaprasad Thirunarayan, Ph.D.

---

Vincent A. Schmidt, Ph.D.



## ABSTRACT

Abel, Marc W. Ph.D. Department of Computer Science and Engineering, Wright State University, 2022. A Solder-Defined Computer Architecture for Backdoor and Malware Resistance.

This research is about securing control of those devices we most depend on for integrity and confidentiality. An emerging concern is that complex integrated circuits may be subject to exploitable defects or backdoors, and measures for inspection and audit of these chips are neither supported nor scalable. One approach for providing a “supply chain firewall” may be to forgo such components, and instead to build central processing units (CPUs) and other complex logic from simple, generic parts. This work investigates the capability and speed ceiling when open-source hardware methodologies are fused with maker-scale assembly tools and visible-scale final inspection.

The author has designed, and demonstrated in simulation, a 36-bit CPU and protected memory subsystem that use only synchronous static random access memory (SRAM) and trivial glue logic integrated circuits as components. The design presently lacks preemptive multitasking, ability to load firmware into the SRAMs used as logic elements, and input/output. Strategies are presented for adding these missing subsystems, again using only SRAM and trivial glue logic. A load-store architecture is employed with four clock cycles per instruction. Simulations indicate that a clock speed of at least 64 MHz is probable, corresponding to 16 million instructions per second (16 MIPS), despite the architecture containing no microprocessors, field programmable gate arrays, programmable logic devices, application specific integrated circuits, or other purchased complex logic.

The lower speed, larger size, higher power consumption, and higher cost of an

“SRAM minicomputer” compared with traditional microcontrollers may be offset by the fully open architecture—hardware and firmware—along with more rigorous user control, reliability, transparency, and auditability of the system. SRAM logic is also particularly well suited for building arithmetic logic units, and can implement complex operations such as population count, a hash function for associative arrays, or a pseudorandom number generator with good statistical properties in as few as eight clock cycles per 36-bit word processed. 36-bit unsigned multiplication can be implemented in software in 47 instructions or fewer (188 clock cycles). A general theory is developed for fast SRAM parallel multipliers should they be needed.

All tools and work product of this research are available online with open-source licenses.

# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Problem statement . . . . .	1
1.2	Seeking a solution . . . . .	4
1.3	Research questions . . . . .	11
1.4	Original results . . . . .	18
<b>2</b>	<b>Definitions</b>	<b>21</b>
<b>3</b>	<b>Components</b>	<b>25</b>
3.1	Logic family selection . . . . .	25
3.2	SRAMs as electrical components . . . . .	29
3.2.1	Asynchronous SRAM . . . . .	32
3.2.2	Synchronous SRAM . . . . .	35
3.2.3	Dual-ported SRAM . . . . .	37
3.3	Traditional logic ICs . . . . .	38
3.4	Clock skew with mixed logic families . . . . .	39
3.5	Derived components . . . . .	43
3.5.1	Multiplexers . . . . .	43
3.5.2	Shift registers . . . . .	43
3.5.3	Counters . . . . .	43
3.6	Non-computing component security . . . . .	47
3.6.1	Firmware reservoir . . . . .	47
3.6.2	Oscillators and clock buffers . . . . .	49
3.6.3	Peripherals . . . . .	51
3.6.4	Capacitors . . . . .	52
<b>4</b>	<b>Logic blocks for SRAM ALUs</b>	<b>53</b>
4.1	Hierarchy of ALU capabilities . . . . .	53
4.2	Simple lookup elements . . . . .	55



4.3	Arbitrary geometry adders . . . . .	56
4.4	Carry-skip adders . . . . .	57
4.5	Swizzlers . . . . .	63
4.6	Logarithmic shifters . . . . .	64
4.7	Semi-swizzlers . . . . .	65
4.8	Substitution-permutation networks . . . . .	66
4.9	Fast multipliers . . . . .	67
4.10	Open question: medium-speed multipliers . . . . .	71
<b>5</b>	<b>Three-layer ALU structure</b>	<b>73</b>
5.1	Superpositions of SRAM logic blocks . . . . .	73
5.2	Word sizes for minicomputer architectures . . . . .	78
5.3	CPU flags . . . . .	82
5.4	SRAM bit assignments . . . . .	85
5.5	Alternate diagram for ALU . . . . .	85
<b>6</b>	<b>Two-layer ALU structure</b>	<b>89</b>
6.1	An elegant two-layer ALU for 36-bit words . . . . .	89
6.2	A tiny ALU for 18-bit words . . . . .	97
<b>7</b>	<b>A three-layer, 36-bit ALU firmware</b>	<b>99</b>
7.1	What is SRAM ALU firmware? . . . . .	99
7.2	ALU opcodes and their implementations . . . . .	101
7.2.1	Additive opcodes . . . . .	103
7.2.2	Bitwise boolean opcodes . . . . .	106
7.2.3	Compare opcodes . . . . .	108
7.2.4	Shift and rotate opcodes . . . . .	109
7.2.5	Multiply opcodes . . . . .	111
7.2.6	NUDGE instruction . . . . .	115
7.2.7	Bit permute opcodes . . . . .	117
7.2.8	Mix opcodes . . . . .	125
7.2.9	Simple unary instructions . . . . .	135
7.2.10	Stacked unary instructions . . . . .	137
7.3	ALU operations and their implementations . . . . .	138
7.3.1	$\alpha$ layer operation . . . . .	139
7.3.2	$\beta$ layer operation . . . . .	146
7.3.3	$\gamma$ layer operation . . . . .	149

7.3.4	$\theta$ operation . . . . .	154
7.3.5	$\zeta$ operation . . . . .	156
7.3.6	Simple unary operations . . . . .	160
7.3.7	Stacked unary operations . . . . .	161
7.4	Leading and trailing bit manipulation . . . . .	167
7.5	A reference implementation . . . . .	171
7.6	Future work . . . . .	171
<b>8</b>	<b>A solder-defined CPU with protected memory</b>	<b>175</b>
8.1	Physical characteristics of the CPU . . . . .	179
8.2	Machine word structure . . . . .	185
8.3	Register organization . . . . .	186
8.3.1	Register splitting and reverse subtraction . . . . .	187
8.4	Memory organization . . . . .	187
8.4.1	Data memory organization . . . . .	188
8.4.2	Code memory organization . . . . .	191
8.4.3	Stack memory organization . . . . .	193
8.5	Machine instruction format . . . . .	195
8.5.1	Alternative instruction formats . . . . .	197
8.6	CPU topology and instruction cycle . . . . .	199
8.6.1	Instruction cycle for ALU opcodes . . . . .	202
8.6.2	Memory access opcodes and routes . . . . .	207
8.6.3	Flip-flops not involved in memory accesses . . . . .	211
8.7	Control unit . . . . .	213
8.7.1	Clock driver . . . . .	213
8.7.2	Click counter . . . . .	213
8.7.3	Control decoder . . . . .	215
8.8	Simplicity and scale of the CPU . . . . .	218
<b>9</b>	<b>Forthcoming subsystems</b>	<b>221</b>
9.1	Preemptive multitasking . . . . .	222
9.2	Firmware loader . . . . .	225
9.2.1	Option 1: Purchased complex logic . . . . .	227
9.2.2	Option 2: Hardwired logic after NOR flash . . . . .	229
9.2.3	Option 3: Finite state machine after NOR flash . . . . .	231
9.2.4	Option 4: Parallel NOR flash finite state machine . . . . .	233
9.3	Input and output . . . . .	234

<b>10 Fast parallel multipliers</b>	<b>241</b>
10.1 Background . . . . .	242
10.2 Notation and definitions . . . . .	246
10.3 Generation of partial products . . . . .	247
10.3.1 Unsigned partial products . . . . .	248
10.3.2 Signed partial products . . . . .	250
10.3.3 Mixed-signage multipliers . . . . .	253
10.4 Partial product summation . . . . .	255
10.4.1 Carry-save addition . . . . .	256
10.4.2 Carry-skip addition . . . . .	259
10.4.3 Fast subproduct totals . . . . .	264
10.4.4 Multilayer carry-skip addition . . . . .	269
10.5 Implication and contribution . . . . .	275
<b>11 Minicomputer implementation</b>	<b>277</b>
11.1 Firmware implementation . . . . .	279
11.2 Assembler . . . . .	283
11.2.1 What the assembler includes . . . . .	283
11.2.2 Future assembler features . . . . .	286
11.3 Netlist definition and processing . . . . .	288
11.3.1 Off-the-shelf electronic design automation software . . . . .	288
11.3.2 A typewritten netlist . . . . .	291
11.3.3 Component placement . . . . .	306
11.3.4 Netlist summary information output . . . . .	308
11.3.5 Estimating propagation delay between pins . . . . .	312
11.4 Electrical and timing simulation . . . . .	315
11.4.1 Off-the-shelf simulation software . . . . .	315
11.4.2 Electrical simulator description . . . . .	317
11.4.3 Simulator test script semantics and example . . . . .	321
11.4.4 Simulator output example . . . . .	327
11.4.5 Hazards, limitations, and next steps of simulation . . . . .	330
<b>12 Opposing viewpoints</b>	<b>333</b>
<b>13 Findings, motivation, significance</b>	<b>341</b>
13.1 Major findings to date . . . . .	341
13.2 Motivation for this work . . . . .	344

13.3	Security advantages of the architecture . . . . .	350
13.4	Drawbacks of the architecture . . . . .	354
13.5	Significance of this work . . . . .	355
13.6	Future work and timeframe for availability . . . . .	358
<b>A</b>	<b>Assembly language conventions</b>	<b>359</b>
A.1	Source code character set . . . . .	359
A.2	Comments . . . . .	359
A.3	Numbers . . . . .	361
A.4	Identifiers . . . . .	362
A.5	Abbreviating keywords . . . . .	363
A.6	Declaring registers . . . . .	363
A.7	Overriding register signedness . . . . .	364
A.8	Permutation notations . . . . .	364
<b>B</b>	<b>Instruction reference</b>	<b>367</b>
A	Add . . . . .	368
ABS	Absolute value . . . . .	368
AC	Add with carry . . . . .	369
AND	AND . . . . .	369
ASL	Arithmetic shift left . . . . .	370
ASR	Arithmetic shift right . . . . .	371
AW	Add with wrap . . . . .	372
AWC	Add with wrap and carry . . . . .	373
BO-	Brighten ones . . . . .	374
BOUND	Bound . . . . .	374
BZ-	Brighten zeros . . . . .	375
CALL	Call . . . . .	376
CLO	Count leading ones . . . . .	377
CLZ	Count leading zeros . . . . .	377
CMP	Compare . . . . .	378
CRF	Clear range flag . . . . .	378
CTO	Count trailing ones . . . . .	379
CTZ	Count trailing zeros . . . . .	379
CX	Check and extend . . . . .	380
DSL	Double shift left . . . . .	380
EO-	Erase ones . . . . .	381

EZ-	Erase zeros . . . . .	381
FABS	Fast absolute value . . . . .	382
FO-	Find one . . . . .	382
FZ-	Find zero . . . . .	383
GO-	Grow one . . . . .	383
GZ-	Grow zero . . . . .	384
HALT	Halt . . . . .	384
IPSR	Instruction pointer shift register . . . . .	385
JUMP	Jump . . . . .	386
LANR	Left and not right . . . . .	387
LAS	Logical assignment . . . . .	388
LFSR	Linear feedback shift register . . . . .	389
LO-	Light ones . . . . .	390
LONR	Left or not right . . . . .	390
LSL	Logical shift left . . . . .	391
LSR	Logical shift right . . . . .	391
LZ-	Light zeros . . . . .	392
MAX	Maximum . . . . .	392
MH	Multiply high . . . . .	393
MHNS	Multiply high no shift . . . . .	394
MIN	Minimum . . . . .	394
MIRD	Mirrored decrement . . . . .	395
MIRI	Mirrored increment . . . . .	395
MIX	Mix . . . . .	396
ML	Multiply low . . . . .	398
NAND	NAND . . . . .	398
NAS	Numeric assignment . . . . .	399
NOP	No operation . . . . .	399
NOR	NOR . . . . .	400
NOT	NOT . . . . .	400
NUDGE	Nudge . . . . .	401
OR	OR . . . . .	401
PARTY	Parity . . . . .	402
PAT	Permute across tribbles . . . . .	402
PAIT	Permute across and inside tribbles . . . . .	404
PIT	Permute inside tribbles . . . . .	404

POPC	Popcount . . . . .	405
PRL	Prepare to rotate left . . . . .	405
PRR	Prepare to rotate right . . . . .	406
PSL	Prepare to shift left . . . . .	406
PSR	Prepare to shift right . . . . .	407
RANL	Right and not left . . . . .	407
ROL	Rotate left . . . . .	408
RONL	Right or not left . . . . .	408
RS	Reverse subtract . . . . .	409
RSC	Reverse subtract with carry . . . . .	410
RSW	Reverse subtract with wrap . . . . .	411
RSWC	Reverse subtract with wrap and carry . . . . .	412
RTGL	Rotate T going left . . . . .	412
RTGR	Rotate T going right . . . . .	413
S	Subtract . . . . .	413
SC	Subtract with carry . . . . .	414
STGL	Shift T going left . . . . .	414
STGR	Shift T going right . . . . .	415
SW	Subtract with wrap . . . . .	415
SWC	Subtract with wrap and carry . . . . .	416
SWIZ	Swizzle . . . . .	416
TXOR	Transposing XOR . . . . .	417
XIM	Undo mix . . . . .	418
XNOR	XNOR . . . . .	418
XOR	XOR . . . . .	419
XPOLY	XOR polynomial on T flag . . . . .	420
<b>C</b>	<b>Advance corrections</b>	<b>421</b>
C.1	Instruction format fields to move . . . . .	421
C.2	ALU operation to be deleted . . . . .	422
C.3	Physical address format to change . . . . .	423
<b>D</b>	<b>What's where in the source tree</b>	<b>425</b>
<b>References</b>		<b>431</b>

# List of Figures

1.1	To increase security, reduce complexity. . . . .	3
1.2	Proposed SRAM CPU pipeline, before it was shortened to figure 1.3. . . . .	16
1.3	Reworked data paths after shortening the CPU cycle. . . . .	17
3.1	BGA layout of a 36-bit ALU using asynchronous SRAMs. . . . .	33
3.2	Call stack depth counter using LFSRs. . . . .	45
3.3	Instruction pointer incrementer using AUC glue logic. . . . .	46
4.1	Simple lookup element. . . . .	56
4.2	Arbitrary-geometry addition. . . . .	56
4.3	Subword carry decisions for 3-bit addition. . . . .	58
4.4	Two-layer carry-skip adder. . . . .	58
4.5	Two-layer carry-skip adder with old carry via top. . . . .	60
4.6	Two-layer bidirectional carry-skip adder. . . . .	60
4.7	Three-layer carry-skip adder. . . . .	62
4.8	$4 \times 4$ swizzler. . . . .	63
4.9	16-bit logarithmic shifter. . . . .	65
4.10	$4 \times 4$ semi-swizzler or half-shifter. . . . .	66
4.11	4-bit S-box. . . . .	66
4.12	16-bit substitution-permutation network. . . . .	67
5.1	Block diagram of a 36-bit ALU. . . . .	75
5.2	Superposition of major components of a three-layer, 36-bit ALU. . . . .	76
5.3	Bit transposition as a square matrix reflection. . . . .	77
5.4	Bit slice and carry propagation SRAMs for a 36-bit, three-layer ALU. . . . .	79
5.5	CPU flag approximate logic. . . . .	84
5.6	Block diagram of a 36-bit ALU (landscape). . . . .	88
6.1	Data signals for a 36-bit, two-layer ALU. . . . .	91

6.2	Control signals for a 36-bit, two-layer ALU. . . . .	92
6.3	Data signals for a 18-bit, two-layer ALU. . . . .	98
7.1	Decomposition of a random 36-bit permutation. . . . .	119
8.1	CPU floorplan on facing pages. Actual size. . . . .	180
8.2	Unbroken CPU floorplan. Actual size. . . . .	182
8.3	Virtual address format for data memory. . . . .	189
8.4	Page table RAM's input and output bit assignments. . . . .	189
8.5	Physical address format for data memory. . . . .	189
8.6	Principal data paths of the CPU. . . . .	200
8.7	The clock driver uses eight buffer ICs with stages wired in parallel. . . . .	214
9.1	A single-RAM finite state machine with hardware to load its firmware. . . . .	231
10.1	Subproduct summation using carry-save adders. . . . .	258
10.2	Two-layer, radix-agnostic, carry-skip adder. . . . .	263
10.3	Subproduct summation using carry-save and carry-skip adders. . . . .	264
10.4	64-bit either-signage multiplier using six layers. . . . .	267
10.5	Three-layer carry-skip adder. . . . .	270
10.6	Hierarchical scheme for a five-layer, 28-term carry-skip adder. . . . .	272
11.1	Partial KiCad drawing of the ALU datapath. . . . .	289
11.2	Partial KiCad drawing of the ALU power connections. . . . .	290
11.3	A drawing of the chapter 8 minicomputer's non-power connections. . . . .	310
11.4	A drawing by KiCad of the chapter 8 minicomputer. . . . .	311
11.5	Visual illustration of track length estimates. . . . .	314



# List of Tables

1.1	Categories of vulnerability-inducing hardware irregularities. . . . .	2
1.2	Category II (unplanned and unexpected) hardware irregularities. . . .	2
1.3	Category III (maliciously introduced) hardware irregularities. . . . .	4
1.4	Proposed VLSI supply controls for Category III backdoors. . . . .	5
1.5	Potential applications for the proposed architecture. . . . .	10
5.1	CPU flag meanings. . . . .	82
5.2	ALU SRAM input bit assignments. . . . .	86
5.3	ALU SRAM output bit assignments. . . . .	87
6.1	Suggested operations for a 36-bit, 2-layer ALU. . . . .	94
7.1	Principal opcodes and macros for a 36-bit, 3-layer ALU. . . . .	102
7.2	Additive instructions and their implementations. . . . .	104
7.3	Bitwise boolean instructions and their implementations. . . . .	107
7.4	Comparison instructions and their implementations. . . . .	108
7.5	Shift and rotate instructions and their implementations. . . . .	110
7.6	Other ALU instructions and their implementations. . . . .	112
7.7	Frequently-used swizzle operations for $\beta$ .swz. . . . .	114
7.8	Directly-available tribble permutations. . . . .	124
7.9	Compact set of tribble permutations for two instructions. . . . .	124
7.10	Unary instructions and their implementations. . . . .	136
7.11	ALU $\alpha$ layer operations. . . . .	140
7.12	ALU $\beta$ layer operations. . . . .	147
7.13	ALU $\gamma$ layer operations. . . . .	150
7.14	ALU $\theta$ RAM operations. . . . .	154
7.15	ALU $\zeta$ RAM operations. . . . .	157
7.16	ALU simple unary operations. . . . .	161
7.17	ALU stacked unary operations. . . . .	162
7.18	Assembler macros that (mainly) use stacked unary operations. . . . .	163

7.19	Leading and trailing bit manipulation macros. . . . .	168
7.20	Visual index to the leading and trailing bit manipulation macros. . .	169
8.1	These major subsystems remain for design and test. . . . .	177
8.2	Non-ALU instructions. . . . .	178
8.3	Bill of materials for a simulated CPU that ran listing 8.1. . . . .	179
8.4	Non-ALU RAMs visible in CPU floorplan. . . . .	184
8.5	A new CPU instruction starts every fourth clock cycle. . . . .	203
8.6	Memory-read opcodes and their circuitous routes. . . . .	208
8.7	Memory-write opcodes and their register-to-memory routes. . . . .	208
8.8	Control decoder assigned bits. . . . .	216
9.1	Data exchange flip-flops from the firmware loader to the CPU. . . . .	227
10.1	Number of $256\text{Ki} \times 18$ SRAMs needed for various multipliers. . . . .	254
10.2	SRAM multiplier widths as a function of latency and SRAM size. . .	268
10.3	Computation of the 27 carry decisions for figure 10.6. . . . .	271
10.4	Metrics of hierarchical carry-skip unsigned multipliers. . . . .	274
11.1	Number of instructions required for some common tasks. . . . .	278
11.2	Tally of the minicomputer firmware size, measured in rows. . . . .	279
11.3	Overcurrent conflicts at the nets of $\alpha_4$ 's input pins. . . . .	331
A.1	Peculiar uses for various ASCII characters. . . . .	360
A.2	Digits for bases up to 64. . . . .	362
A.3	Keyword abbreviations. . . . .	363
B.1	Tribble permutation operations. . . . .	403
B.2	Swizzle operations. . . . .	417
D.1	<code>code/</code> Implementation directory root. . . . .	425
D.2	<code>code/asm/</code> Assembler and virtual machine. . . . .	426
D.3	<code>code/consts/</code> Constants that represent opcodes and operations. . . .	426
D.4	<code>code/firmware/</code> Modules to compute SRAM firmware. . . . .	426
D.5	<code>code/logic-solver/</code> Synthesize optimal SN74AUC-series glue logic. .	427
D.6	<code>code/misc/</code> Helper functions, constants, and macros. . . . .	427
D.7	<code>code/netlist/</code> “Electrical source code” of the minicomputer. . . . .	428
D.8	<code>code/netsim/</code> “Electrical object code” of the minicomputer. . . . .	429
D.9	<code>code/vm/</code> Testing for assembler and ALU firmware. . . . .	430

# List of Listings

7.1	36 subtraction opcodes. . . . .	105
7.2	Assembler code for 36-bit multiplication. 47 instructions are executed. . . . .	113
7.3	Using the XIM instruction to hash a four-word object. . . . .	127
7.4	This two-instruction PRNG withstands all Dieharder tests. . . . .	128
7.5	First portion of Dieharder output evaluating the PRNG of listing 7.4. . . . .	130
7.6	An explanation of the LFSR and XPOLY polynomials. . . . .	132
7.7	Python 3 specification of S-boxes for the MIX and XIM opcodes. . . . .	134
8.1	Program to compute the largest 36-bit Fibonacci number. . . . .	176
8.2	Four “bookend” instructions for call stack safety. . . . .	195
11.1	Command line options for <code>vm</code> tool. . . . .	281
11.2	Manual ALU test of a five-bit logical shift left. The bits to be shifted appear at input L. Input R has the number of positions to shift available at all subwords. The shifted word appears at output $\gamma$ . . . . .	281
11.3	Virtual machine output running the Fibonacci program of listing 8.1. . . . .	285
11.4	Instruction pointer incremter (figure 3.3) implemented. . . . .	304
11.5	Component placement syntax showing four circuit board tiles. . . . .	307
11.6	Connectivity syntax example with five signal paths. . . . .	312
11.7	Simulator script to start CPU and run Fibonacci program. . . . .	322
11.8	Final portion of simulator output with Fibonacci program. . . . .	328

# Acknowledgments

Nothing this page can hold could adequately thank everyone I leaned on through more than ten years at Wright State University. It's been a deep honor to study at this school, and I thank and commend our caring and devoted students, staff, faculty, administration, trustees, sponsors, and anyone I neglected to include on this list.

To my brothers and sister on the signature sheet, your faith in and tolerance of me has been a great support and pleasure. I hope our capers together will not end soon.

To the Institute of Electrical and Electronics Engineers, the Wikimedia Foundation, and their respective communities, thank you for your curation and publication of human advancements. I have depended on your work throughout.

To my longtime fans, all of you know who you are, and I owe you each a more personal salute than the moment it would have taken to add your name to this page. I am grateful for your long interest in my work as a privacy advocate, and predictably, I am withholding every one of your names. I love every one of you.



# 1

## Overview

### 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 1.1.

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 [Seacord14], and increasing the separation between the instructions that programmers write and the silicon executing them [Dannenberg10].

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 [Bratus12, CTS18, Ermolov17, Ermolov20, Kocher19, Lipp18, Mutlu19, Rutkowska15a, Tatar18, VanBulck20]. Table 1.2 lists a few well-known cases. They have whimsical names like “hidden God

**Table 1.1:** Categories of vulnerability-inducing hardware irregularities.

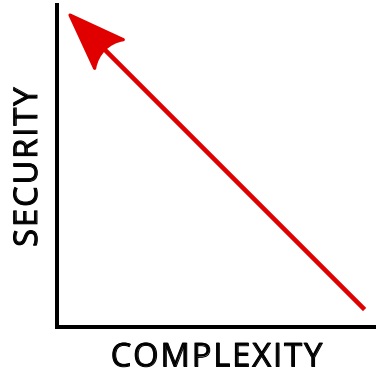
	<b>Category I</b>	<b>Category II</b>	<b>Category III</b>
<b>Origin</b>	purposeful	unexpected	malicious
<b>Example</b>	arithmetic wrap	RowHammer	hidden backdoor
<b>Software workaround?</b>	yes	no	no
<b>VLSI architect can fix?</b>	yes	yes	no
<b>Supply chain owner can fix?</b>	yes	yes	yes

**Table 1.2:** Some Category II (unplanned and unexpected) hardware irregularities.

<b>when</b>	<b>architecture</b>	<b>name</b>	<b>synopsis</b>
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

mode,” “Meltdown,” and “Silent Bob is Silent.” These pitfalls can sidestep the most attentive programmer’s precautions, and they can surface years after a hardware, operating system, or application platform is no longer supported by its original producer. To first order, vulnerabilities from Category II irregularities cannot be solved through software, but require architectural changes to future generations of VLSI by the supplier.

Category 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 1.1 during my explanation.



**Figure 1.1:** To increase security, reduce complexity.

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 Morris disproved this presumption [Stoll88, Spafford89].

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 [Becker13, CTS18, Domas18, NSA08, Portnoy17]. Table 1.3 lists some examples. Such vulnerabilities are of particular urgency when national boundaries are crossed [Pompeo20]. Another model that falls under Category III is the outright counterfeiting of critical equipment by unknown manufacturers, such as reported in July 2020 for network switches [Janushkevich20], or for many years for integrated circuits for aviation and military use [Greenemeier17]. Another Category III example would be a manufacturer shipping genuine semiconductors that are known to not meet specifications for use in U.S. weapons. I have listened to chilling firsthand accounts from a longstanding friend who was dismissed for refusing to sign off on such shipments.

Table 1.3 slips in an important point about *right to repair*. ISO 27000 [ISO18] incorporates *availability* into the scope of *information security*, so when a manufacturer



**Table 1.3:** 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
ZTE	5G cellular infrastructure	potential for China influence

elects to make a deployed tractor unavailable to 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 suppression 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 satisfactorily either via software or through hopes that a supplier will abstain from inserting them. Instead, the buyer needs to extend some form of control over the manufacturing process and/or the final assembled product.

## 1.2 Seeking a solution

Various proposals have been made for safeguarding the VLSI supply chain against intentional defect introduction, including [Pompeo20, Waksman14, Love11, Holler17] which are summarized in table 1.4. The suggestions range from never trading with adversaries, to securing a multi-billion dollar supply chain across hundreds or perhaps

**Table 1.4:** 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 dissertation	complex logic to be built by end user

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 within specific settings.

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

nections 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 dissertation is that it is possible to build an easy-to-reproduce minicomputer<sup>1</sup> today that meets the following informally stated supply chain tamper resistance criteria:

### **Supply chain tamper resistance criteria**

1. The computer's electronic components are simple and generic enough, that it would not be practical for an adversary to introduce exploitable defects through the component supply chain.
2. Each electronic component already exists in the marketplace for other uses, and at least two manufacturers currently produce any particular component.
3. The computer can be assembled using current surface-mount practices, using either manual or automated placement and soldering.
4. The assembled computer can be inspected against open-source specifications, resulting in a modest level of confidence that exploitable defects were not injected into the product.
5. The computer can be warranted by the manufacturer to exactly match identified open-source specifications.
6. The computer can be warranted by the manufacturer to be assembled from components supplied by the buyer, or from a specified bill of materials.

---

<sup>1</sup>This term is defined on page 22.

These tamper resistance criteria, although helpful for excluding Category III hardware irregularities that are maliciously introduced, cannot directly address exploitable defects from Category I's unintended consequences of purposeful design features or Category II's unplanned and unexpected behaviors. Without offering a blueprint for how to address Categories I and II, here are a few of my outcome expectations for computers and their CPUs:

### **Behavioral expectations of computers and CPUs**

1. Computers must not run operating systems. Instead, operating systems must run computers.
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,<sup>2</sup> 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.

---

<sup>2</sup>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 [Rice53].

8. A computer's security must not be fragile in the presence of malformed input.
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 the operating system are one and the same.
12. A computer must provide hashing, pseudorandom number generation, and cryptography capabilities consistent with 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 replaceable by its owner in its as-used form.
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. I have two sorobans in my lab that generally meet the parameters of this list, even though they don't use electricity to do arithmetic. Two TRS-80s here also run fine, and although lacking, conform closer to the above than the 64-bit machines that this dissertation was written on.

This dissertation's emergent minicomputer seeks a 36-bit word size, memory protection, and preemptive multitasking. The target speed is twenty million instructions

per second (20 MIPS), and the eventual board footprint is estimated as  $20 \times 30$  cm. There does not appear to be much margin to improve either metric using currently available components.

In light of the low ceiling on instruction throughput, the design seeks a high yield of computational work as each CPU instruction is executed. There are two prongs to this strategy. First, all instructions have a common small number of clock cycles. For the design advanced, an 80 MHz clock is equivalent to a 20 MIPS computer. Second, the architecture is designed to minimize the number of instructions needed to accomplish a task. For example, a context switch from running one program to running a different program is principally a write to a register, and would take no more than 20 clock cycles. No registers or flags need to be spilled to RAM in order to switch programs. Another example is the generation of statistically robust pseudorandom numbers, which despite only requiring two instructions per 36-bit word generated, passes all components of a leading suite of random number generator tests [Brown20].

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

As the central hypothesis of this work is the constructability of a computer with a specified architecture using readily available parts, the best test of this hypothesis would be to attempt its construction and report the outcome. The distance in effort between producing a correct simulation and producing a functioning machine is not a large one, but there is a significant difference in how the two endpoints would be perceived. Although I hoped to report here that a working model has been built, I have neither reached that target, nor a full specification and functioning simulation

**Table 1.5:** Potential applications for the proposed architecture.

<b>fast enough for</b>	<b>too slow for</b>
<ul style="list-style-type: none"> <li>• hardened desktop applications</li> <li>• electronic mail</li> <li>• light- to moderate-use servers</li> <li>• controlling objects that move</li> <li>• process controls</li> <li>• peripheral and device controllers</li> <li>• telephony</li> <li>• modest Ethernet switches</li> </ul>	<ul style="list-style-type: none"> <li>• contemporary Web surfing</li> <li>• machine learning</li> <li>• image and video processing</li> <li>• fast raster or vector graphics</li> <li>• fast symmetric cryptography</li> <li>• fast asymmetric cryptography</li> <li>• self-driving vehicles</li> <li>• computational biology</li> </ul>

of a complete machine. My work and progress towards these ends in support of soon manifesting such a machine are the main subject of this dissertation.

Table 1.5 lists potentially compatible and incompatible uses for the emergent architecture. The minicomputer anticipated would be fast enough to control most systems that physically move: industrial and commercial devices, factory automation, electric grids, wells and pumps, heavy machinery, trains, dams, traffic lights, chemical plants, engines, and turbines. It would also be fast enough for many uses not involving motion, such as measurement and sensing, peripheral and device controllers, telephony, and even Ethernet switches to medium speeds. It would also permit desktop use writing and editing documents, making spreadsheets, sending and reading email, and writing and building software, although desktop software would need to be specifically written for or adapted to the architecture. The architecture is not small or fast enough for smartphones or video.

For servers, the applicability of the emergent architecture will depend on workload and surrounding components, especially software. A web platform intended for an eight-core CPU and 64 GB of RAM is not within reach of this technology. Even if an application on this scale could be accommodated, the sheer size of the software will often present a larger attack surface than the hardware it runs on. On the other hand, server applications specifically designed to run on and thoughtfully matched to

the 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.

## 1.3 Research questions

It is canon that VLSI is the key which during the 1980s unlocked all practical computing. The cost efficiencies of the microprocessor and its descendants teach the field that practicality and miniaturization cannot be separated, just as one cannot separate waves from particles or matter from energy. My suggestion to exclude semiconductor fabrication from certain stages of computer manufacturing in order to increase transparency and accountability would sacrifice much practicality. But how much practicality? Orders of magnitude, if practicality is measured by speed. But is speed the right metric? For video rendering, yes, speed is a critical metric. What about for treating wastewater? Or for tabulating votes? What kinds of applications are compatible with the architecture of this dissertation? Which are not? How flexible is the compatibility boundary, and how easy is it to influence which applications may be served and which may not? How good can a near-term “high bar” be with respect to optimizing production cost, attack surface, and supportable applications? My research did not directly tackle these questions, but it did produce performance estimates—which turned out to be anything but obvious—that inform reasoning as to what the a solder-defined architecture could be capable of supporting.

What are the economics of installing a solution that may not just run 1 000 times slower, but also cost 1 000 times more? And that’s just the hardware cost. But consider avoided costs: IBM Security reports in August 2022 that the cost to remedy data breaches of 50–60 million records averages \$387 million [IBM22]. Also, there are liabilities around the planet that could be even more serious, as evidenced by the August 4, 2020 explosion of 2 700 tons of ammonium nitrate in Beirut [Gambrell20].



The cause of this incident has been described as not relating to computing, but potential for other atomic weapon-scale explosions which might relate undoubtedly exists. Air handlers for biosafety level 4 laboratories are another application where a \$1 ARM processor may not be the architecture of choice. My research wasn't meant to yield an actuarial model for these questions, but it has shown that the one-off cost of a solder-defined minicomputer can be around \$1 000 for the physical machine, assuming the builder is donating his or her time. Assembly time can range from less than an hour to a few days, depending on volume and capital equipment.

Other questions stem from whether and how a solder-defined architecture is going to work. What are the essential system blocks just above the RAM level, and how can they interoperate? What table-based mechanisms exist that can reduce the number of cycles needed for multiplication, division, single-precision floating arithmetic, permutations, pseudorandom number generation, hashing for associative arrays, and other necessary functionality? Is it effective to include special hardware, such as fast multipliers, for some of this functionality when a significant increase in component count would result? Can practical encryption be implemented on a network-connected register machine with a word size much smaller than 128 bits? These questions are each explored, to different extents, in this document. Also, what special handicaps, beyond than those imposed by the speed of light, are introduced by decisions I made about my architecture? How serious are these handicaps, and what can be done to mitigate them?

What are the limits to cycle time and pipelining on SRAM-based CPUs? My topic proposal of August 2020 asked if it would be possible to achieve 10 MIPS throughput on a system if the basic logic gate has a best-case delay of 7 ns, assuming such features as protected memory and preemptive multitasking are mandatory. I also asked if 10 MIPS could be achieved, could it all be within the same thread, or would data hazards necessitate simultaneous multithreading—that is, interleaving instructions from two independent threads—to achieve that speed? By early 2021, changes

were made to the assumptions that underlied these questions. I abandoned using 44-pin  $64\text{Ki} \times 16$  asynchronous RAMs in favor of 100-pin  $256\text{Ki} \times 18$  synchronous RAMs. They cost more than quadruple, but access times fell from 10 ns to 5.5 ns and some interfacing was simplified. I upgraded the logic family for basic gates to use faster parts. I changed the components for getting firmware to the points it was needed on the board. I changed the power supply from 3.3 V to 2.5 V. I gave up on pipelining, dropped the idea of interleaved instructions, and sacrificed base plus offset addressing in exchange for fewer clock cycles per instruction. I simulated large circuits with known components in known layouts with reasonable track length and capacitance estimates. These shifts in the landscape more or less devoured my 10 MIPS questions, because it appears now that 20 MIPS is the current “stretch” goal. More recently, it occurred to me there may be bubbles in the CPU timing that could fold in a second thread of execution if they are distributed carefully. This arrangement is sometimes called a barrel processor, and although I am not ready to investigate it yet, the possibility is raised for a readily-constructable, 40 MIPS, single-CPU, solder-defined minicomputer.

Another question in my topic proposal was, if the CPU’s speed were to be drastically increased by sacrificing security-related features such as overrange checking and protected memory, can we arrive at the same net usefulness on smaller, cheaper hardware by employing a meticulously scrutinized interpreter for a language reminiscent of Python, Java, or possibly Lua? On one hand, this document intersperses some information about design elements of “abridged” solder-defined CPUs, with compacted circuits like two-layer carry-skip adders instead of three-layer, semi-swizzlers instead of swizzlers, 18-bit words instead of 36-bit words, asynchronous RAM instead of synchronous, and smaller physical layouts. But the principal focus of my present research looks at full machines, with the focus toward how fast they may run as opposed to how easily can they be built. Minimalist solder-defined hardware would be worthy of its own research and development along the line of “what would MacGyver build?”

The parallel question as to minimalist toolchains, I also find exciting and believe that much remains to be discovered. Multi-million-line compilers such as GCC, CLANG, and LLVM would be a great challenge to audit for security, and self-hosted compilers come with ledger-scale traceability problems with respect to ruling out the possibility of backdoors.

Wandering exploration is one of my favorite research methods. Many scientific discoveries happen serendipitously before questions can be formalized. Radio astronomy is a frequently cited example, born of an engineer who pondered interference to transatlantic radio conversations [Jansky33]. Likewise, the work of this dissertation led to several spontaneous findings. When I began, I presumed that a 32-bit mini-computer would be a good starting point for migrating applications to solder-defined machines. But after a few iterations of 32-bit ALU designs, it became evident that SRAM exhibits a natural optimality for a 36-bit word size. Who knew? And by apparent sheer coincidence, other system components such as primary storage turn out to be as readily obtained in the market for 36 bits as for 32 bits. That 36-bit words appear to be a preferred implementation is an important result with many practical consequences and advantages, but no one embarked to make such a discovery. No questions were ever posed on this subject—and had they been, they would probably have made biased assumptions and found in favor of 32 bits. What happened instead is, I set out to do a project, and its outcome was new understanding. The discovery was intentional, no more accidental than when a drift net catches fish, but the process was not guided by questions. Even the topic of what the best word size is could not have been anticipated, but what I learned turned out to be important. I set out on an expedition to catch something unknown, much like simulated annealing, self-organizing maps, and SAT solvers search meanderingly for optimal configurations and informative patterns.

The discovery that 36-bit words are a good specification was not an isolated event. I likewise observed that repeated design cycles of SRAM ALUs tend to “pull”

their architecture into very regular, dense networks like that drawn in figure 5.1 (p. 75). These networks offer a high ratio of computational expressivity to component count, and their proclivities stem from branches of mathematics, network science, and combinatorics unfamiliar to me and perhaps not widely explored at all. But the observation is noteworthy and consequential.

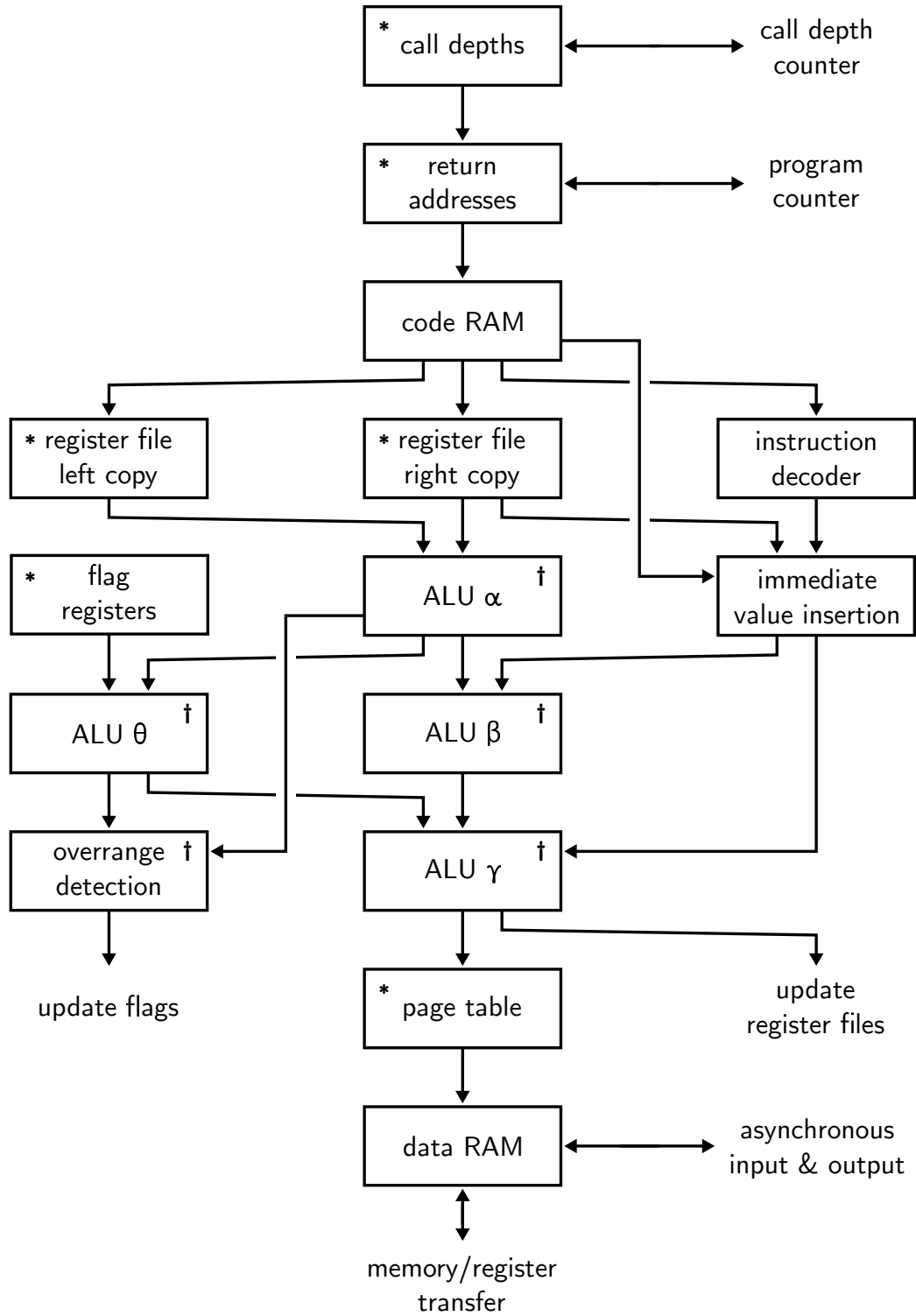
Likewise, I believe the SRAM-derived logarithmic shifter and semi-swizzler to both be novel, and their discovery was accidental. Earlier designs for my ALU involved shifters that used 8-to-1 multiplexer ICs and/or exponentially wasteful SRAM input configurations. They offered lackluster, but not worrisome, performance. But elsewhere in the ALU, I sought a mechanism for quickly changing the endianness (byte order) of a word, because that operation was one of several where I expected SRAM ALUs to improve speed. The endianness problem's solution came out isometric to what is now the first layer of the logarithmic shifter of figure 4.9 (p. 65), and the second layer was already drawn below it as a mechanism for fast permutations of 4-bit subwords. Seen together, these two provisions explained how to shift or rotate any number of bit positions in one machine cycle, while employing very few components. There are more instances like the foregoing, where contributing discoveries were made, not only without questions being asked first, but possibly because carefully targeted questions were never asked.

In my topic proposal, I wondered if similar gains would emerge in the my CPU, memory subsystem, and I/O subsystem. I had long dreamed of CPUs made from connected memories where the datapath is not a loop, but computations proceed instead by allowing data to reciprocate, as if regulated by a system of second-order differential equations, among a small number of RAMs with their data lines sharing some kind of common bus.<sup>3</sup> Alternatively, perhaps a bit-serial architecture can implement minicomputers with few components and yet somehow be fast enough to be beneficial.<sup>4</sup> No ideas had emerged as to how to implement either vision, so the

---

<sup>3</sup>I apparently did not consider that SRAMs do not read via their data lines.

<sup>4</sup>This idea resembles my present thoughts for the firmware loader and the I/O subsystem.



\* Additional input specifies the program that is running.

† Additional input comes from the instruction decoder.

**Figure 1.2:** Proposed SRAM CPU pipeline, before it was shortened to figure 1.3.

### letter codes for flip-flops

**A**ddress for code reads and writes

**B**ypass page table

**C**all (save return address)

**D**estination register

**F**rom incrementer

**I**nput from i/o

**J**ump and call destinations


**iM**mediate argument

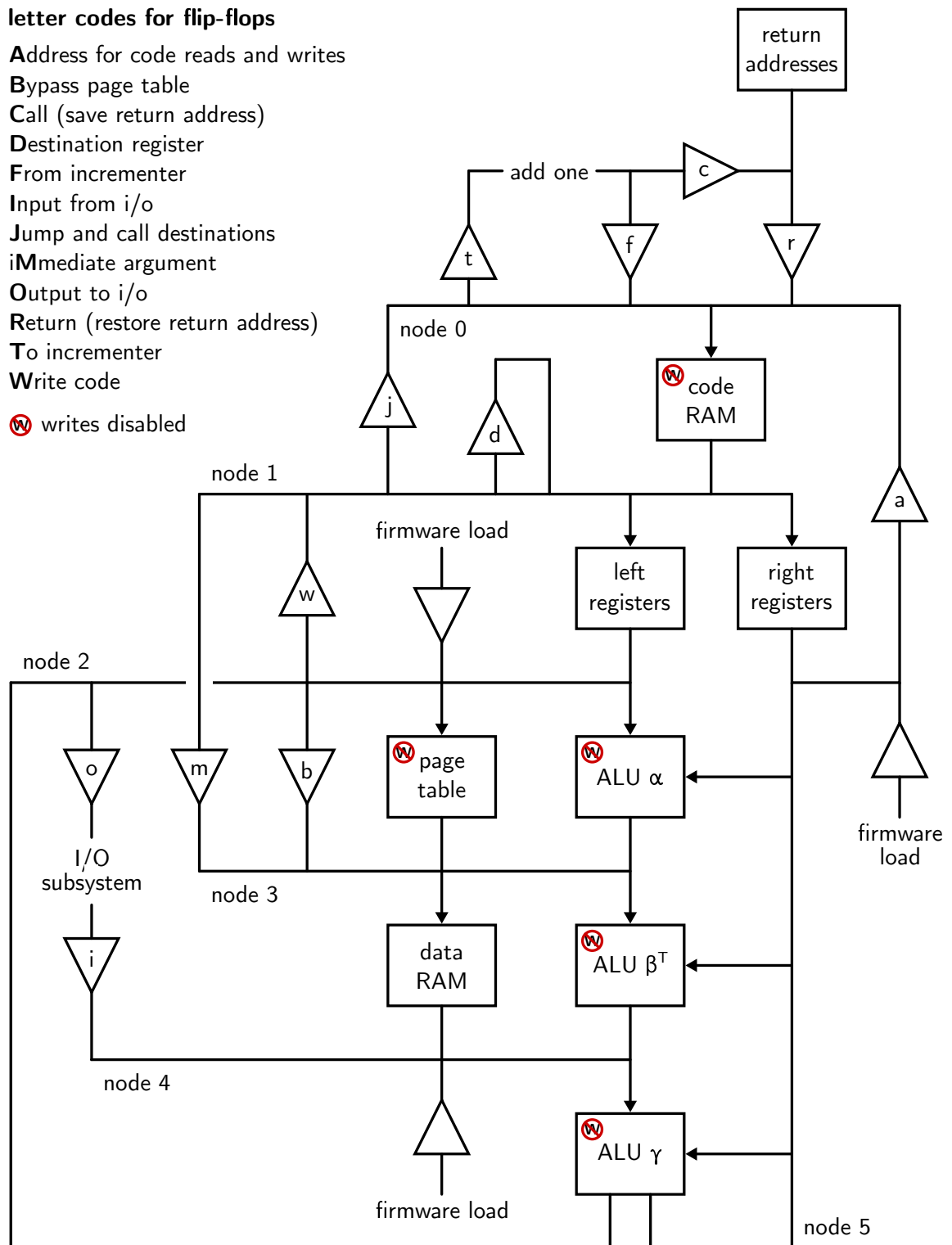
**O**utput to i/o

**R**eturn (restore return address)

**T**o incrementer

**W**rite code

 writes disabled



**Figure 1.3:** Reworked data paths after shortening the CPU cycle.

starting point for my design was an SRAM extrapolation of a rather traditional RISC pipeline, shown as figure 1.2. In the work that followed, my principal departure was to make the pipeline a little shorter and wider, which can be seen in figure 1.3.

In summary, questions from many directions attach to the work of this dissertation, in part because no recent precedent exists for practical solder-defined architectures. Although I can now report answers to several of these questions, it is too early to know which answers, if any, might be found useful by others. In the meantime, my continuing thrust to produce a high-quality, reproducible prototype is likely to spark insights into questions that have not come to mind yet.

## 1.4 Original results

Here are some places where my independent, incremental discoveries and writings may be firsts. I'll disclaim that computer science and computer engineering literature is vast, and robust concept indexing within the text of articles does not exist at this time. Also, seeking out previous art can be more expensive than unintended replication. The searches I have been able to make have turned up little. I believe that my original contributions include:

- Adaptation of carry-skip adders to SRAM logic, to include two-layer, three-layer, hybrids that combine two-layer and three-layer methods, and hierarchical schemes for carry-skip adders of more than three layers.
- Repurposing of SRAM-based carry propagation to also perform non-addition computations within existing adder circuits.
- Bidirectional and potentially simultaneous carry operations via SRAM.
- Offset-binary subproduct representations for SRAM multipliers (chapter 10).

- Generalization of carry-skip addition to accept more than two addends (chapter 10).
- The first explanation of carry-skip addition from the standpoint of mixed-radix arithmetic (chapter 10).
- The first measurements for SRAM multipliers to state number of components or number of gate delays for various word sizes and signages (chapter 10).
- The first software for synthesizing and validating SRAM multipliers [Abel22a].
- SRAM logarithmic shifters and semi-swizzlers (half-shifters).
- Superposed adder/shifter “textbook” three-layer SRAM ALUs [Abel21].
- Superposed adder/semi-swizzler for compact two-layer SRAM ALUs.
- Efficient bit permutation instructions for SRAM ALUs.
- Multidimensional S-box mixers superposed in SRAM ALUs.
- Firmware and code providing SRAM long multiplication and fast SRAM short multiplication.
- Fast SRAM hash functions for associative arrays.
- Fast SRAM pseudorandom number generators with good statistical behavior.
- A fast SRAM cipher round function.<sup>5</sup>
- Arithmetic stratification by signedness in hardware with uniform, practical, sticky arithmetic range checks [Abel22b]. This is at least novel for SRAM, and at least uncommon in general practice.

---

<sup>5</sup>Caveat emptor. I am not a cryptographer, and 36 bits form a worryingly small block.



- A pointer **NUDGE** operation that repurposes left-to-right carry propagation.
- A bar established for functionality and firmware that could be standard for SRAM RISC instruction sets.
- Proposed approaches to solder-defined preemptive multitasking.
- Proposed approaches to solder-defined firmware loaders.
- Proposed approaches to solder-defined I/O controllers.
- Progress toward a first “gold standard” for transparently functioning, fully auditable, user-constructable controllers, CPUs, and minicomputers for integrity- and confidentiality-critical missions.
- Open-source tools for design, representation, program assembly, logic simulation, layout, circuit simulation, and drawing of SRAM-based digital circuits.
- A warning of a possibility of exploitable backdoors in MEMS oscillator ICs.

## 2

# Definitions

This dissertation uses terms defined in [ISO18], plus the following definitions for concepts that do not have well-known terms.

**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.

**Chapter 8 minicomputer.** A fully-simulated minicomputer in the architecture of this dissertation, but still lacking multitasking, firmware loading, and I/O.

**Click.** The enumerated position of a clock cycle within a CPU cycle. The architecture of this dissertation uses four clicks, numbered click 0 through click 3.

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

**Code RAM.** In the architecture of this dissertation, one or more primary storage SRAMs that contain 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 the architecture of this dissertation, a CPU cycle is four clock cycles.

**Data RAM.** In the architecture of this dissertation, one or more primary storage SRAMs that are read or written by load or store instructions.

**Direct memory.** Memory where the location being accessed is beyond a program’s dynamic control, such as a register specified in a CPU instruction.

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

**Family.** In the architecture of this dissertation, a group of opcodes that have the same purpose but need different control signals due to minor variations.

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

**Indirect memory.** Memory where a program dynamically controls which location is accessed, such as data memory accessed via a register-specified address.

**Instruction.** In the architecture of this dissertation, 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.

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

**Macro.** In the architecture of this dissertation, a sequence of CPU instructions that to a programmer appears to be written as a single assembler instruction.

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

**Microcomputer.** According to custom and [Butterfield16], 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.** In the architecture of this dissertation, a nine-bit field in a CPU instruction that the control decoder uses to define and execute the instruction.

**Operand.** In the architecture of this dissertation, a field of an instruction containing a 9-bit register number, 18-bit integer constant, or 27-bit code address.

**Operation.** In the architecture of this dissertation, a computation task executed by one specific ALU SRAM or layer during the execution of an instruction.

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

**Primary storage.** Non-cache memory that is accessible to or contains individual CPU instructions. Ordinarily termed “RAM” outside this dissertation.

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

**RAM.** Within this dissertation, an informal abbreviation for SRAM.

**Reserved.** Unallocated. This term appears in table 5.3 (p. 87) and other tables where a resource is present, but no purpose or implementation is present as yet.

**SRAM.** Static RAM. Most SRAM in this dissertation 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.

**Supply-chain firewall.** A preventive control that protects a buyer from unwanted procurement of exploitable defects via the buyer’s supply chain.

**Support (verb).** To be used in something’s implementation. The statement “ $\gamma$ .pit supports PIT” means that  $\gamma$ .pit is a part of PIT’s implementation.

**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. In this dissertation, the architecture’s natural size is 36 bits for both code and data.

This dissertation employs backticks to indicate numeric radix (section A.3).

# 3

## Components

### 3.1 Logic family selection

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

Almost all CPUs made out of anything except silicon wafers lately have been somebody's avocation, such as the many homebrew CPUs cataloged in [Toomey17]. But the research of this dissertation was not undertaken for hobbyists. Trustworthy CPUs are needed to control dams, fly planes, protect attorney-client privilege, mix chemicals, leak secrets, coordinate troops, transfer funds, and retract the five-million-pound roof of Lucas Oil Stadium at the option of the Colts. All components selected must be for sale at scale, not scrounged from scrap, and they must remain available for the expected duration of manufacturing and servicing.

Here is a list of logic families we might be able to procure and use. Neither practicality nor seriousness was a requirement to appear on this list, because every choice here has important drawbacks. It is better to start with an overly imaginative list than to overlook a meritorious possibility. Because I had to select components early in my work, a few specifics below may no longer be current.

**Electromagnetic relays** have switching times between 0.1 and 20 ms, are large, costly, and have contacts that wear out. Relays generally have the wrong scale: a practical word processor will not run on relays. Relays offer certain benefits, such as resistance to electrostatic damage, and purring sounds when operated in large numbers [Dovgalyuk19].

**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 [Schwartz08].

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

**Transistors** in individual packages are barely within scale. The VML0806 package size is the smallest available, measuring 0.8 x 0.6 x 0.36 mm. An advantage to using discrete transistors is that no component sees more than one bit position, so slipping a hardware backdoor into the CPU unnoticed would be particularly difficult.<sup>1</sup> 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 [Schlaepfer16].

**7400 series** and other **glue logic** have largely been discontinued. NAND gates and inverters aren't a problem to find, but the famed 74181 4-bit ALU is gone, the 74150 16:1 multiplexer is gone, etc. Most remaining chips have slow specifications, obsolete supply voltages, limited temperature ranges, through-hole packages, and/or single sources. 4-bit adders, for example, are still manufactured, but their specs are so

---

<sup>1</sup>One possible backdoor would be to install several RF retro-reflectors like NSA's RAGEMASTER [NSA08] in parallel, or a single retro-reflector in combination with a software driver.

uncompetitive as to be suggestive for use as replacement parts only. Counter and shift register selection is equally dilapidated. As of 2020, even some leading manufacturers were distributing datasheets that appeared to be scanned from disco-era catalogs. The SN74AUC series, however, is very helpful. See section 3.3.

**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. 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. If anyone still makes stand-alone mask ROM, they are keeping very quiet about it.

**EPROM** with a parallel interface apparently comes from only one company as of 2020. 45 ns access time is available, requiring a 5V supply. Data retention was 10 years in vendor advertisements, but omitted from datasheets. [Bailleux16a] and [Bailleux16b] describe a CPU that uses EPROM as its principal logic family.

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

**NOR flash** with parallel interfaces is suitable for combinational logic, offering speeds to 55 ns. Storage density is not as extraordinary as NAND flash, but  $128\text{Mi} \times 8$  configurations are well represented by two manufacturers as of early 2020. Although access time is much slower than static RAM, the density offered can make NOR flash faster than SRAM for uses like finding transcendental functions (logs, sines, etc.) of single-precision floating-point numbers. Data 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. NOR flash is sufficiently reliable that error-correcting code is not required for most applications.

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

**NAND flash** is available with parallel interfaces, but data and address lines are shared. These devices aren't directly usable as combinational logic. Serial NAND flash with external logic could be used to feed firmware into SRAM-based ALUs. Periodic rewrites are required as with NOR flash. 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 [Mutlu19], as well as to cosmic ray and package decay soft errors. The sole upside to DRAM is that an oversupply resulting from strong demand makes it disproportionately inexpensive compared to better memory.

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

As a logic family, static RAM's main selling point is its speed, due in part to its large number of input and output pins relative to basic glue logic. 10 ns access time 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. As a sequential logic family using standalone components, SRAM offers the best combination of cost and computation speed available at present. 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 to use as logic.

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

**Programmable logic devices**, or **PLDs**, and **field programmable gate arrays**, or **FPGAs**, can implement CPUs but are not inspectable, 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.

## 3.2 SRAMs as electrical components

Static RAM is confusingly named, as here static means the RAM does not spontaneously change state, not that it uses a static charge to hold information. Dynamic RAM uses a static charge, and therefore spontaneously forgets its stored contents and

requires frequent refreshes. SRAM is said to be expensive, but this is only compared to DRAM. SRAM generally uses four to ten transistors to store each bit, while DRAM uses one transistor and one capacitor. SRAM is about 600 times as expensive per bit, so factors beyond transistor count influence price. Even so, the price of SRAM has fallen more than 100-fold since I first started programming computers in 1981.

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

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

The parallel interface we need to compute with SRAM is a problem at power-up, because no facility is included to load the data. The forward-feed nature of SRAM combinational logic makes it possible to progressively set many address lines to desired locations as initialization continues, but nothing connects within the layers to access the data lines—note these are separate from and work differently than the address lines—and write the memory. Some RAMs come with JTAG pins, which provide a serial boundary scan capability that might be repurposable for firmware loading, but

there are a few concerns.

The first concern is, we're most likely to find JTAG on synchronous SRAMs with ball grid array (BGA) packaging, because there is no physical means to reach the connections after parts are soldered. JTAG in this case replaces the diagnostic and verification functionality offered by a flying probe. But small-scale buyers and builders are likely to disfavor BGA components, because connections can't be visually or electrically inspected after soldering. Moreover, a pick-and-place machine with a bottom-facing camera would be needed to position BGAs prior to reflow soldering, because placement by hand would not be able to see the connections. Hand soldering BGAs is tedious and requires specialized equipment, while gull-wing and other surface mount parts that have leads are readily hand-soldered after a little practice. So a high percentage of minicomputers would be built with SRAMs that don't offer JTAG.

A second problem with considering JTAG that I have encountered is, datasheets and standards describe the function of SRAM JTAG pins only minimally and ambiguously. It is unclear from manufacturer to manufacturer exactly what access is available and in what direction. I would have had to experiment with some parts for a while before I could determine whether JTAG would or would not aid with firmware loading, and I would only be confident of the answer for the specific component lines that I have tried.

A third concern with JTAG is that its bit-serial interface could make firmware loading very slow when the minicomputer needs to boot.

A fourth concern is one of perception and acceptance. There may be people who either perceive a JTAG interface as complex logic who aren't willing to discount it, or perceive JTAG's out-of-band serial connections into a CPU's parts as an actual or potential backdoor in its own right.

With JTAG looking unpromising, the approach I have taken is to connect extra components at every SRAM data pin that can't otherwise be reached by firmware. Unfortunately, these pins number in the hundreds. The added components and track

length will consume power and board space, as well as add capacitance to the CPU and slow it down nominally. But even with this bad news, SRAM is still the best logic family I can point to for building solder-defined computers. Moreover, the overhead of the firmware loader hardware, although significant, isn't prohibitive. In figure 1.3 (p. 17), three “firmware load” annotations denote flip-flops that introduce firmware into the CPU's main data paths. Although these flip-flops do not attach to the data lines for all of the RAMs directly, other flip-flops that support specific CPU instructions are able to relay the firmware to the remaining locations.

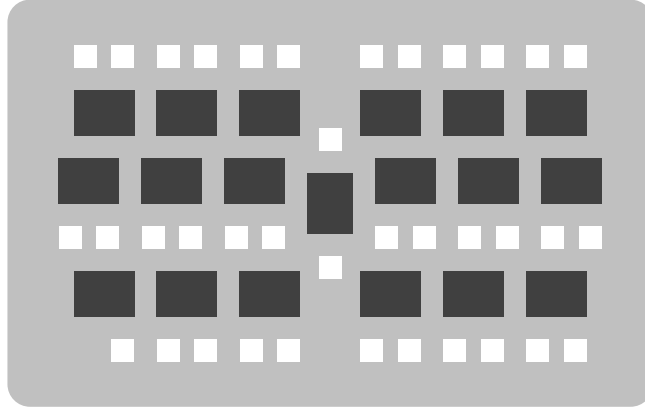
### 3.2.1 Asynchronous SRAM

In my topic proposal, I anticipated using asynchronous SRAM for most of the logic. These parts have no clock input. Instead, the parts are continuously read, and changes at the address lines will, after some nanoseconds, cause the data lines to indicate the contents of the addressed location. As of 2020, sizes are offered to about 32 Mibit, with access times around 10 ns. For more money, 7 ns is offered. An organization named JEDEC has standardized pinouts not only between manufacturers, but also across SRAM sizes. For primary storage RAMs, which are costly in larger sizes, careful planning enables one to make boards for computers in advance, then wait for an order before deciding how much to spend on code memory and data memory. Trends as of 2020 favor 3.3 V power, operating temperatures from  $-40$  to  $+85$  C, and a certain but imperfect degree of component interchangeability. The JEDEC pinouts are somewhat undermined by manufacturers ensuring they offer non-overlapping—although standardized—package geometries.

At least one manufacturer offers error correction code, or ECC, as an optional feature for its SRAM.<sup>2</sup> These RAMs with ECC are plug-in-replacements for ordinary RAMs, so it's possible to add ECC to a system design even after circuit boards have been made and early models have been tested. There isn't really a price or speed

---

<sup>2</sup>This particular SRAM uses four ECC bits per byte stored.



**Figure 3.1:** Ball grid array layout of a 36-bit ALU using  $64\text{Ki} \times 16$  asynchronous SRAMs. Black boxes indicate SRAMs. White boxes are analog switches for firmware loading. This circuit fits easily in the space of a bank card. The ALU did not have a  $\zeta$  RAM when this design was considered. Actual size.

penalty for specifying ECC, but it definitely limits supplier selection and appears to preclude dual sourcing. Also, as ECC RAM packages are only visually distinguishable by supplier markings and are not operationally distinguishable, it would be prudent to verify ECC's presence (by electron microscopy or comparing error rates in the presence of noise or radiation) in samples prior to claiming that computing hardware was built using ECC RAM. Although ECC RAM is interesting and may help win some orders for parts, SRAM ordinarily is highly dependable without ECC.

The smallest common size for asynchronous SRAM at the time of my topic proposal was  $64\text{Ki} \times 16$ , and this size worked very well for what I intended to do. One of the SRAMs ( $\alpha_5$ ) would have been a more expensive  $128\text{Ki} \times 16$  size to support overflow checking. Space to store ALU firmware was a little tight, so a few infrequent, less-essential operations were absent relative to today's design. The compactness of these parts appealed to me: figure 3.1 shows how a whole ALU fits easily in the space of a bank card for users who are comfortable with BGA packaging.

Because most of the ALU components produce outputs that are only a few bits wide, it would have been reasonable to use  $64\text{Ki} \times 8$  RAMs instead of  $64\text{Ki} \times 16$ .

There are a few reasons this isn't practical, but the key one is no one bothers to make the narrower size. This omission may seem foreign to early microcomputer users who dreamed of someday upgrading their systems to 64 kilobytes. Although 16-wide RAM uses double the transistors than we strictly require, there are tangible benefits to having the extra width. Fan-out gets heavy at various points, and being able to double up on some outputs may spare the cost and propagation delay of additional buffers. There is also a point in the datapath where the  $\gamma$  layer output must be duplicated between two otherwise segregated buses. The redundant outputs already included in today's SRAM make this need for isolation easier to accommodate.

A cool feature that JEDEC standardized for asynchronous SRAM, but unfortunately not for synchronous SRAM, is separate output enable pins for the two halves of their output words. Although this separability is not needed for my emergent architecture, chapter 6 describes how byte separation can provide for interesting, minimalist designs for two-layer ALUs.

Had the datapath through the CPU turned out to be long enough to warrant pipelining, some form of latch would have been needed between pipeline stages. This could have used latches, flip-flops, or synchronous SRAM with internal flip-flops or latches.

When I was planning to use asynchronous RAMs, I was not as informed as I am now about components for injecting firmware at the needed locations. What I had found at that point were analog switches such as PI3B3253 from Diodes Inc. and IDT's QS3VH253.<sup>3</sup> They are available with footprints as small as  $3 \times 2$  mm in BGA packages. The capacitance of either of these devices is of similar order to that of another RAM pin or even two, so there is a penalty. Each package has two copies of a 1-to-4 pass transistor mux-demux with an all-off option, so reaching the data lines of a  $64\text{Ki} \times 16$  RAM would require roughly two switch ICs on an amortized basis, plus additional logic farther away from where computations are done.

---

<sup>3</sup>Analog switch models and manufacturers mentioned are to facilitate identification only. I have tested none, and I endorse neither.

One benefit of using these analog switches is data can flow in either direction when they're in the on state. We can not only write to individual RAMs within a CPU or ALU, but read back their contents for diagnostic and development purposes. Of course we can also build a backdoor this way, but we're not interested. Such a backdoor would also be plainly visible. Concerns that this extra path to the RAMs may present a security concern can be mitigated by adding a non-resettable latch that locks out reads and writes once all tables are loaded and the system is ready to compute. This same latch could also enable the CPU clock, ensuring that once the minicomputer is running, there is no way an attacker can modify the loaded firmware. An LED indicator that indicates when the firmware is frozen may bolster comfort for some users.

### 3.2.2 Synchronous SRAM

I had ample reason to avoid synchronous SRAM. To start with, the smallest common size is  $256\text{Ki} \times 18$ , which is  $4\frac{1}{2}$  times larger than the already-adequate smallest asynchronous SRAMs. Because SRAM logic functions are implemented at the package level, denser RAMs don't translate to fewer components used. Either by coincidence or market strategy, synchronous RAMs also cost about  $4\frac{1}{2}$  times more, although the cost per bit does not change much. The synchronous parts have 100 pins, and pitch tightens from 0.80 mm to 0.65 mm, increasing my nervousness about soldering. The packages are larger, consuming not just board area, but also precious picofarad and nanosecond margins. The state complexity of synchronous RAMs is greater, requires study to understand, and could fuel speculation about backdoors. The system designer would need to know what linear and interleaved burst modes do, the difference between pipeline mode and flow-through mode, what the numbers 3-1-1-1 and 2-1-1-1 mean, what is meant by zero bus turnaround, and be able to understand state diagrams and timing graphs. To use these parts in a design, I would have to not only assimilate, but also simulate the behavior of these ICs.



As I considered how to build a minicomputer using asynchronous RAMs as logic, there were several points where they would have to communicate with clocked logic elsewhere, and my ideas for these interconnects didn't seem very smooth. A time came to ask *what if* the design used synchronous RAMs instead, how would that influence the machine, and would it shorten the time it would take me to produce a working model? I also felt intuitively that 5.5 ns was faster than 10 ns, so perhaps synchronous RAMs could allow me a faster CPU speed, provided those burst modes and 3-1-1-1s and 2-1-1-1s don't stand in the way somehow.

I started reading datasheets carefully from a few synchronous RAM manufacturers, and I gradually came to feel that I could tame these clocked devices to benefit the design. I chose flow-through mode, where inputs at the address lines are clocked, and the calculated result appears at the data lines after about 5.5 ns without another clock pulse. I estimated that a 7 ns budget could work for wiring delays and setup time, so a clock period of 12.5 ns (80 MHz) could be a goal.

A burst mode is offered on many synchronous SRAMs that can read or write up to four words per externally supplied address. In these SRAMs, the flip-flops for the two least significant address bits are configured as a two-bit presettable counter. A memory block that is aligned on a four-word boundary can be read or written by supplying an address from an outside source for the first word, and using the internal address counter to access the three remaining words. The control decoder described in section 8.7.3 finds use for this burst mode.

Zero bus turnaround, under various names, is offered by most manufacturers but not across all product lines. The idea is that if you write data to a RAM on one clock cycle, you can immediately read from any address on the very same cycle. This read would happen faster than the RAM can even get its storage array written, and the method involves a combination of clever innovations. A use for this feature is described in section 8.6.1. When the result of an instruction is stored to a register in my CPU, the same rising clock edge can fetch an operand for the next instruction

from the same or a different register. This is what enables my design to read from a register, spend three clock cycles getting through the ALU, then write the result back in only four cycles. Counting the steps would have it seem as if five cycles were needed, but the clock pulse that completes a register write is the same pulse that initiates a read. Part of the process that makes this work is that addresses to write are clocked to the RAM one cycle before the data, so the address to read is clocked with the data being written. The data bus direction switches from write to read at that clock edge. Conveniently, my CPU knows which register a result will be stored in well before the result is actually computed, so supplying the write address a cycle early is straightforward. Another innovation that enables zero bus turnaround is that although the SRAM functionally does the write operation first, it internally does the read first, or in the event of an address collision, returns a cached copy of the data before the write can occur.

As with asynchronous SRAM, synchronous SRAM is available with operating temperature specified as  $-40$  to  $+85$  C. Planning to stay with 3.3 V, I chose parts advertised for use at either 2.5 V or 3.3 V with the same timing specifications. This helped me in an unexpected way later, when I learned that my best option for glue logic can't be used above 2.7 V.

### **3.2.3 Dual-ported SRAM**

In many architectures including mine, ALU opcodes often require two operands. Obtaining them efficiently calls for two simultaneous register fetches. A natural way to do this is to use dual-ported SRAM, but I have some security concerns. Although dual-ported SRAM is commercially available, it's rather esoteric, comes at a higher price, and has few suppliers. Also, its lack of genericness makes its selection as a component somewhat conspicuous. A supply chain may be able to infer that that a particular component shipment is destined for an SRAM minicomputer, as well as the component's likely use as a register file. I opted instead to have the architecture

keep two identical copies of the registers in separate SRAM ICs.

Dual-ported SRAM may also find use in the I/O subsystem, simplifying sharing of buffer memory between a peripheral and the CPU. But I intend to use a memory access mechanism that uses ordinary SRAM.

Dual-ported SRAM may also aid in hardware multithreading on CPUs where alternate clock cycles execute two different programs. This may allow, for example, a 20 MIPS minicomputer design to be expanded to run two or even four programs at 20 MIPS each without significantly increasing the size of the CPU. The reason dual-ported SRAM becomes helpful is related to leveraging zero bus turnaround. Here again, alternatives should be considered that use ordinary SRAM.

### 3.3 Traditional logic ICs

Few people advocate building CPUs out of soldered components, so no one seems to mind when a basic gate's propagation delay is 5, 12, or even 25 ns. Unfortunately, delays on this scale don't fit into a machine with a 12.5 ns clock period. But there are a couple of high-speed logic families for small gates. One is called AVC and purportedly stands for "advanced very-low-voltage CMOS." Texas Instruments touted AVC as "the industry's first logic family to achieve maximum propagation delays of less than 2 ns at 2.5 V" [TI98]. An advantage of AVC is that certain components are available from more than one supplier, but overall, extremely few gates are offered.

There is one more family, called AUC ("advanced ultra-low-voltage CMOS"), and it's intended for 1.8 V designs but is operable from 0.8 V to 2.7 V [TI02]. Only Texas Instruments offers this logic family, so I am uncomfortable about not having a second source. On the other hand, the AUC logic family has been on the market for two decades, and nothing has appeared since that threatens to displace it. Most parts are available with leaded or leadless carriers,<sup>4</sup> and more gates are available than

---

<sup>4</sup>Pronounced LEED-ed, as in having wires. Not related to Pb (lead) in the periodic table.

for AVC.

AUC component specs are funky and may result from two design generations. For example, an individual NAND gate has a propagation delay of 1.4 ns, but the dual NAND package offers 1.0 ns. In this family are basic glue logic: AND, NAND, OR, NOR, XOR, inverters, buffers, tristate buffers, D flip-flops, and some others. A one-of-two decoder/demultiplexer may be also usefully provide “A implies B,” sometimes called “B or not A.” Sixteen-bit offerings include a buffer, inverting buffer, latch, D flip-flop, and bus transceiver. Not seen in the AUC family are counters, adders, shift registers, or similar conveniences, but the tiny size and appreciable speed of the items offered are immensely helpful. When compared against the four- and five-input NAND gates used in the Cray I for *all* of its combinational logic [Russell78], even the limited selection offered by TI’s AUC family feels luxurious.

### 3.4 Clock skew with mixed logic families

SRAM computing is primarily a process of functional composition, where a result obtained at the data lines of one SRAM (or more likely, a simultaneously-clocked layer of SRAMs) is delivered to the address lines of another SRAM (or layer), which is clocked at the same time. Every clock cycle advances a computational process one layer forward, with all RAMs using the same clock signal in the same phase.

If some RAM X feeds information to RAM Y, with both clocked simultaneously, then on any rising edge, X and Y are presented with a new computation. The output of X will change very soon after rising clock edge, and before that time, Y needs to be done using the former output from X. SRAM datasheets include two parameters that are essential to success:

- **Clock to output invalid**, customarily abbreviated **tKQX**, is how long a RAM’s previous output will continue to be available after a new clock pulse.

The manufacturer of the RAMs I’ve been using guarantees that tKQX is no less

than 2.0 ns.

- **Hold time**, customarily abbreviated **tH**, is how long a RAM's input must remain stable after a new clock pulse. The manufacturer of the RAMs I've been using guarantees that tKQX is no more than 0.5 ns.

The largest permissible clock skew between RAMs is the difference between these guaranteed parameters,  $tKQX - tH$ , which for the RAMs I've been working with is 1.5 ns. Although we may think of skew as a difference in clock arrival times, skew also includes waveform variations due to noise—and a minicomputer CPU is a high-noise environment—and component-to-component variations in sensitivity to clock transitions.

Keeping clock skew to 1.5 ns is going to be a very restrictive timing requirement. Although this constraint only applies to RAMs that directly connect to each other, the connectedness between RAMs in a CPU is very high. In my architecture, the right register file RAM supplies data to 18 other RAMs, and the control decoder RAM connects to at least 27 other RAMs. So for practical purposes, the 1.5 ns skew limit must be honored between every pair of clock input pins throughout the whole minicomputer.

Friends who have listened to me explain this problem have suggested reducing the clock speed as a temporarily solution, at least until a first prototype is functioning. This does not help, because tKQX and tH result from physical properties of silicon gates that are not affected by clock speed at all. Even if the clock period is 30 days instead of 12.5 ns, the CPU will produce incorrect results once its clock skew appreciably exceeds 1.5 ns.

Clock skew problems worsen when glue logic is added to SRAM CPU designs. As a simple example from the real architecture, consider flip-flop d (hereafter, “ff d”) in figure 1.3 (p. 17) at node 1. Node 1 tells the left and right register files which registers to read and write, information that comes from the instruction word from

the code RAM. The 36-bit instruction has this form:

opcode	dest. register	left register	right register
bits 35–27	bits 26–18	bits 17–9	bits 8–0

During register fetches, the left and right register RAMs read directly from the locations supplied by the code RAM, and simultaneously, two copies of the destination register number are clocked from the code RAM into ff d. At the end of the instruction when its result is written to both registers, the code RAM outputs are disabled, and ff d’s output is turned on, sending the correct destination register number to the left and right register RAMs.

The problem in this example is clocking the correct destination into ff d. The flip-flop’s hold time is 0.4 ns, and the code RAM’s tKQX is still 2.0 ns, so the clock skew mustn’t exceed 1.6 ns. And for both register files and ff d, the action to take on the clock pulse is *conditional*. We don’t *always* read from or write to the registers, and we don’t *always* write a new destination register number into ff d. These things must happen on exactly the right clock pulses, never on the wrong clock pulses, according to control signals supplied to the RAMs and ff d.

In the case of synchronous SRAM such as the register files, it’s permissible to connect the CPU’s master clock directly to the SRAM clock inputs. This is because the SRAM clock inputs are true clock inputs: they control timing, but they do not control functionality. Other control signals to the SRAMs determine what happens when the clock goes high: a write, a read, a deselect, etc. There is even a control pin that tells the RAM to ignore the next rising clock. But not so with ff d, because on a flip-flop, the input we erroneously call a *clock* is actually a *strobe*. This strobe doesn’t just determine timing; it also determines functionality. When the strobe goes high, the data at the flip-flop’s inputs are unconditionally captured, which is the opposite of what the CPU needs. Instead, the inputs must be captured when the control unit says to capture them, and at no other time. So the CPU’s master clock may not be

connected ff d's clock pins.

It may be tempting to add logic between the CPU master clock and ff d's clock pins, so the flip-flop can know when to capture the destination register number. The problem with this is that any glue logic we insert is unclocked and has variable delay that is affected by temperature and other conditions. We may choose, for instance, to use an AND gate to modulate the system clock coming to ff d under the direction of the control unit. But we can't say how long the AND gate will take. The AUC logic family's faster AND gate has propagation delay that is guaranteed to be at least 0.5 ns, but no more than 1.0 ns. We would have to design this 0.5 ns uncertainty into the circuit board somehow, which will erode our already-worrisome skew budget of 1.6 ns down to 1.1 ns. A further concern is, the AND gate output may not be guaranteed to transition smoothly, and the datasheet makes no guarantee of that kind. All we know is that an AND gate won't change its output for 0.5 ns, but will have the correct output after 1.0 ns. Theoretically—I know this from designing AND gate cells in my Wright State coursework—there could be spurious output between 0.5 ns and 1.0 ns after any transition of the CPU master clock.

What I have done for the interim is, I have conditioned ff d's clock pins not using an AND gate, but with a one-bit flip-flop named “ff dflop,” the flip-flop that controls ff d. The CPU master clock drives the clock pin on ff dflop, and the control unit feeds its data pin. The added flip-flop may or may not have less output noise than an AND gate; actual fabrication and testing will be necessary to confirm the design is suitable. As with the AND gate, ff dflop erodes the CPU's clock skew budget, so this remains a problem to work through.

When it comes time to fabricate a first physical model of my architecture, I estimate that the clock skew problems mentioned in this section are the greatest single threat to correct operation in early attempts. In section 8.6.1, I present three ideas for managing this risk.

## 3.5 Derived components

Most logic ICs that I knew of during high school have yielded to VLSI and never made it into the AUC logic family. Because they cannot be used now, below are AUC derivations of functions that are useful to the architecture of this dissertation.

### 3.5.1 Multiplexers

In the SN74AUC series, multiplexers can be assembled from tristate buffers or brute-force glue logic. A one-of-two decoder/demultiplexer also exists, which combined with two AND gates can achieve a one-bit multiplexer.

### 3.5.2 Shift registers

So far as I know, fast shift registers are not available in the form of standard logic ICs. What can be done is use a SN74AUC16374 16-bit D flip-flop from the ALU logic family, feeding its outputs back to the inputs with a positional offset of one. I anticipate this will be useful for the firmware loader as well as the I/O controllers.

### 3.5.3 Counters

So far as I know, fast counters are not available in the form of standard logic ICs. There are two places in my design where I improvised a fast counter from other parts.

Figure 1.3 shows a “return addresses” RAM at the top of the drawing. This is the call stack for the CPU.<sup>5</sup> An up/down counter is needed to adjust the call depth, which is supplied as some of the address bits to this RAM.<sup>6</sup> The up/down sequence needs to be deterministic and non-repeated, but does not need to be in numeric order. I designed this using the two 8-bit halves of an SN74AUC16374 flip-flop as two 8-bit shift registers, with one half implementing “successor,” and the other implementing

---

<sup>5</sup>The stack frame of this architecture contains only return addresses, never data.

<sup>6</sup>The remaining address bits indicate which in-memory program is being executed.

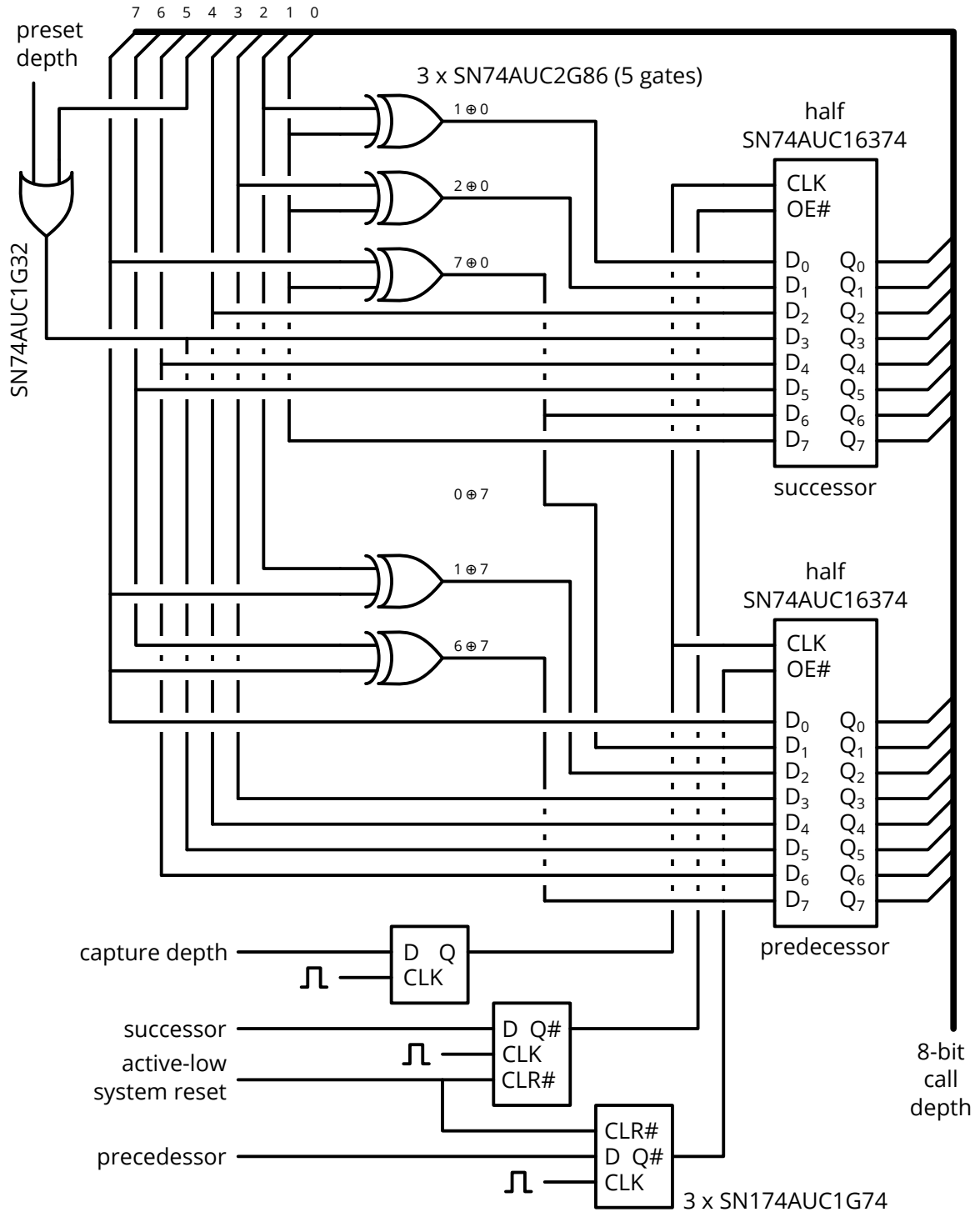


“predecessor.” This is shown as figure 3.2. The sequence generated is a three-tap Galois linear feedback shift register (LFSR). Five XOR gates—three for successor, and two for predecessor—provide the necessary LFSR taps. An OR gate into one of the successor flip-flop inputs ensures that the LFSR can be initialized to a nonzero state. Without this initialization, the LFSR could collapse to generating all zeros, making subroutine calls impossible.

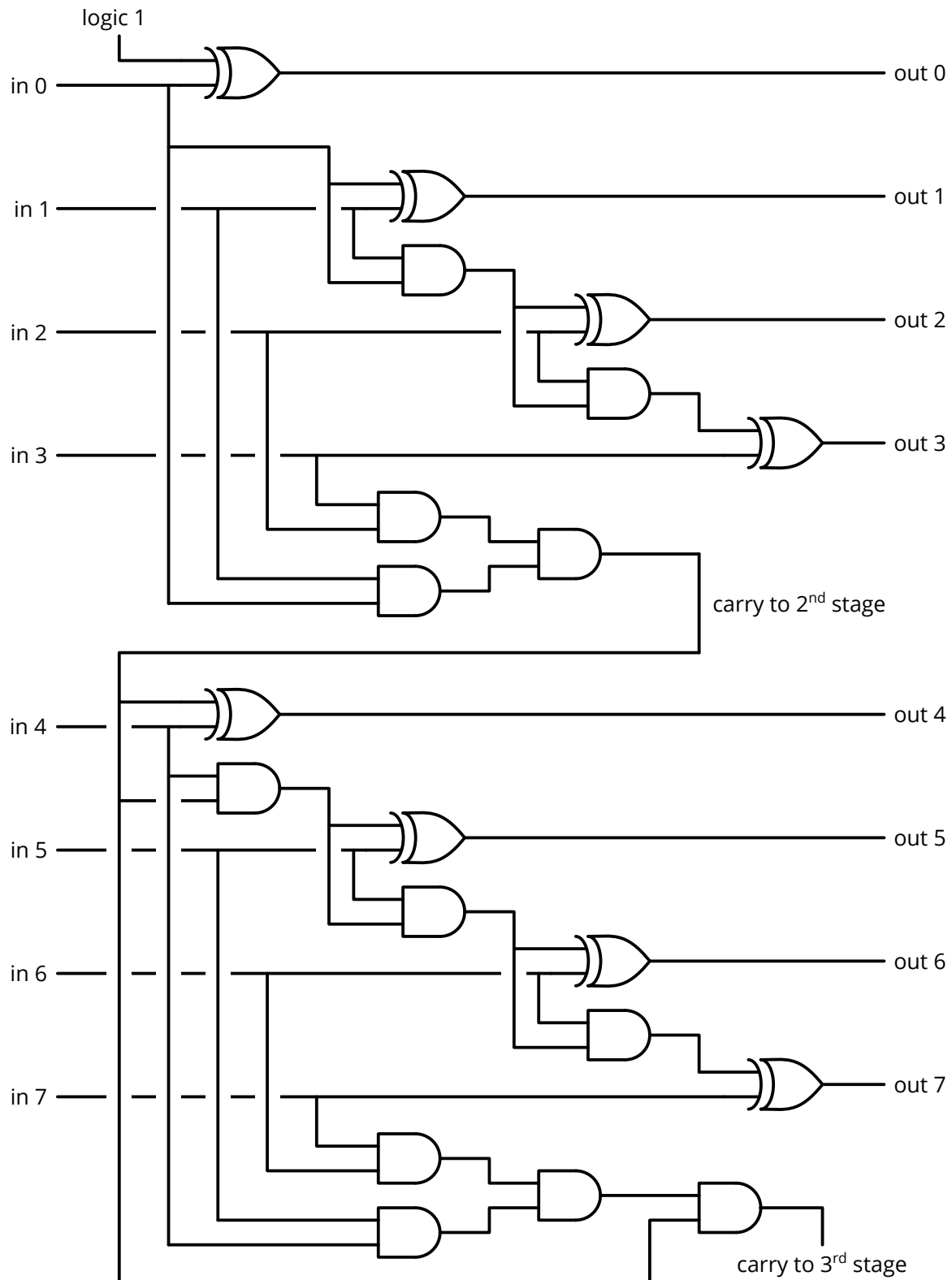
The benefit to using an LFSR in lieu of a counter is that LFSRs do not need to propagate any carries. Only one gate delay, involving a small number XOR gates wired in parallel, is required to transition from one state to the next, regardless of word size. The circuit of figure 3.2 is able to cycle forward or backward through  $2^8 - 1 = 255$  states, allowing any program’s stack to be up to 254 calls deep. I believe this is sufficient, because the architecture specifically disallows recursion via the stack in order to guarantee return address integrity. Section 8.4.3 has more information regarding the stack for my architecture.

The other fast counter in the architecture is the instruction pointer (IP). Before I found the AUC family for glue logic, I was certain the IP would necessarily be an LFSR. It would have been a strange arrangement, because instructions that execute consecutively would not have been consecutive in memory. I believed also that to simplify memory management, a paged model would be necessary where an LFSR only controlled the lower 10, 11, or 12 bits (I hadn’t chosen exactly how many yet) of an address, and a branch instruction would be required to move from one page to the next. I planned to write the assembler to automatically insert these page-to-page branches, as well as a mechanism for the programmer to protect tight loops from containing page changes.

There is precedent for using a paged LFSR instruction pointer. The TMS1000 “one-chip microcomputer” [TI75], which was used in some popular toys of the late 1970s, used a 6-bit LFSR and 4-bit page register together as a program counter for its  $1024 \times 8$  code ROM. It was hard enough in 1975 to make 4-bit CPUs at all, so this



**Figure 3.2:** A 16-bit D flip-flop and some XOR gates provide a fast “up/down counter” (actually a successor/predecessor linear feedback shift register) for call stack depth. The control signal introduction mechanism at the bottom of the drawing is no longer current, but shows the essential functioning of these signals.



**Figure 3.3:** The first two 4-bit stages of the instruction pointer incrementer. Fast-enough counter ICs are not sold, so SN74AUC-series glue logic is chosen instead.

was a good approach for that day. RAM was so expensive that Texas Instruments hacked the LFSR sequence to generate all 64 elements instead of the 63 that naturally occurred.<sup>7</sup> But for my architecture, a way to use a linear counter appeared.

Figure 3.3 shows the first 8 bits of the 27-bit instruction pointer incrementer. All this circuit does is add one to a 27-bit binary number near the top of figure 1.3 (p. 17) at the “add one” label. The circuit is a carry-skip adder with a single addend; a hardwired carry input of logic 1 sets off the increment task. Each stage of the incrementer tackles four input bits, with a shortcut carry-propagate chain extended by one AND gate per stage. Although the drawing shows an unnecessary XOR gate computing the “out 0” bit, it’s there only to aid the reader’s understanding of the carry chain. The actual netlist uses an inverter. Place values  $2^8$  through  $2^{26}$  are five repetitions of the same 4-bit stage as for place values  $2^4$  through  $2^7$ , so the remaining three pages of the schematic have been omitted. The final stage is truncated to three bits and does not require a carry output, so the total bill of materials is 41 AND gates, 26 XOR gates, and an inverter. No path through the incrementer contains more than one XOR, and the longest path has nine AND gates and one XOR, for a maximum propagation delay of 10.3 ns plus wiring overhead. Although that does not fit safely into my 12.5 ns CPU clock period target, increments only occur every fourth clock cycle. Even with ff f and ff t also in the path, the incrementer is sufficiently fast.

## 3.6 Non-computing component security

### 3.6.1 Firmware reservoir

The emergent architecture has approximately 100 Mibit of static RAMs that will need firmware moved into them at power-up. Because there are times when the power is off, trustworthy non-volatile storage will be needed for this firmware.

---

<sup>7</sup>If the current 6-bit word was `abcdef`, let `g = a XOR b XOR AND5(b, c, d, e, f)`. The next 6-bit word was `bcdefg`. I found this by reverse-engineering assembler listings from the manual.

Although an offline, traditional computer could be used to bootstrap a solder-defined minicomputer, a causality dilemma ensues as to whether the traditional machine’s supply chain is sufficiently free of exploitable defects. I prefer that instead, a solder-defined minicomputer have its own solder-defined firmware loader, in which case the remaining question is what kind of non-volatile component will hold its firmware.

If freedom from remote exploits were the only criterion, probably nothing can beat punched Mylar tape for holding firmware. If the tape is dependable enough to not require error correction, no synchronization overhead is needed, and the historic pitch of 2.54 mm (0.1 in) per byte is retained, 33 km (21 mi) would suffice to boot the machine. This is within human ability to build, but is not practical for use.

After passing on Mylar tape and a few other technologies, I settled on having a single NOR flash IC to hold the entire firmware. Part of my reasoning is, most contemporary computers have many components that retain state when power is not applied [Rutkowska15b], providing many enclaves for malicious code to lodge in. The presence, nature, extent, meaning, and alteration of the state these components hold are seldom disclosed to computer buyers, much to the joy of their adversaries. As SRAM minicomputers require firmware by their nature, what I have sought to do is centralize all firmware into one central, generic, well-documented part.

As a starting point, I bought ISSI’s IS25LP128F-JBLE to try out,<sup>8</sup> which as of September 2022 is available new for less than \$3 in unit quantities. This is a 128 Mibit serial NOR flash with an 8-pin package. Its claimed data retention is more than 20 years. As with comparable products from a few manufacturers, this chip can be configured to serially read the entire device as soon as power and a clock are applied. This relieves the minicomputer of needing logic to provide read instructions to the NOR flash chip, increment any addresses, etc. A clocked stream of bits containing the firmware simply appears after power-up, and all that is needed is to direct that

---

<sup>8</sup>Not an endorsement. In fact, I haven’t tried it yet.

stream into the appropriate components. My idea as to how I will accomplish that is in section 9.2.

### 3.6.2 Oscillators and clock buffers

Solder-defined minicomputers should use passive components such as crystals as clock sources. This is no longer a trend for other electronics, so care needs to be taken in terms of what is used. The problem is that the oscillator market has been taken over by programmable PLL ICs, which are usually stabilized to a fixed internal micro-electromechanical system (MEMS) oscillator. Because the PLLs can be programmed long after production by either the supplier or sometimes the consumer, it is unnecessary to specify the output frequency at the time of manufacturing. This allows one part number to cover a very wide frequency output range, instead of the costly historic practice of producing and stocking a different part number for each frequency that will be needed. These ICs are also available with multiple outputs that can be independently programmed; these parts are usually called *clock generators*.

The problem with these configurable oscillators is, they contain *complex logic*. They therefore may contain unseen exploitable defects and particularly backdoors. Although these devices are assumed to produce a never-ending stream of clock pulses, and they do not *appear* to allow input from an adversary, one particular supply chain backdoor would be very easy to embed. The underlying MEMS resonator's frequency is fixed, and the device has nonvolatile storage available for configuration information. This means that the PLL controller can be designed to keep track of the total amount of time it has been powered up. After a period of time selected by an adversary, the device can maliciously degrade its clock output until the surrounding electronics fail to operate correctly. This degradation can be gradual or sudden, deterministic or pseudorandom, soft (resettable by turning the power off) or hard (permanent). It would be very difficult, to the point of being improbable, for a victim of a VCO oscillator attack to determine what happened.

Even if a buyer suspected that an oscillator may be programmed to fail, there is no way to test that assertion other than to power it up until problems appear. Such a test could require years, because for some threat scenarios the attacker would be willing to wait. A nuclear weapon made tomorrow could no longer be an asset in five years. Or a turbo generator could reverse phase after two years of service.

A frightening aspect of the VCO oscillator problem is that this attack can be targeted to specific locations, owners, and applications. Not only can a backdoor be enabled on a per-shipment basis, or by shipment interception, or by a saboteur at the eventual installation, but it could also be tailored to only appear if programmed for a frequency needed by a specific device, such as an FGM-148 Javelin. The number of time power is applied, or the distribution of the durations when power is applied, could also be used to arm the attack.

Even if programmable VCO oscillators are not subject to malicious influence, I am concerned as to their longevity. What is the retention time of this programming? *Is there an acceptable retention time?* Deployed technology has a way of hanging around longer than planned. Voyager 1 is about to celebrate its 42nd year since the end of its planned mission, and the U.S. Air Force only stopped using 8-inch floppy disks in 2019.

Another component related to programmable VCO oscillators is a specific class of clock buffers, namely the zero-delay buffer. These devices provide amplification for clock trees without any propagation delay, and sometimes also offer synchronization input from a point elsewhere in the circuit. The absence of propagation delay allows tighter clock tolerances to be achieved, overcoming problems that passive buffers have with part-to-part and temperature-dependent delay variations.

The problem with zero-delay buffers is, the delay is not zero but an undetectable multiple of the clock period, as covered up by a digital controller and VCO. With the addition of a little persistent storage (if not already present), these gadgets can be manufactured with a backdoor that, after allowing time for deployment, can effect

clock failures in critical infrastructure electronics.

To limit concerns about exploitable oscillator vulnerabilities, my design uses an 80 MHz crystal oscillator. It set me back \$1.24.

### 3.6.3 Peripherals

Realistic computers always connect to *something*, and it would often be the case that solder-defined computers connect to equipment that is not solder-defined. This means there will be interconnections between an architecture like mine, which may be protected to an extent against threats in the supply chain, and architectures that a buyer will have no control over. In the near term, the best that can be done may be to ensure that exploitable defects cannot cross these interconnections.

Section 9.3 outlines how I would provide a minimalist I/O subsystem that confines every peripheral its own I/O bus and buffer memory. It also describes the controllers to manage these buses and buffers, so that the CPU is free for its own work. In order to minimize components and wiring, bit-serial buses are used with clocking supplied by the I/O controllers at their convenience, instead of being clocked by the peripherals or at specific frequencies.

When I looked at specific types of peripherals, such as real-time clocks, GPS receivers, and mass storage, I found that two serial bus specifications cover a very large range of available components. These are Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I<sup>2</sup>C). Components and peripherals that support neither of these usually support another interface, such as a UART or GPIO pins, for which an SPI or I<sup>2</sup>C adapter is readily available.

I recommend ordinarily providing both SPI and I<sup>2</sup>C. Although the two buses have different relative strengths, much of the same digital circuitry can control either. The SPI bus is generally faster, due in part to push-pull drivers. On the other hand, the popular DS3231 temperature-compensated real-time clock (RTC) by Maxim Integrated only supports I<sup>2</sup>C, and I am not aware of an alternate part with equivalent



accuracy and cost.<sup>9</sup>

Both I<sup>2</sup>C and SPI were intended to support bus sharing by peripherals. To do so could permit, at a minimum, peripherals to eavesdrop on each other as well as inject harmful noise on the line. Considering that I<sup>2</sup>C and SPI are two- and four-wire buses respectively, it's my position that the tiny hardware investment needed to isolate each peripheral on its own bus is mandatory for all non-discounted peripherals.<sup>10</sup>

### 3.6.4 Capacitors

Because ISO 27000 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) on at least the first versions of my circuit boards. I am less concerned about electrolytic capacitors in commodity power supplies that can be replaced easily, subject to their use's sensitivity to outages and servicing.

---

<sup>9</sup>By convention, nearly all RTCs come with two regrettable defects that add complexity and bloat to driver code. First, they count time in years, months, days, hours, minutes, and seconds instead of a single duration elapsed from some epoch, like sane operating systems do. Translation is required to set an RTC, and translation with validation is required to read it. Second, although most RTCs count time at 32768 Hz, they can only be read or written in whole seconds. This requires a driver to loop around a seconds transition in order to use the clock near its internal precision.

<sup>10</sup>Discounted logic is defined on page 22.

# 4

## Logic blocks for SRAM ALUs

### 4.1 Hierarchy of ALU capabilities

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

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

The next step up for ALUs is hardware multiplication. Typically two words are accepted, and a doubleword product is produced. At this step, signed and unsigned genders of operation are offered, but combinations of signedness are not offered. The foremost use for multiplication is to locate array elements that aren’t a power of two

in size. Serious numeric computing also needs multiplication, but arrays will appear in a much broader range of applications. The distinction between multiplying for arrays and multiplying for numeric processing becomes important, because finding array element locations can usually be done using short multiplication, implying that many machines will do fine with only a short multiplier.

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

The next stage of ALU robustness adds word rearrangement. The ability to whip subwords around outside the linear order of shifts and rotations is combinatorially difficult to implement, because the number of potential rearrangements is larger than the word size can represent. Even rearranging an 8-bit word requires a 24-bit argument if we are to have full selection including bit duplication. If only 8-bit permutations are supported and no bits are duplicated, a 16-bit argument is still needed, and fancy logic or a big table has to be added to decode it. Yet rearrangement within words comes up periodically, often with foreseeable gyrations, and ALUs that provide common ones in constant time are desirable. In the x86 family, the 80386 introduced some sign extend instructions, the 80486 added a byte swap for fast endianness modification, and BMI2 recently brought parallel bit deposit and extract.<sup>1</sup> Word rearrangement is useful for serialization, floating-point routines on

---

<sup>1</sup>Bit Manipulation Instruction Set 2 arrived with Haswell in 2013.

integer-only CPUs, swizzling for graphics, evaluating chess positions, hash functions, pseudorandom number generators, and cryptography. Where practical, ALUs should aid these operations rather than stand in their way.

Another family of ALU operations to consider is cryptography. Today's computers connect to networks, most of which are shared, and many of which are wiretapped. To be suitable for use, a computer must encrypt and decrypt fast enough to keep up with its own network traffic. Even if a computer will never be attached to a network, it's still likely to need unbiased pseudorandom number generation, effective hash functions, and protection (by encryption, information splitting using a random variable, or otherwise) from storage devices manufactured overseas.

With the above wish list for ALU features, here are some SRAM building blocks to deliver the goods. All exist in the literature using conventional logic families, but I have not found any literature to suggest implementation using RAM, ROM, or any kind of lookup table. Perhaps no one wants credit for giving away three orders of magnitude for execution speed, let alone an even larger spike in transistor count. But there are applications for which I would be willing to wait nanoseconds rather than picoseconds to assure a CPU's conformance with the expectations on page 7.

## 4.2 Simple lookup elements

*Simple lookup elements* are the fundamental building block of all SRAM logic. Remember that our decision to use RAM has nothing to do with its mutability: we actually want ROM, but RAM is much faster and does not require custom masks. The overall constraining parameter is the number of input bits, which is the base two logarithm of the number of rows. We need enough inputs bits to select among all operations the RAM supports, plus these operations' inputs which are generally one or two subwords and sometimes a carry bit. Figure 4.1 shows an example with two logical and two arithmetic functions. Figure 4.2 show a another problem type that

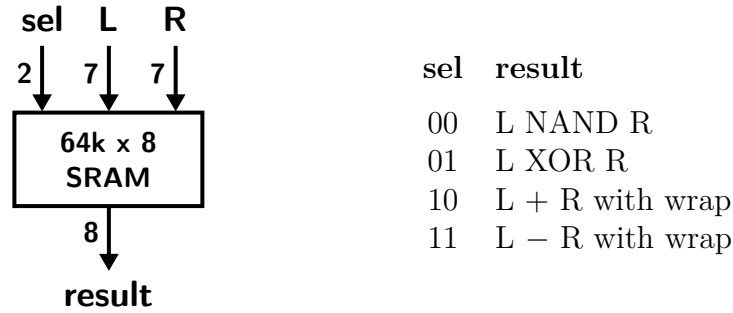


Figure 4.1: Simple lookup element.

$$\begin{array}{r}
 101 \\
 0100 \\
 001 \\
 110 \\
 + 101 \\
 \hline
 011010
 \end{array}
 \qquad
 \begin{array}{r}
 01001 \\
 110 \ 00 \\
 + 0 \ 01010 \\
 \hline
 010001110
 \end{array}$$

Figure 4.2: Arbitrary-geometry addition.

can be solved using simple lookups.

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

### 4.3 Arbitrary geometry adders

*Arbitrary geometry adders* are a special case of simple lookup elements. Figure 4.2 shows two addition problems with more than two addends and irregular column spacing. In the right example, there's either a bit missing within an addend or a line with more than one addend, depending on interpretation. Both examples share a non-obvious feature of exactly 16 input bits, because each is a one-RAM subtask from the binary multiplication demonstration in section 4.9. There is nothing tricky, just

tedious, about how these two RAMs are programmed: some kind person computes all  $2^{16}$  possibilities using the place values shown.

Simple RAM lookup elements are all over the computing literature and especially prevalent in PLDs and FPGAs. The case of arbitrary geometry adders using SRAM isn't in widespread use or discussion known to me, but relates to and conceptually derives from the full- and half-adders used in Wallace [Wallace64] and Dadda [Dadda65] multiplier circuits since the mid-1960s.

An independent introduction to arbitrary geometry adders in the context of fast SRAM multipliers appears in section 10.4.1.

## 4.4 Carry-skip adders

*Carry-skip adders* combine subwords by augmenting not-always-certain carry outputs with a *propagate* output that is set whenever the correct carry output is indicated by the carry output from the immediate right subword. Figure 4.3 shows the carry outputs that will occur when two 3-bit subwords are added with a possible carry input. In practice we add wider subwords, and several drawings to come use 4-bit subwords, but using three bits for this drawing helps keep it legible. The key concept is that at times, the carry output cannot be computed until the carry input is known, and when this occurs, the carry output is always exactly the carry input. So if two machine words are added by means of parallel subword additions, the subword place value results must be determined later than subword carry results. The circuit responsible for this is called a *carry-skip adder*, and a 12-bit example combining 4-bit subword additions appears as figure 4.4.

As figure 4.4 shows, the number of stages to consider increases as we move leftward; this is like a spatial transposition of the time sequence of ripple-carry adders. Traditional carry-skip adders have a time-sequential propagation of carry information from right to left, but they move by subwords and therefore offer a speed benefit over

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	<b>C</b>
1	0	0	0	0	0	0	<b>C</b>	1
2	0	0	0	0	0	<b>C</b>	1	1
3	0	0	0	0	<b>C</b>	1	1	1
4	0	0	0	<b>C</b>	1	1	1	1
5	0	0	<b>C</b>	1	1	1	1	1
6	0	<b>C</b>	1	1	1	1	1	1
7	<b>C</b>	1	1	1	1	1	1	1

Figure 4.3: Subword carry decisions for 3-bit addition.

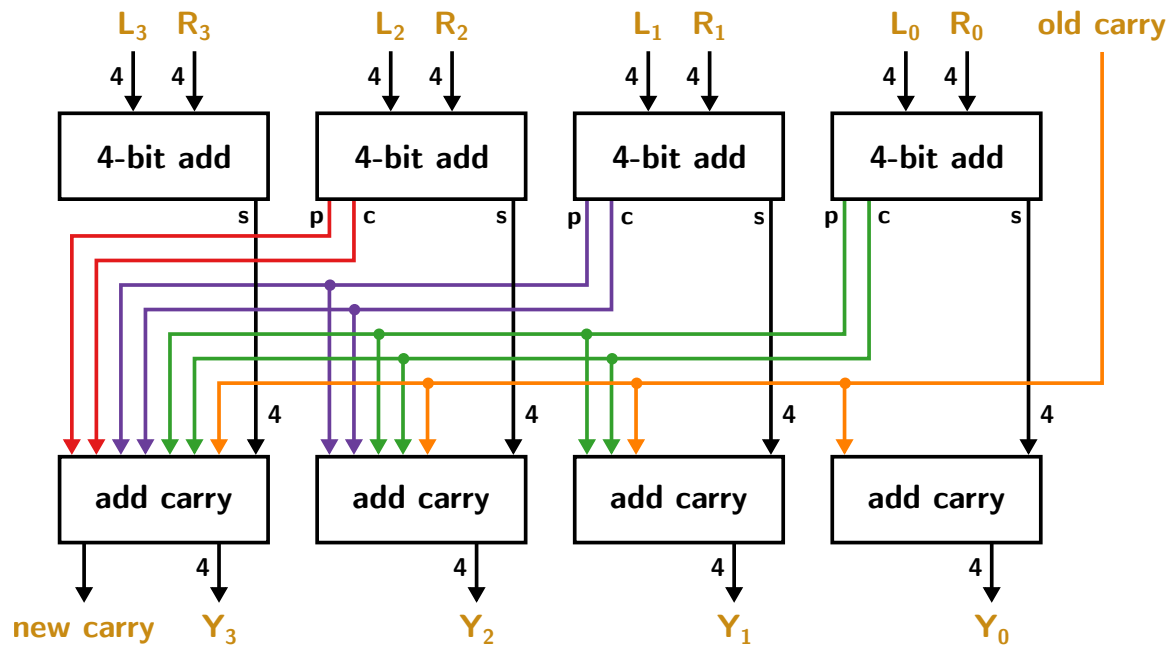


Figure 4.4: Two-layer carry-skip adder.

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

SRAM carry-skip adders can be categorized into designs that use either two or three gate delays. A distinguishing characteristic of two-layer adders as pictured in Figs. 4.4, 4.5, and 4.6 is that their carry deciding and final subword incrementing happen simultaneously. They don't determine the carry decision *and then* apply it, nor even compute the needed carry decision at all. They merely apply the correct carry decision as if it had been computed. Here, the magic of table-driven computation removes a stage.

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

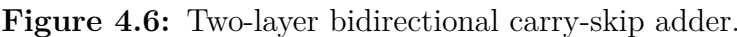
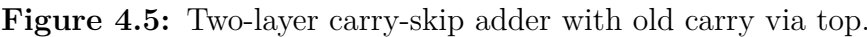
Numerous tradeoffs and alternatives appear in SRAM carry-skip adder designs. Figure 4.5 moves the old carry input to the first layer, saving much address space and a RAM in layer two, but crowding the least significant RAM in the first layer more. This change also reduces potential to reuse this circuit for purposes other than addition—recall that these devices are merely RAMs, and they only become adders because of their contents. In fact, potential uses other than addition sometimes require that carry information propagate not from right to left as for addition, but from left to right.<sup>2</sup>

Figure 4.6 shows a “bidirectional” carry-skip adder that can propagate information in either direction. Note the choice to introduce the old carry at the second layer,

---

<sup>2</sup>Left-to-right examples include NUDGE (section 7.2.6) and FOL (find one left, section 7.4).





and which SRAM the new carry is deemed to appear at depends on the direction of operation. Although bidirectional operation consumes more inputs at layer 2, their use flattens out among the RAMs and is in one respect a simplification. Although this circuit appears to be useful due to its directional flexibility, an ALU that supports the bit reversal permutation can mirror bits with a preliminary instruction, do the intended operation with ordinary right-to-left carries, and if need be mirror back the resulting bits. Considering the scarcity of occasions that a left-to-right carry is needed, using bit reversals instead—provided they are fast—might be a better design for memory-constrained ALUs.

Figure 4.7 offers a remedy for the crowded propagate and carry wiring associated with two-layer carry-skip adders. It isn't the wiring, of course, but the address space, memory, and transistor count for the high-place-value subwords which need conservation. Two ends of a candle are burning in two-layer carry-skip adders. First, the available subword size for computation soon reaches zero in the bottom layer, which the top layer must also mirror. This sets an upper limit on the total word size that can be added. Second, the adders in Figs. 4–6 *only* add, due to the absence of input bits to select other operations. We need to find more address bits.

Figure 4.7 adds a middle layer to our carry-skip adder. Only one SRAM is needed for this layer, because it doesn't touch any of the tentative sums. It simply reckons the propagate and carry subword outputs into *carry decisions* that are passed to layer 3. The last layer of SRAM is much cleaner, with each RAM requiring only its own carry decision alongside its tentative sum. The conserved address bits can be used for other purposes, notably function selection, and perhaps another operand if a use for one emerges.

Switching to a three-layer adder adds 50% to its propagation delay, a cost that requires justification. Here are three points to consider. First, a lot of address space opens up in the third layer, enabling other computation by these RAMs for non-additive instructions. Second, a lot of *time* opens up in the second layer, because

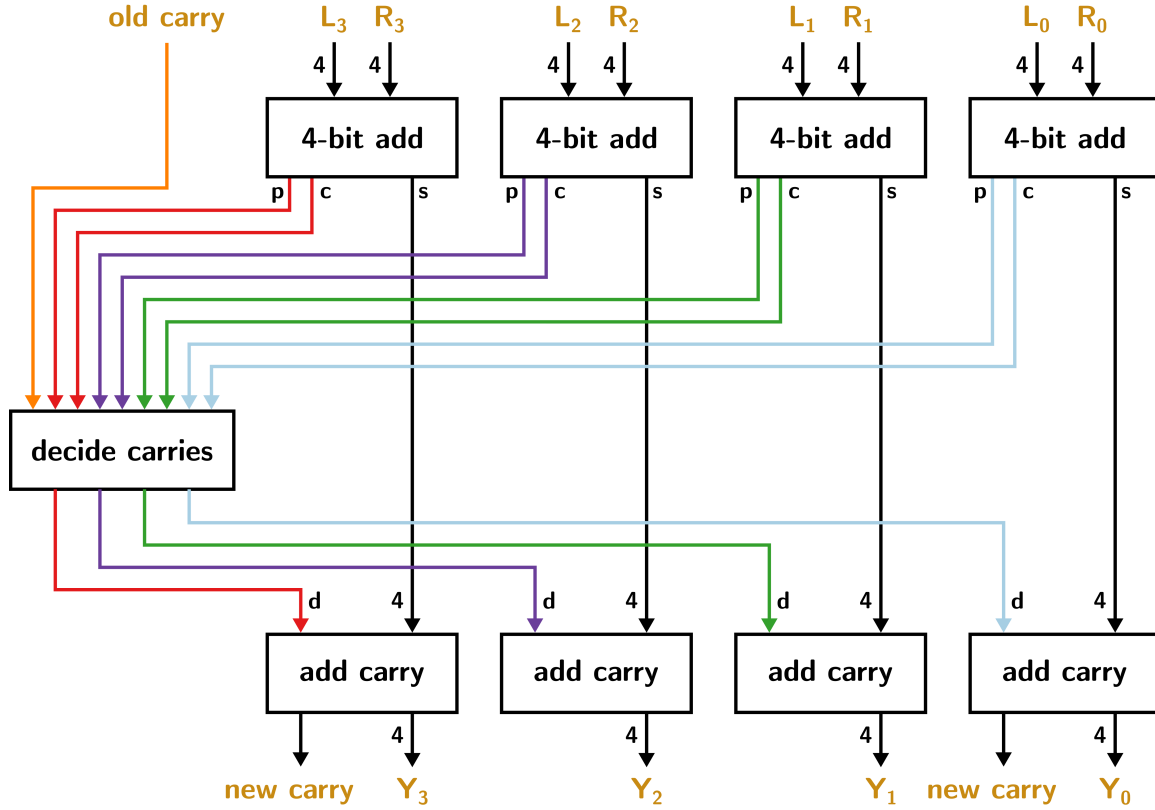


Figure 4.7: Three-layer carry-skip adder.

figure 4.7 has nothing happening to the tentative sums as the carries are being decided. The second layer can be filled in with more RAMs to do more computation with no further performance penalty. Third, the large fan-out in two-layer adders for propagate and carry signals incurs a small speed penalty.

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

An independent introduction to carry-skip adders in the context of fast SRAM multipliers appears in section 10.4.2.

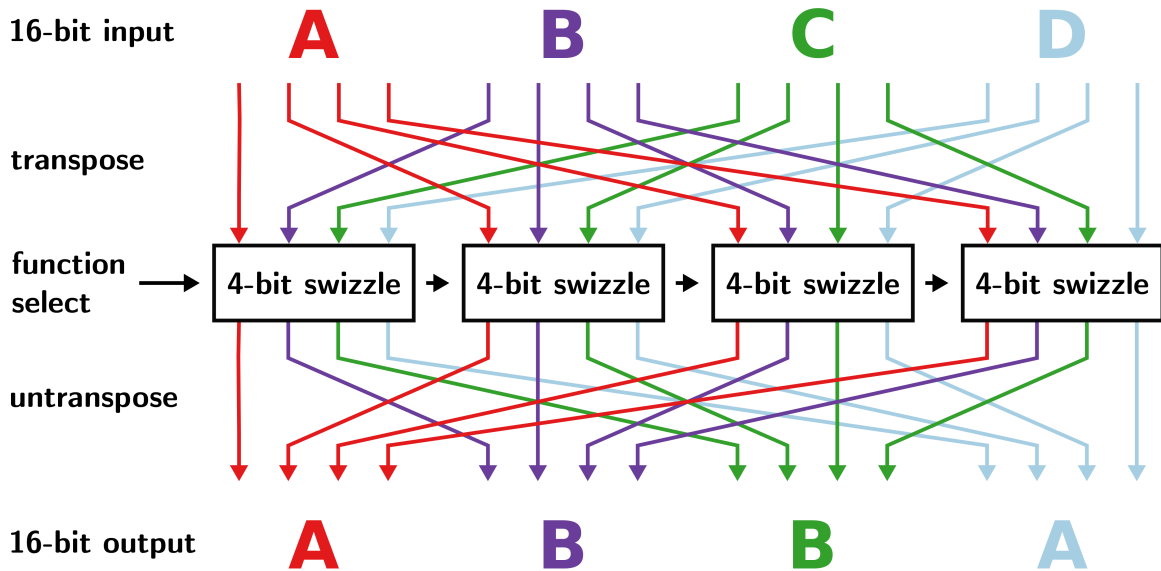


Figure 4.8:  $4 \times 4$  swizzler.

## 4.5 Swizzlers

A *swizzler* is a layer of RAMs that operate on transposed subwords, meaning that each RAM gets one address bit from each subword, looks something up, and outputs one data bit to each subword. Figure 4.8 shows how this transposition is wired for a 16-bit word with 4-bit subwords. From right to left, each of the four RAMs operates solely on 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.

Some magic needs to happen in terms of advance planning, because the number of functions these RAMs can hold (and therefore execute) is a tiny sliver of the number of functions possible. Additionally, we need to consider that the RAMs might require unequal contents to achieve certain objectives. It's also possible that the function select bits applied might not be the same across all of the RAMs. It's even possible

that there isn't an input bit that corresponds directly to each output bit: some outputs might be fixed ones, zeros, or some function of multiple inputs. It turns out that all three of these "mights" turn out to be very advantageous, although none are suggested by figure 4.8.

## 4.6 Logarithmic shifters

A *logarithmic shifter* overcomes a key limitation of swizzlers, which are perfect for fast rotation of subwords, but not of bits. If we assign one bit each to letters a–p, we can swizzle `mnop abcd efgh ijkl` to become `abcd efgh ijkl mnop`. But when we try to rotate just one bit position from here instead of four, the result will be `ebcd ifgh mjkl anop`, because place values remain fixed. In fact, only the leftmost RAM would move anything, because the remaining transposed subwords are already correct. To finish our one-bit rotation, we have to clean up the subwords individually to yield `bcde fghi jklm nopa`, which is what we want. This requires a second layer of RAM that can operate within subwords rather than across them. Figure 4.9 shows this combination, which permits rotation of any number of bits in a single pass. Masking is easily added to the values in the RAMs to supply left and right shifts of any number of bits.

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

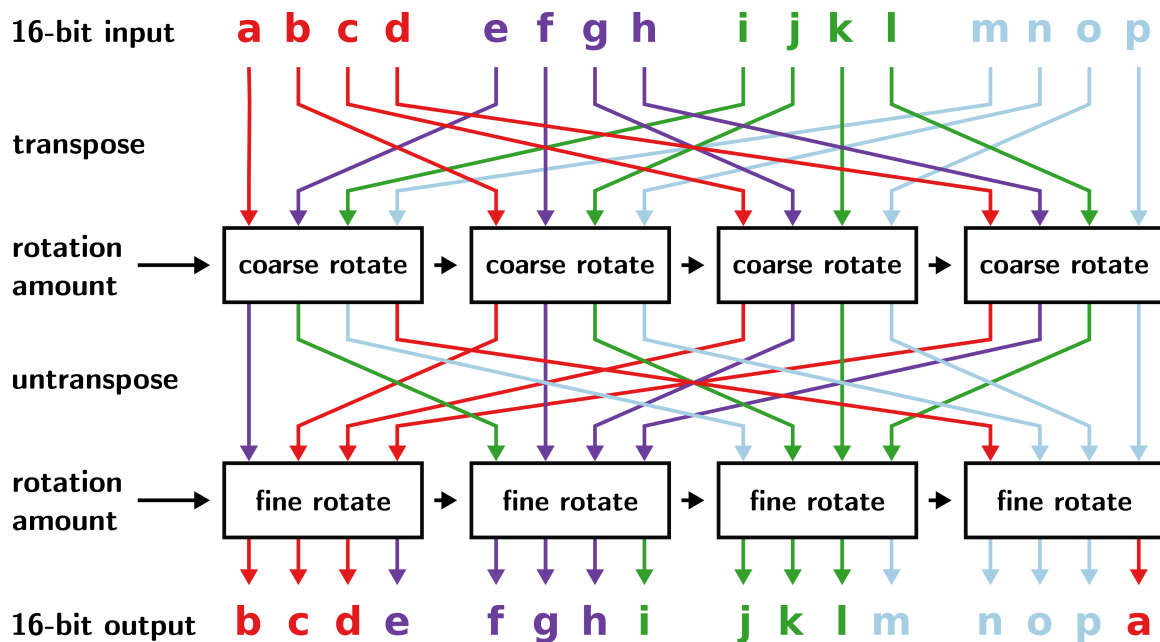
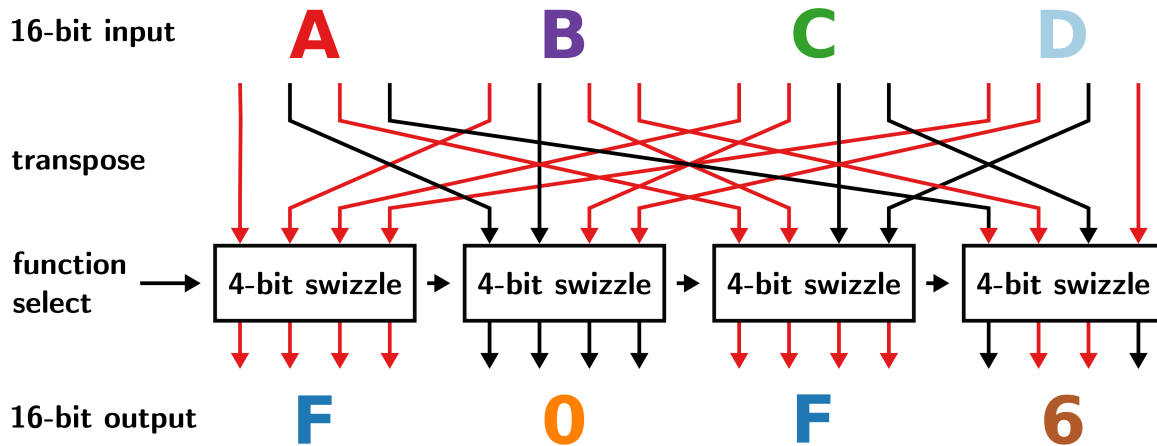


Figure 4.9: 16-bit logarithmic shifter.

## 4.7 Semi-swizzlers

A *semi-swizzler* or *half-shifter* is a contiguous half of a logarithmic shifter; that is, a transposition preceding or following a layer of swizzle RAMs, but no transposition going the other direction. An example appears at figure 4.10, using the same input data and SRAM contents as figure 4.8. The computed output looks like gibberish, because it is still transposed. A semi-swizzler can be applied twice by a program, using two consecutive instructions, to achieve ordinary shifts and rotations. Chapter 6 discusses the mechanism and tradeoffs of superposing a two-layer adder with a semi-swizzler to build a compact ALU.

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



**Figure 4.10:**  $4 \times 4$  semi-swizzler or half-shifter. Two passes through this circuit can emulate a logarithmic shifter.

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

**Figure 4.11:** 4-bit S-box.

## 4.8 Substitution-permutation networks

An *S-box* is an invertible substitution that can operate on subwords. Its purpose is to help progressively alter words in a key-dependent manner, until the alteration sequence is impractical to reverse without knowledge of the key that was used. Figure 4.11 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 un-

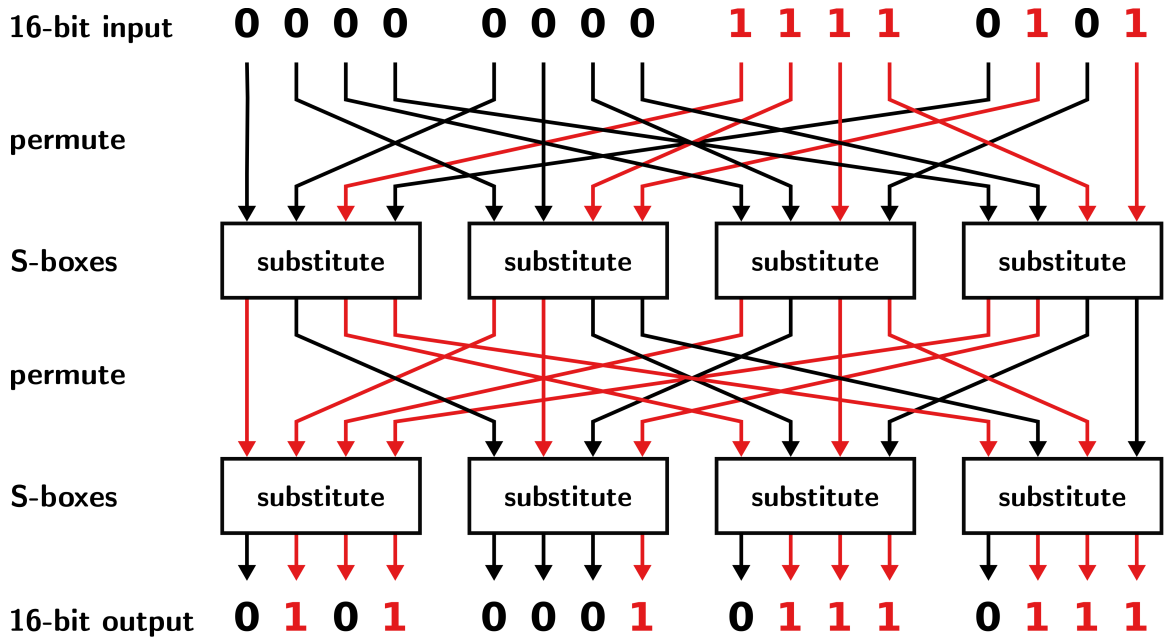


Figure 4.12: 16-bit substitution-permutation network.

scramble 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 4.12 shows a small SPN built using the S-box of figure 4.11.

There is nothing novel about SRAM SPNs, but their mixing capability is very useful for ALUs to incorporate. They are best used under non-adversarial circumstances; e.g., to implement hash functions and pseudorandom number generators (PRNGs). Suitability for cryptography is considered in section 7.2.8.

## 4.9 Fast multipliers

Fast SRAM multipliers are nearly as fast as adders in practice; however, the number of RAMs needed grows a little faster than the square of the word size. The process itself is primarily one of summing partial products quickly. As schematics of multipliers are



tedious to draw and not particularly legible, here is a walkthrough of the process for 32-bit unsigned factors. The 64-bit product will be ready after only five gate delays.

## Layer 1

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

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

```

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

```

## Layer 2

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

$$\begin{array}{r}
\begin{array}{cccccccc}
& & & & 00010110 & 01000100 & & \\
& & & 11011100 & 11100000 & 00000011 & 10000100 & \\
& & 01000000 & 00100000 & 00100010 & 11100000 & 00001010 & 10001100 \\
11100000 & 01110000 & 00001010 & 00100000 & 01101000 & 10100000 & 00010000 & 01001111 \\
& 00100011 & 01110000 & 00011110 & 01100000 & 10100001 & 11001000 & \\
& & 01101010 & 01010000 & 00101110 & 11111000 & & \\
+ & & & & 10100100 & 01100100 & & \\
\hline
& & 00100 & 00 & 001 & 101 & 0010 & 11 & 01001 & & 0100 & 010 \\
11100000 & & 010000 & 001000 & 000110 & 0001010 & 0100100 & 01001111 & & & & \\
& 00 & 110100 & 0 & 11010 & 0 & 10011 & 01 & 1010 & 010 & 1011 & 
\end{array}
\end{array}$$

In this picture, the bits above the line are an exact repetition of the previous layer's output, but their colors now indicate which RAMs they are grouped into for addition. The rightmost place value of each color group can be observed to be the same before and after this operation. Each band can group as many as 16 input bits. The process used is the arbitrary geometry addition from section 4.3, and is a simple table lookup. The black digits on the left and right are not being added, but are passed forward for later addition (left) or for direct use in the final product (right). The reduction in bits is considerably faster than one finds in Wallace [Wallace64] or Dadda [Dadda65] multipliers, because the SRAMs offer fuller capability than full- and half-adders. As this step starts, 8 bits are finished and 248 remain to add. Afterward, 13 bits are finished and only 99 remain to add. 15 RAMs are used.

### Layer 3

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

$$\begin{array}{r}
\begin{array}{cccccccc}
& & & & 00100 & 00 & 001 & 101 & 0010 & 11 & 01001 & & 0100 & 010 \\
11100000 & & & & 010000 & 001000 & 000110 & 0001010 & 0100100 & 01001111 & & & & \\
+ & & 00 & 110100 & 0 & 11010 & 0 & 10011 & 01 & 1010 & 010 & 1011 & & \\
\hline
1110 & & 00 & 1110000 & & 00100 & 011 & & 1001000 & 01100100 & 01001111 & & & \\
& 00000 & 1101010 & & 010 & 01001 & & 0100011 & 10 & & & & & 
\end{array}
\end{array}$$

## Layer 4

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

$$\begin{array}{r}
 \begin{array}{cccccccccccc}
 1110 & & 00 & 1110000 & & 00100 & 011 & & 1001000 & 01100100 & 01001111 \\
 + & 00000 & 1101010 & & 010 & 01001 & & 0100011 & 10 & & 
 \end{array} \\
 \hline
 \begin{array}{cccccccccccc}
 1110000 & 11010100 & 1 & & & 0100 & 10000011 & 11001000 & 01100100 & 01001111 \\
 000 & & 0 & 01100010 & 01001 & & & & & & 
 \end{array}
 \end{array}$$

## Layer 5

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

$$\begin{array}{r}
 \begin{array}{cccccccccccc}
 & & 0 & & & & & & & & & \\
 & 00 & & & 0 & & & & & & & \\
 11100 & & 0 & 0 & & & 0 & & & & & 
 \end{array} \\
 + \begin{array}{cccccccccccc}
 00000 & 11010100 & 11100010 & 01001100 & 10000011 & 11001000 & 01100100 & 01001111 \\
 \hline
 11100000 & 11010100 & 11100010 & 01001100 & 10000011 & 11001000 & 01100100 & 01001111
 \end{array}
 \end{array}$$

To multiply signed numbers, the sign bit(s) of any signed factor(s) has a negative place value; e.g., bit position 7 of an 8-bit signed number has place value  $-128$ . The partial products and addition reductions thereafter will consume and generate a smattering of bits with negative place values. This won't cause the SRAM logic elements any hardship: the only change will be to assure enough output bits for each

RAM, group inputs in the correct address widths, and precompute its addition table correctly.

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

The division of the original factors into subwords does not need to be symmetric, as long as all product and sum place values are grouped correctly. For example, an any-signedness multiplier for 32 bits using  $64\text{Ki} \times 16$  SRAMs does not need to spill from 16 partial products into 25 in order to fit the signedness bits. One factor can have four subwords of [7 bits + signedness, 8 bits, 8 bits, 9 bits], and the other five subwords of [5 bits + signedness, 6 bits, 7 bits, 7 bits, 7 bits], for 20 partial products calculated by 20 SRAMs.<sup>3</sup> Another configuration would be to maintain 16 devices in layer 1 by enlarging some of the RAMs.

Chapter 10 develops a more general theory of fast SRAM multipliers and introduces an open-source tool to automate and optimize their design.

## 4.10 Open question: medium-speed multipliers

The fast multipliers in section 4.9 and chapter 10 add significantly to component count, board space, and cost. But without such added hardware, multiplication takes significantly longer. I looked for compromises that could offer modest speed improvement for modest added hardware. One problem I had in this search was that

---

<sup>3</sup>This configuration could use mainly  $32\text{Ki} \times 16$  and  $16\text{Ki} \times 16$  RAMs, but the former sell for more money than  $64\text{Ki} \times 16$ , and the latter aren’t offered for sale. There is, at least, a slight reliability benefit in leaving much of a  $64\text{Ki} \times 16$  RAM unused, as the opportunity for soft errors is proportionately reduced. There may also be some energy saved, depending on the cell design of the chips selected.

in section 7.2.5, the firmware already does well at adapting long multiplication to the peculiar capabilities of SRAM ALUs. I found it hard to improve on the speed that an ALU and its firmware can achieve on their own.

By partitioning 36-bit words in half, a 36-bit multiplication can be divided into four 18-bit multiplications that could be accelerated via table lookups in a modest auxiliary memory. The method of quarter squares and a few variations are described in [Glaisher1889], [TienLin73], [Johnson80], and [Ling90]. I estimate that the half-word extractions and various subtractions and additions that surround the table lookups would make this “medium-speed” method little if at all faster than the firmware-only acceleration used in listing 7.2. I have not tried to experimentally confirm this estimate by writing medium-speed multiplication code.

Another approach to medium-speed multiplication would be to consider a small hardware addition for multiplying numbers that are not the full word size. With parallel  $128\text{Mi} \times 8$  NOR flash available, it may be possible to do 27-bit multiplication with just two table lookups, plus a handful of instructions before and after. There are downsides to this proposition, such as NOR flash access time that may exceed the duration of a CPU cycle, as well as a large expansion in the number of nonvolatile memory ICs in the machine. I suspect that if a niche use case is identified for a NOR flash approach, its customer would instead prefer a fast SRAM multiplier using the methods of chapter 10.

# 5

## Three-layer ALU structure

This chapter introduces an elegant design for three-layer ALUs in terms of their physical organization. A description of its programmable functionality will wait for chapter 7 where firmware is discussed. Readers who would like a quick look at the capabilities are directed to table 7.1 (p. 102). Timing information for common tasks, including multiplication schemes that don't need more components, appear in table 11.1 (p. 278).

### 5.1 Superpositions of SRAM logic blocks

The SRAM logic blocks of chapter 4 complement each other like pieces in a puzzle, as if there are circuits they are meant to assemble into. Most strikingly, the three-layer carry-skip adder of figure 4.7 (p. 62) shows a void in its second layer while and where carry decisions are made. There is space to insert four more RAMs within the sum datapaths, with each processing four bits. The swizzler of figure 4.8 (p. 63) matches this description exactly. Figure 5.1 shows the superposition of these two circuits with only minor changes:

- The circuit is enlarged from four 4-bit subwords to six 6-bit subwords, and thus contemplates a word size of 36 bits instead of 16.
- To keep figure 5.1 legible, the transpose and untranspose wiring is no longer

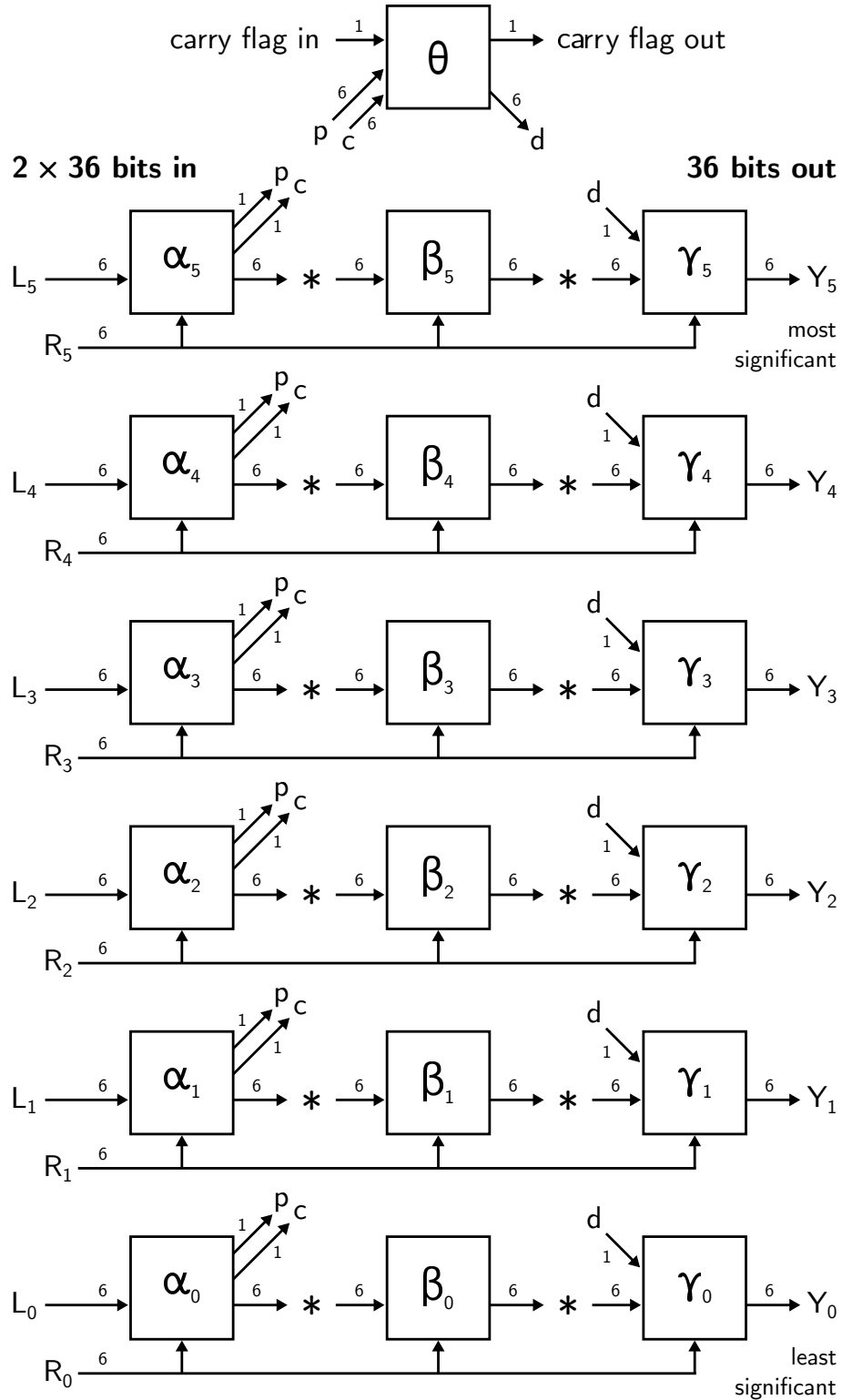
shown, but indicated using  $\star$ s.

- Because no wires crossings need to be drawn, color has been omitted.
- To keep the drawing on the page without using landscape, the datapath is drawn from left to right instead of top to bottom.
- The RAM operations are indicated by Greek letters instead of text.

The reason for the Greek letters is one of generalization: figure 5.1 is neither an adder nor a swizzler, but is something more akin to an FPGA. Any RAM IC in the circuit can do whatever it's programmed to, or do nothing and pass its input along unchanged or even zeroed. So why these particular symbols?

- $\alpha$ ,  $\beta$ , and  $\gamma$ , being the first, second, and third letters (alpha, beta, and gamma) of the Greek alphabet, indicate the first, second, and third layers of the circuit. They don't indicate *what* they do, because the RAMs' contents can change that. Instead, they indicate *where* the RAMs are, because the RAMs aren't capable of unsoldering themselves and moving to other places.
- $\theta$  (theta) was chosen for its canonical use in trigonometry instruction, because relative to the other RAMs drawn,  $\theta$  is “off at an angle.”
- One more RAM denoted  $\zeta$  (zeta) will come into the discussion later for handling CPU flags such as Z(ero).

What's transformational about the circuit of figure 5.1 is that its subsets include all of the logic blocks from chapter 4 except for fast multipliers. Figure 5.2 uses gold fill to indicate the RAMs of figure 5.1 that can do 36-bit carry-skip addition, shifts and rotations, swizzles, and S-box operations. The significance of this circuit is that a 36-bit ALU can be assembled using very few soldered components—19 SRAMs—that its operation is very fast (3 gate delays), and that its functionality is very robust.



**Figure 5.1:** Block diagram of a 36-bit ALU. Each square corresponds to one SRAM IC. The two 36-bit transpose operations, shown as  $*$ , are self-inverse. Both connect output  $i$  of tribble  $j$  to input  $j$  of tribble  $i$ . Small digits that are not subscripts indicate number of wires. Figure 5.6 offers a landscape version of this drawing.



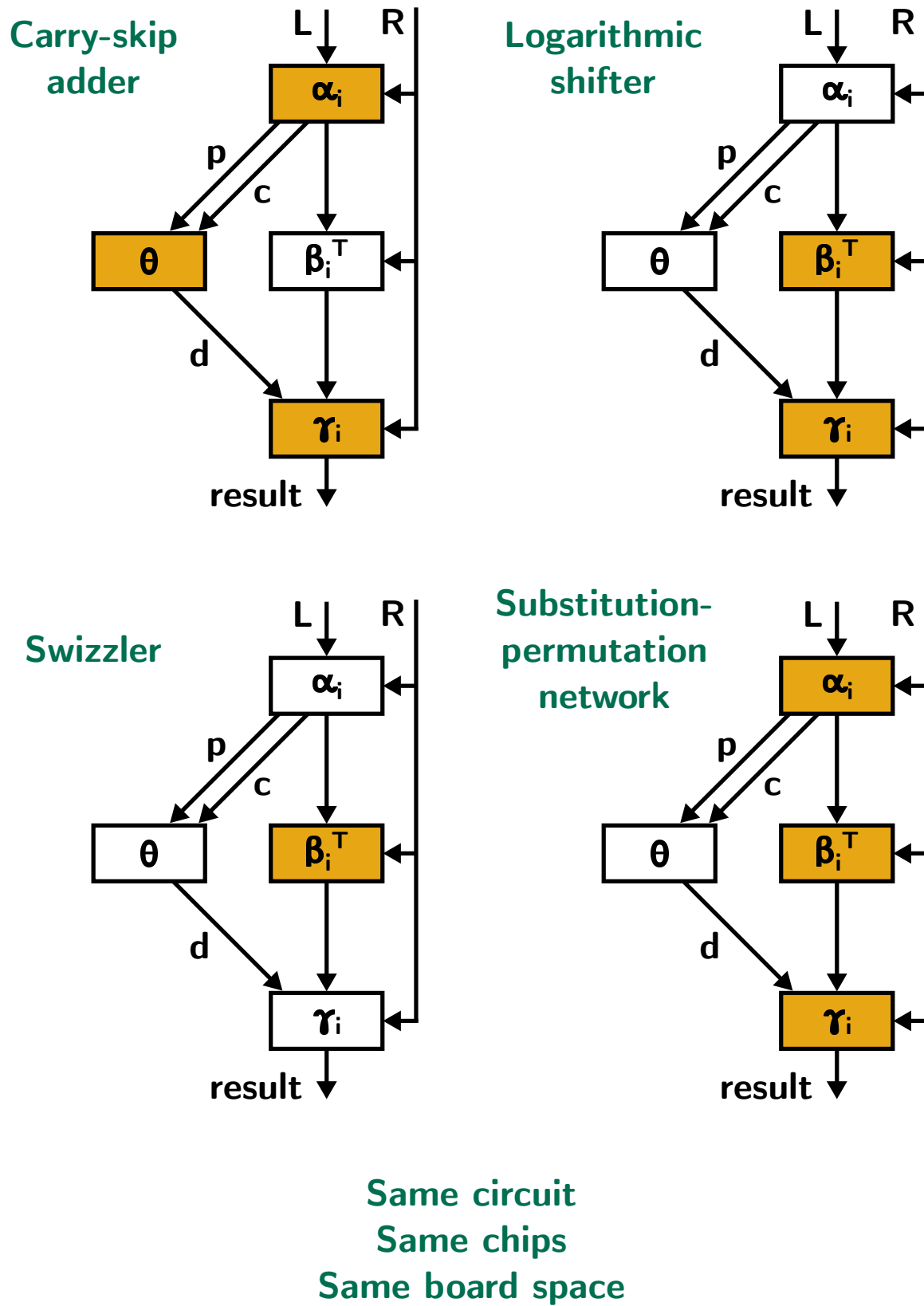


Figure 5.2: Superposition of major components of a three-layer, 36-bit ALU.

$w$						$w^T$					
z	y	x	w	v	u	z	t	n	h	b	5
t	s	r	q	p	o	y	s	m	g	a	4
n	m	l	k	j	i	x	r	l	f	9	3
h	g	f	e	d	c	w	q	k	e	8	2
b	a	9	8	7	6	v	p	j	d	7	1
5	4	3	2	1	0	u	o	i	c	6	0

**Figure 5.3:** The bit positions of 36-bit word  $w$  can be written as a  $6 \times 6$  square matrix using base 36. Transposition is simply reflection through the main diagonal.

The flow of data through the ALU can be thought of as beginning with the left operand  $L$ , which undergoes a transformation at the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers. Each transformation “loses” its previous input, in the sense that only  $\alpha$  receives the left operand, only  $\beta$  receives  $\alpha$ ’s output, etc. In contrast, the right operand  $R$  is available unmodified to all three layers, as figure 5.1 shows. This imbalance between how  $L$  and  $R$  feed through the ALU causes asymmetry with respect to how the operands can be used. For instance, if a word is to be rotated five bits left, the rotation amount (five bits) needs to be available at every RAM that participates in the rotation. Right operand  $R$  can achieve this, because all 18 RAMs of the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers connect directly to it. With the right operand thus spent, the word to be rotated can only be supplied via left operand  $L$ . So the asymmetry in the wiring is comparable to the asymmetry of binary operations such as  $C$ ’s  $\ll$  operator for left shifts. But there are also many symmetric binary operators used for programming, such as  $C$ ’s  $\wedge$  operator for exclusive OR, and the ALU “suppresses” its asymmetry to achieve it. How? By having the  $\alpha$  and  $\beta$  layers pass left operand  $L$  unchanged to  $\gamma$ , which in turn computes  $L \wedge R$ .

In figure 5.2, a superscript  $\top$  appears in the  $\beta$  layer to remind the reader of its transposition relative to  $\alpha$  and  $\gamma$ . One way of looking at the transposition is to think of a 36-bit word as a 36-bit square array instead. The individual RAMs of the  $\alpha$  and

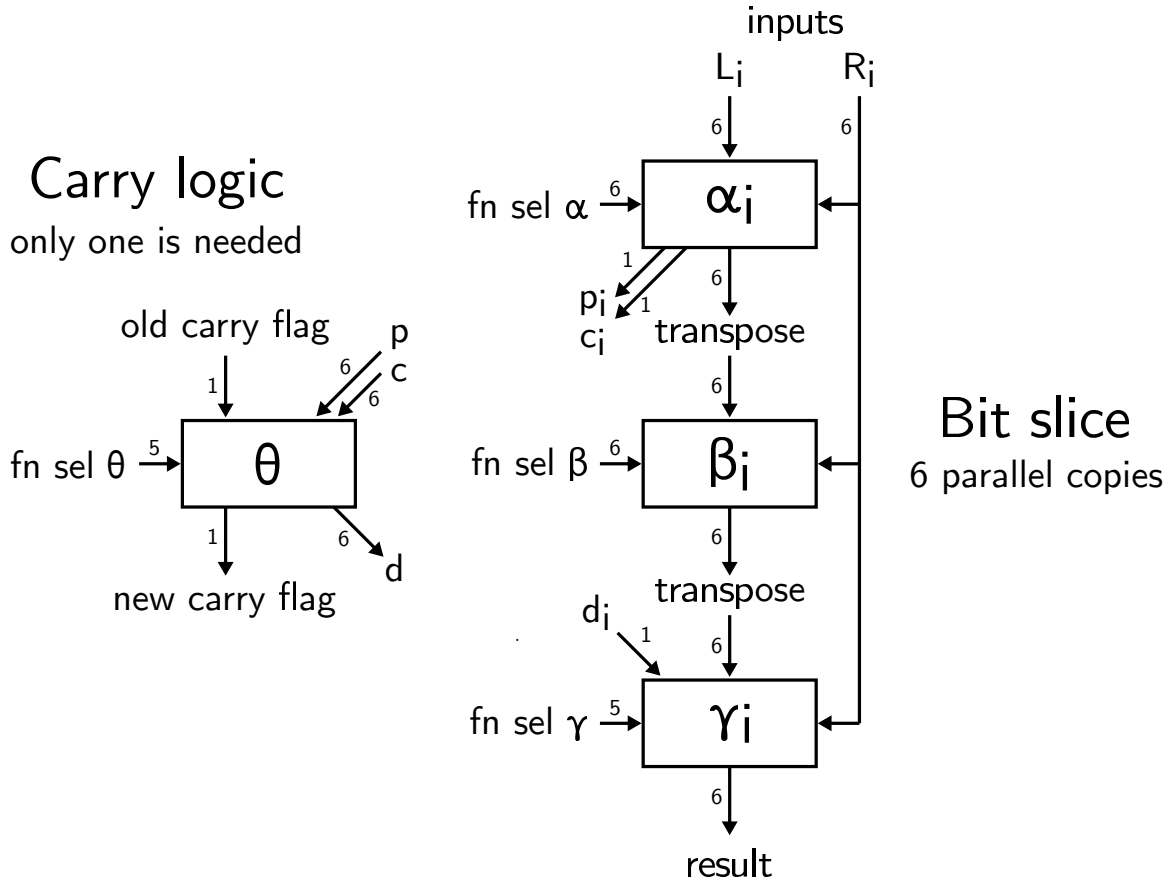
$\gamma$  layers operate on the rows of the array, and the RAMs of the  $\beta$  layer operate on the columns. With suitable firmware, the ALU can therefore operate on a word’s “rows,” then its “columns,” and then its “rows” again in a single pass. All this work is done in only three gate delays. Figure 5.3 shows a word’s bit positions in array form, with 0 and z indicating the least and most significant bits respectively. The relocation of the original bits following a transposition are also shown.

It’s appropriate to select vocabulary to mean “six-bit subword,” because this is the basic amount of computation handled by nearly all of the ALU RAMs. *Byte* or *nibble* would be confusing as a six-bit quantity, so I have chosen *tribble*. The derivation could be viewed as a “tri-nibble” that has been increased in size to the nearest multiple of three. The word may also be a good companion for *tetrad*, an archaic term for a four-bit quantity—except the architecture doesn’t use tetrads.

A question arises as to how the ALU RAMs know what they are to do from one instruction to the next. A “function select” is shown in the swizzler example of figure 4.8 (p. 63), but there hasn’t been room to draw such inputs in Figs. 5.1 or 5.2. Figure 5.4 shows a “zoom in” of a single bit slice along with  $\theta$ , where a few of each RAM’s input bits, marked “fn sel,” are supplied from a control unit. The number of function select bits, and therefore the number of possible operations, depends on how many address bits are left over for an SRAM after its other inputs have been assigned.

## 5.2 Word sizes for minicomputer architectures

The smallest common asynchronous SRAM ICs are  $64\text{Ki} \times 16$ , so they will have  $\log_2 65536 = 16$  input bits. If the left and right operands to each SRAM are six bits each, there will be four bits left, identifying 16 “slots,” to determine  $\alpha$  and  $\beta$ ’s operation.  $\gamma$  would only support 8 slots on account of needing a carry decision bit to each  $\gamma_i$  from  $\theta$ ; however, for those operations that do not consider any carry decision



**Figure 5.4:** Bit slice and carry propagation SRAMs for a 36-bit, three-layer ALU. The small digits indicate number of wires.

input, a slot can be shared between two operations, with a forced 0 or 1 carry decision from  $\theta$  selecting between the two. Like  $\gamma$ ,  $\theta$  only has three function select input bits due to requiring a bit for an incoming CPU carry flag,<sup>1</sup> and therefore only has 8 slots available for operations.

The smallest common synchronous SRAM ICs are  $256\text{Ki} \times 18$ . These devices have  $\log_2 262144 = 18$  input bits, so an ALU built with these components will offer four times as many slots compared to using the smallest asynchronous ICs. Because my architecture uses synchronous SRAM for the reasons given in section 3.2.2, figure 5.4's wire counts reflect this  $256\text{Ki} \times 18$  component size.

<sup>1</sup>The carry flag's name in my architecture is *T(emporal) flag*. It is described in section 5.3.

The simple, regular structure of three-layer ALUs, as well as the convenience of keeping the number of subwords and subword sizes consistent across  $\alpha$ ,  $\beta$ , and  $\gamma$ , strongly influence choice of word size for an architecture. Rather than choose a convenient power of two such as 16, 32, or 64 bits, a convenient square such as 16, 25, 36, 49, or 64 bits is more natural. The sizes of commercially available SRAMs further influence the word size selection. Here are some ideas to keep in mind.

- **16 bits** will not build much of a computer, but may be adequate for small controllers that are not networked, etc. If bit slices operate on four-bit subwords, many SRAM input bits—and therefore most of the SRAM—will go unused, although this drawback is somewhat beneficial from a firmware size standpoint. 16-wide SRAM ICs are available for primary storage.
- **25 bits** is not a standard memory width for primary storage. There may be acceptance issues with an odd word size, because most programmers have never dealt with one. As with four-bit subwords, five-bit subwords may not optimally fill bit slice SRAMs.
- **36 bits** is a common SRAM memory width and is suitable for primary storage. Six-bit subword operations will consume 12 input bits of the bit slice SRAMs, leaving four bits (with asynchronous SRAM) or six bits (with synchronous SRAM) available for function selection and carry information. 36 bits is larger than the physical address space of most machines that use SRAM for primary storage, so this word size will not impose need for memory segmentation or a limit on installed memory size.<sup>2</sup> 36 bits offers a wider range for integers than the 32-bit architectures that were already adequate in many cases. 36 bits is also much better than 32 bits for packing CPU instruction words: in my architecture, 36-bit instructions double the number of possible opcodes and

---

<sup>2</sup>This was a problem for 32-bit machines that offered byte addressability and thus had a 4 Gbyte memory limit. For many users, this was the only benefit of upgrading to 64-bit computers.

registers relative to what 32 bits would have afforded.

- **49 bits** is not a standard memory width for primary storage. There may be acceptance issues with an odd word size, because most programmers have never dealt with one. The bit slice SRAMs for minimum-size asynchronous SRAMs will likely be too full to support the desired number of operations, but synchronous SRAMs may be mostly okay. Some people may consider 49-bit floating-point formats to be disappointing relative to 64 bits.
- **64 bits** may be too many, although 64 bits is a multiple of common SRAM memory widths and works well for primary storage. The need for bit slice SRAMs to accept eight bits each for their left and right operands will leave very little selection of operations that can be supported. There is also no reason this word size is needed to hold pointers, considering that commercial SRAM ICs are only available to about 32 Mibit as of 2020, which for 64-bit words implies 19-bit addresses (22-bit addresses if byte addressability is offered).

Taking these ideas into account, my architecture uses 36-bit words. There are a few drawbacks in terms of relating to existing technology. The IEEE 751 floating-point specification, intended to make floating-point computations reproducible across architectures, does not define any 36-bit storage format or arithmetic. The Rust programming language specifically defines power-of-two integer sizes and no others. Bytes do not pack evenly, nor do IP addresses, nor standard cryptographic hashes, nor Unicode, nor Unicode's transformation formats such as UTF-8. But for languages like C that try to be word-size-agnostic, or Python with automatic support for integers larger than the machine architecture, 36 bits is not going to be a problem. Nor for ordinary users who aren't sure what a bit does, nor for infrastructure that may be controlled by a 36-bit solder-defined minicomputer.

**Table 5.1:** CPU flag meanings.

flag	name	purpose
I	Interrupt	array bound check failed
N	Negative	arithmetic: result $< 0$ ; logic: bit 35 set
R	Range	previous result did not fit destination
T	Temporal range	most recent result did not fit destination
Z	Zero	true result is all zeros

### 5.3 CPU flags

A summary of CPU flags appears in table 5.1. Here is a quick introduction to their intent and implementation:

- The **Z(ero)** flag is true if the output of an arithmetic or logical operation is zero. Zero detection involves each of the six  $\gamma$  RAMs providing a bit indicating that each RAM's output is zero. AND gates over those six bits are used to indicate a fully-zero 36-bit word. But there is a catch: the Z(ero) flag must only be set if the true result of a computation is zero, and never if a nonzero, overrange result happens to be zero modulo  $2^{36}$ .  $\zeta$ 's output bits 3 and 6 are used to ensure the Z(ero) flag will not be set falsely.
- The **N(egative)** flag has out-of-range concerns similar to the Z(ero) flag. It must only be set if the true result of a computation is negative.  $\zeta$  output bits 0 and 5, as well as  $\gamma_5$  bit 13, aid in N(egative) flag determination. Also, unsigned bitwise boolean operations are permitted to change the N(egative) bit as a convenience for branching according to the  $2^{35}$  place value of a result, because bitwise operations self-evidently cannot overflow.
- The **T(emporal overrange)**, usually abbreviated **T(emporal)**, flag combines and replaces the “carry” and “overflow” flags found on other architectures. This

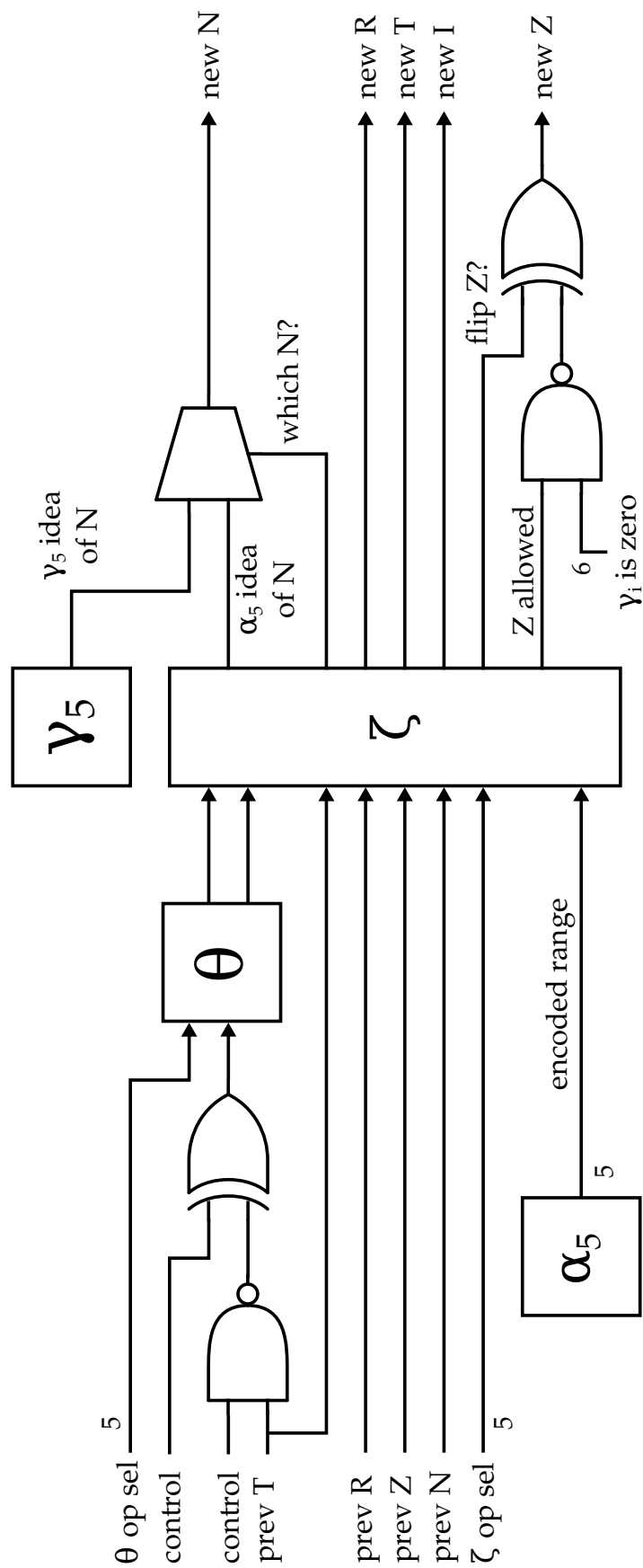
flag correctly indicates when an out-of-range addition, subtraction, or shift instruction produces an out-of-range result. This flag is also changed by more esoteric uses such as the RTGR (rotate through T going right) macro.

- The **R(ange)** flag is a “sticky” version of the T(emporal) flag. The R(ange) flag, which can only be cleared by the CRF (clear range flag) instruction, is set whenever an out-of-range arithmetic operation sets the T(emporal) flag. This sticky flag enables a programmer to execute long sequences of arithmetic without checking for overrange conditions until a tentative result is available, thereby simplifying programming and increasing execution speed.
- The **I(nterrupt)** flag is proposed as something the CPU may use for preemptive multitasking. Little is known yet about how this flag may work, although the BOUND instruction for array index checking contemplates setting the I(nterrupt) flag to raise an exception.

The Z(ero), N(egative), T(emporal), and R(ange) flags in the architecture are branchable, in that they determine whether conditional JUMP instructions are taken. At present there is a timing bottleneck in this branch decision, as well as an as-yet-undisigned flag save and restore mechanism to support preemptive multitasking. Figure 5.5 is a rough sketch of how the flags should work. Some information paths in this figure have significant operational and firmware consequences. On the input side:

- $\theta$ 's view of the T(emporal) flag can be altered by the control decoder. It can be received unchanged, inverted, forced to logic 0, or forced to logic 1. For example, the **A** (add) instruction forces the incoming T to be zero, while **AC** (add with carry) receives T unmodified in order to include it in the sum.
- $\zeta$  receives an “encoded range” that can be used for several purposes. Most often, it is  $\alpha_5$ 's warning that an add or subtract overflowed or underflowed, or





**Figure 5.5:** CPU flag approximate logic. Small digits that are not subscripts indicate more than one wire.

may overflow or underflow depending on the effective carry or borrow at the  $2^{30}$  place value.

- $\theta$  is able to offer  $\zeta$  a two-bit synopsis of  $\theta$ 's 13 input bits.

On the output side:

- $\zeta$  cannot directly generate the Z(ero) flag, because it operates at the same time  $\gamma$  is computing the ALU's output word. Instead,  $\zeta$  offers an output bit that can prevent the Z(ero) flag from being set in the event of arithmetic wraparound.
- $\zeta$  cannot directly generate the N(egative) flag, because  $\gamma$  is still computing the ALU's output word that determines the N(egative) flag after bitwise boolean operations. Instead,  $\zeta$  offers an output bit that selects between  $\gamma_5$ 's idea of the N(egative) flag for bitwise boolean, and  $\alpha_5$ 's idea of the N(egative) flag after adjustment by final carry or borrow information.

## 5.4 SRAM bit assignments

This chapter introduced the essential topology of a family of three-layer SRAM ALU designs, as incorporated in my 36-bit architecture. There remain a few choices to be made in the topology's surrounding logic, in part due to the branch decision timing bottleneck that I have not resolved yet. Accordingly, some of this chapter's descriptions may be challenging to follow. To assist the reader in clarifying doubts, tables 5.2 and 5.3 present all of the input and output bits for the SRAMs of my ALU.

## 5.5 Alternate diagram for ALU

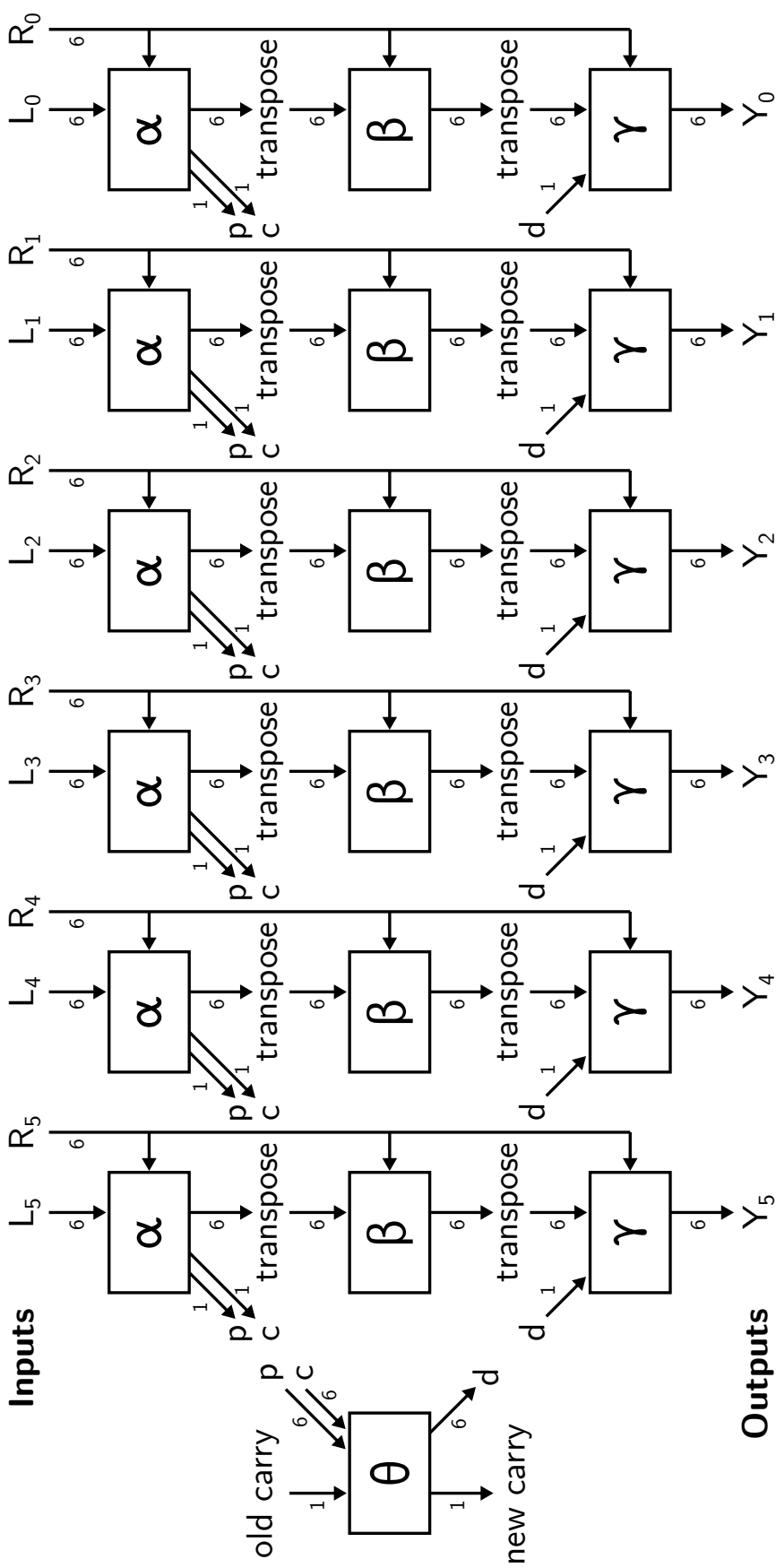
Figure 5.6 is a landscape redrawing of figure 5.1. It may be useful to someone who is writing about SRAM ALUs or documenting a specific SRAM ALU desires a drawing with a top-to-bottom information flow.

**Table 5.2:** ALU SRAM input bit assignments.

RAM	bit	description
$\alpha_0 - \alpha_5$	0–5	L input
$\alpha_0 - \alpha_5$	6–11	R input
$\alpha_0 - \alpha_5$	12–17	$\alpha$ operation
$\beta_0 - \beta_5$	0–5	transposed input from $\alpha$
$\beta_0 - \beta_5$	6–11	R input
$\beta_0 - \beta_5$	12–17	$\beta$ operation
$\gamma_0 - \gamma_5$	0–5	untransposed input from $\beta$
$\gamma_0 - \gamma_5$	6–11	R input
$\gamma_0 - \gamma_5$	12	tribble carry decision from $\theta$
$\gamma_0 - \gamma_5$	13–17	$\gamma$ operation
$\theta$	0–5	carry flags from $\alpha$
$\theta$	6–11	propagate flags from $\alpha$
$\theta$	12	modified previous T(emporal) flag
$\theta$	13–17	$\theta$ operation
$\zeta$	0	previous N(egative) flag
$\zeta$	1	previous R(ange) flag
$\zeta$	2	previous T(emporal) flag
$\zeta$	3	previous Z(ero) flag
$\zeta$	4–8	encoded range from $\alpha_5$
$\zeta$	9	advice from $\theta$ , such as $\vee(c_i)$ from $\alpha_i$
$\zeta$	10	advice from $\theta$ , such as $\vee(p_i)$ from $\alpha_i$
$\zeta$	11–15	$\zeta$ operation
$\zeta$	16–17	logic 0

**Table 5.3:** ALU SRAM output bit assignments.

layer	tribble	bit	description
$\alpha$	all	0–5	transposed output to $\beta$ layer
$\alpha$	all	6	carry to $\theta$
$\alpha$	all	7	propagate to $\theta$
$\alpha$	5	8–12	encoded range to $\zeta$
$\alpha$	0–4	8–17	reserved
$\alpha$	5	13–17	reserved
$\beta$	all	0–5	untransposed output to $\gamma$ layer
$\beta$	all	6–17	reserved
$\gamma$	all	0–5	ALU output, left copy (node 2)
$\gamma$	all	6–11	ALU output, right copy (node 5)
$\gamma$	all	12	zero flag for tribble
$\gamma$	5	13	copy of $Y_{35}$ ( $\gamma$ 's idea of N(egative) flag)
$\gamma$	0–4	13–17	reserved
$\gamma$	5	14–17	reserved
$\theta$	n/a	0–5	carry decisions to $\gamma$ layer
$\theta$	n/a	6	advice to $\zeta$ , such as $\vee(c_i)$ from $\alpha_i$
$\theta$	n/a	7	advice to $\zeta$ , such as $\vee(p_i)$ from $\alpha_i$
$\zeta$	n/a	0	$\zeta$ 's idea of new N(egative) flag or 0
$\zeta$	n/a	1	new R(ange) flag
$\zeta$	n/a	2	new T(emporal) flag
$\zeta$	n/a	3	logic 1 iff Z(ero) flag is allowed
$\zeta$	n/a	4	new I(nterrupt) flag
$\zeta$	n/a	5	new N(egative) flag comes from $\zeta$ (if 0) or $\gamma$ (if 1)
$\zeta$	n/a	6	invert new Z(ero) flag
$\zeta$	n/a	7–17	reserved



**Figure 5.6:** Block diagram of a 36-bit ALU. Each square corresponds to one SRAM IC. The two 36-bit transpose operations are self-inverse. Both connect output  $i$  of tribble  $j$  to input  $j$  of tribble  $i$ . Small digits that are not subscripts indicate number of wires. This is a landscape redrawing of figure 5.1.

## 6

# Two-layer ALU structure

My research to introduce solder-defined minicomputers required a commitment from me to show either how capable and fast such machines could be, or how small and inexpensive. I chose capable and fast in order to develop a general theory as fully as I could, and have that theory apply to as many potential uses as possible. But in the course of this work, I discovered some non-obvious shortcuts that could lead to machines that use fewer components and less labor to build. These abbreviated machines, if developed, may appeal to hobbyists, students, and makers who are building their first minicomputers.

This chapter describes a possible shortcut for ALUs that use commercially available asynchronous SRAMs. This is far short of a functioning machine specification, but it might spark some creative designs.

## 6.1 An elegant two-layer ALU for 36-bit words

Common asynchronous SRAMs offer a capability that is not available in their synchronous counterparts: either half of their data lines can be disabled for input and output. Common synchronous SRAMs can disable half of their data lines for input, but not for output. Although the architecture of this dissertation does not use asynchronous SRAMs, this chapter has been included to show a novel ALU that can be

built using asynchronous SRAMs.

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

Figure 6.1 shows a 36-bit ALU made from just ten RAMs in two layers. These RAMs operate on 6-bit subwords to do all the work, and this illustration only shows the wiring for the subwords themselves. The various control signals are drawn separately for legibility as figure 6.2, but note that the ten boxes refer to the same ten RAMs in both figures.

The top layer of the ALU does almost all of the work. Sixteen functions accepting two inputs of six bits are stored in the upper bytes of the top-layer RAMs. The functions are selected via a 4-bit control input, while the byte select lines force the lower bytes to high impedance. This mode provides a traditional mix of additive and logical functions, as well as subword multiplications. There is no hardware support for  $36\text{-bit} \times 36\text{-bit}$  multiplication.

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

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

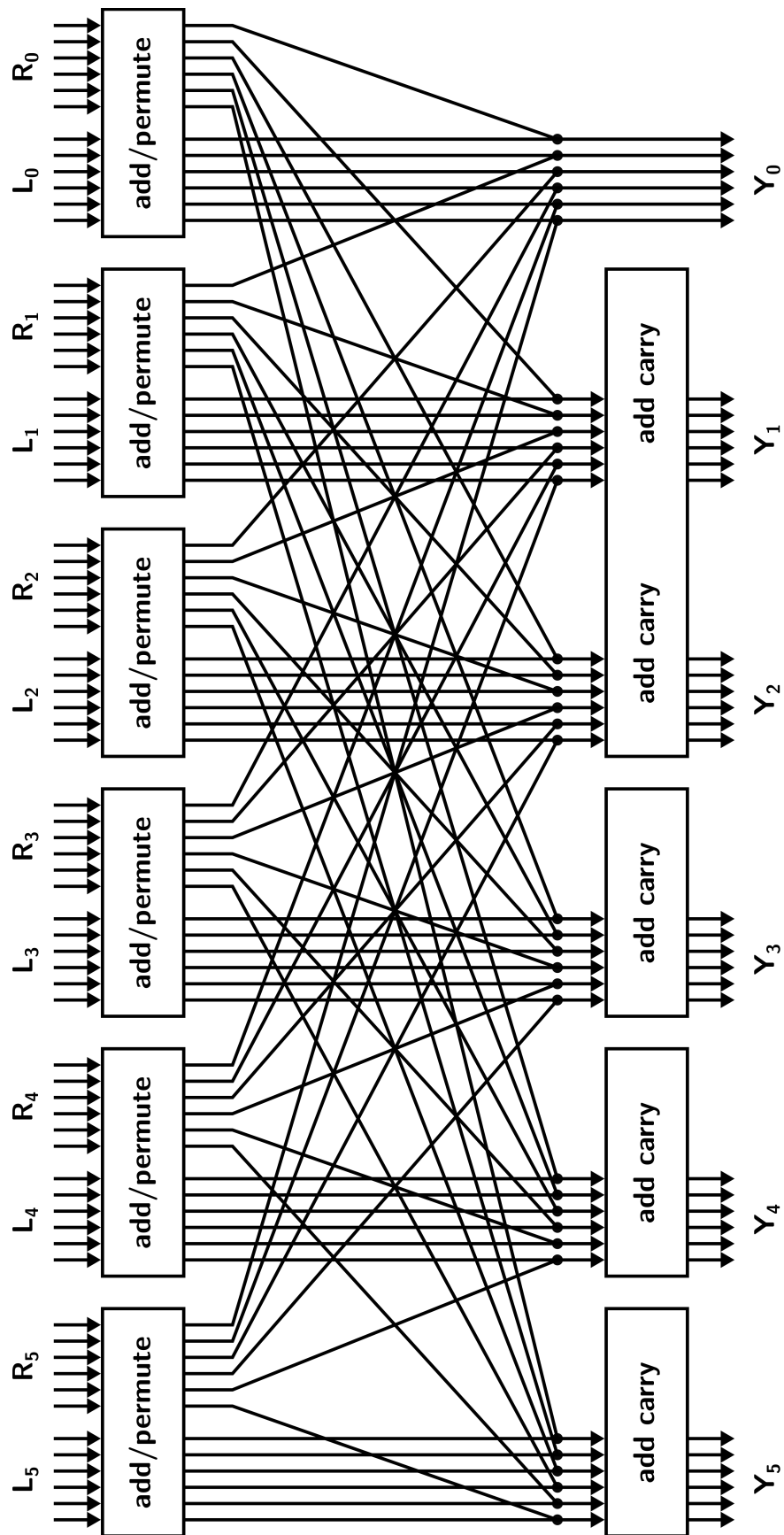


Figure 6.1: Data signals for a 36-bit, two-layer ALU.



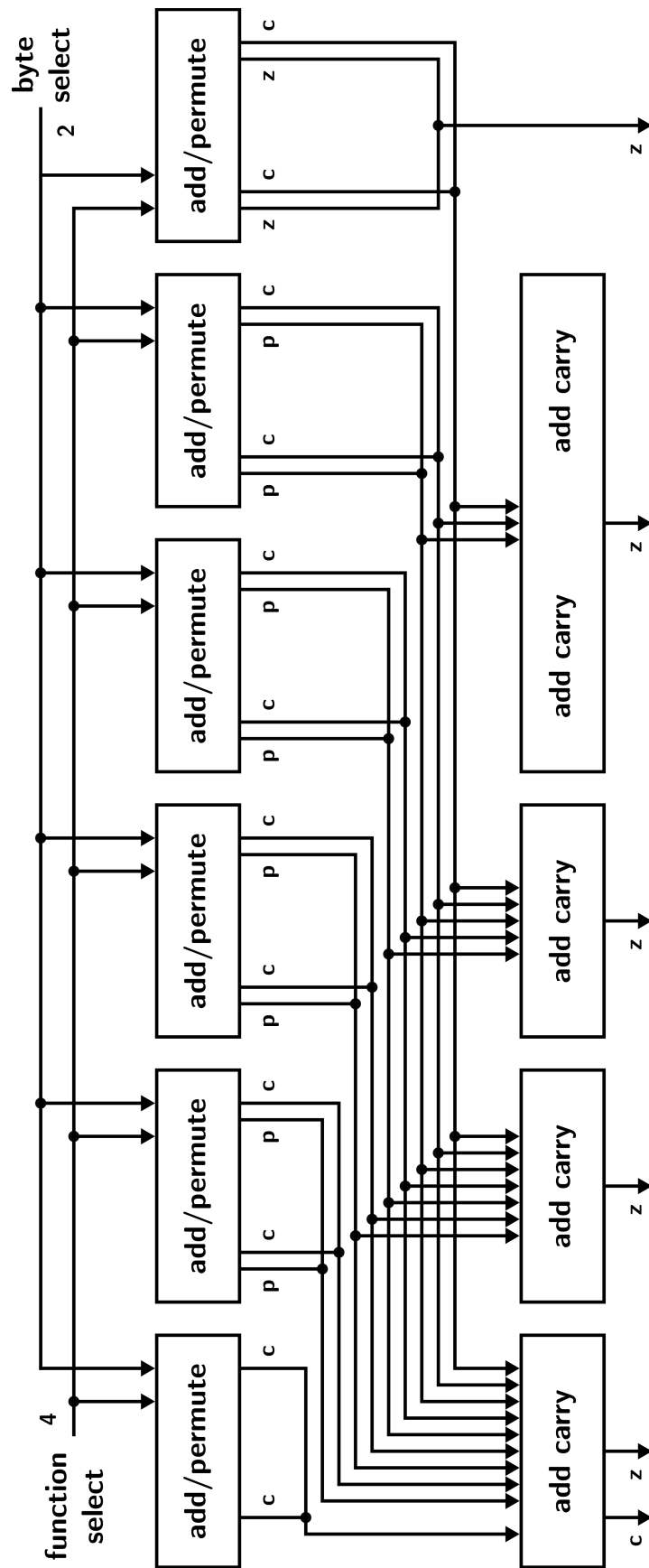


Figure 6.2: Control signals for a 36-bit, two-layer ALU.

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

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

This ALU has no carry input, because all address lines on the top right RAM are tasked for other uses. As the word size is already 36 bits, an occasional branch on the carry flag's value is typically enough support for adding and subtracting wider integers.

Table 6.1 maps out space in the first RAM layer for suggested operations of this ALU. This table's purpose is to show (i) an appropriate set of operations to implement, and (ii) that the complete set can be implemented within the address space available. Several provisions made in this table aren't immediately obvious, so here is some explanation.

This ALU always produces a 37-bit signed result when adding or subtracting irrespective of operand signedness, but the  $-(2^{36})$  bit, inaccurately termed the "sign bit," does not fit within a word and is not retained after the result is inspected for overflow. The inspection is fairly simple. If the  $-(2^{36})$  bit does not match the  $2^{35}$  bit and the result is signed, overflow will occur upon truncation. If the result is unsigned and the  $-(2^{36})$  bit is set, overflow has already occurred. Overflow will not occur if neither situation applies. For signed results, the  $-(2^{35})$  bit then becomes known as the  $-(2^{35})$  bit. All this would work out fine with 37-bit signed inputs, but what come from

**Table 6.1:** Suggested operations for a 36-bit, 2-layer ALU.

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

the registers are the customary 36-bit overloaded-signedness inputs. Our workaround is to have separate ALU operations for the several operand type combinations; this is why table 6.1 requires three add and four subtract operations. The surrounding CPU will test for wrap-around and latch a R(ange) flag when needed.

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

The reader might wonder how overflow is avoided for short multiplication, as

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

Permutations, swizzles, and S-box operations usually require word transposition and are specified accordingly. Plain permutations within subwords also are supported. A set of 64 unary operations, selectable at the subword level, are included. One of these operations computes the fixed-point reciprocal  $\lfloor (2^{36}-1) \div \max(n, 1) \rfloor$ , the largest unsigned word one can multiply by  $n$  without overflow. Most of the remaining 63 are simpler; table 7.16 (p. 161) can be used as a starting list of candidate operations.

Shifts and rotations are implemented in two CPU instructions, because these require two permutation stages and two transpositions. The number of positions to move by must be replicated beforehand into each subword. Thus a rotation may take four instructions if the number of positions cannot be expressed as a constant: two to transpose, replicate, and transpose back the rotation amount, and two to accomplish the rotation itself. Negative shifts and rotations are unavailable, because the range  $-35 \dots 35$  is too large to express within a 6-bit subword (64 slots will not hold 71 possible values). Because right rotations are expressible as left rotations, having a separate function for right rotation is not helpful. When rotating right by a variable amount, no subtraction from 36 is required prior to replication into subwords, because one of the 64 available swizzle operations can combine the needed subtraction and permutation. No speed is lost by leaving out right rotation.

Figure 6.2 purposely omits some control logic that would easily confuse readers who are trying to understand this ALU for the first time. In addition to what is drawn, the propagate signals from the five RAMs which offer them are combined in a five-input NAND gate. The output of that gate is used only for left shifts, and indicates that signed or unsigned overflow resulted from a multiplication by a power

of two. There are more subtleties. The rightmost RAM doesn't have an output bit left for overflow detection; it got used for zero detection. Here we are helped by a kludge. Instruction two of a left shift operates on transposed words, meaning that five of the six bits not checked during instruction one can be tested by other RAMs on instruction two. This leaves only the 20 bit not tested, a bit which can only overflow during shifts of more than 34 or 35 bit positions depending on signedness. Smaller fixed shifts needn't check the 20 bit at all. Of the remaining cases, the only shift amounts that can overflow *always* do when shifting a nonzero word.<sup>1</sup> The few programs that can't be written to preclude all possibility of such unorthodox shifts must assume responsibility for checking the 20 bit. But for most software, this ALU detects left shift overflow as perfectly as it detects additive wrap-around. Masking for left shift can't happen until instruction two, or overflow from the rightmost subword will go undetected.

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

The four parenthesized operations in table 6.1 might be useful, but they can be synthesized using small combinations of the other operations. They may be displaced in the event that more important operations are identified.

This ALU is not the most capable one a user can build on her own for high-confidence applications, but it already towers over prior CPUs made from discrete components. For uses where its speed and operations are suitable, anyone with the

---

<sup>1</sup>The surrounding instruction set implements logical left shifts as unsigned left shifts with the overflow signal suppressed. This is the ordinary case when the rightmost bit might shift more than 35 positions.

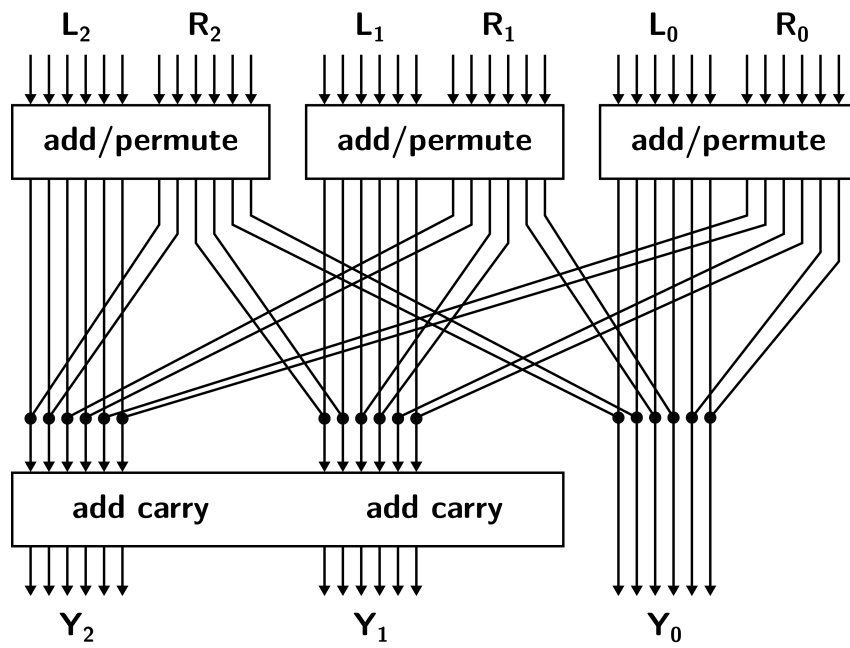
motivation can build or have built a practical, robust, inspectable, and tamper-evident ALU for 36 bits using ten small, fungible RAMs. I believe that for the total component count, as well as the limited availability of parts that might be regarded as trustworthy, it would be difficult to improve on the overall design of this ALU.<sup>2</sup> This device is conceptually mature, and the design frontier has moved to the CPU which surrounds it.

## 6.2 A tiny ALU for 18-bit words

The ALU of Figs. 6.1 and 6.2 can't be cut down to build a 24-bit or 30-bit version, because 24 and 30 are not multiples of 16 or 25 (the squares of the number of subwords in the semi-swizzler) respectively. This means we cannot meet the requirement to have a self-inverse transposition and a single subword size at the same time. But we can build 12- and 18-bit versions, and these ALUs require only three or four RAMs respectively. The transpose wiring, shown for 18 bits on figure 6.3, is rearranged to minimize loss of left shift overflow detection. Only the  $2^0$  and  $2^1$  bits are unchecked, so arithmetic shifts of up to 15 bits signed and 16 bits unsigned come with range checking. The right half of figure 6.2 shows the control signals, where the worst fan-out (including two sinks not shown) is five. The instruction set can be as stated in table 6.1, although the S-box outcomes will not agree with the results of their 36-bit counterparts.

---

<sup>2</sup>A few faster but very expensive and power-hungry computers, such as the CRAY-1, were built from discrete current-mode logic components.



**Figure 6.3:** Data signals for a 18-bit, two-layer ALU.

# 7

## A three-layer, 36-bit ALU firmware

### 7.1 What is SRAM ALU firmware?

Chapter 5 described the physical organization of a 36-bit, three-layer SRAM ALU, explaining why the ALU is arranged as it is, and (at least at a high level) how to solder it together. But the chapter stopped short of showing how the circuit functions as an ALU, or any real-world capabilities in terms of ability to be programmed. These characteristics depend on the information contained in the RAMs, which I have chosen to call *firmware*. So if we think of an ALU as having a body and soul, chapter 5 described its body, or physical structure, and this chapter bares its soul, or firmware.

I chose to use the term *firmware* over *microcode*. The distinctions between these two words are blurry, and usage varies. I have long thought of microcode as being like little subroutine calls within a CPU's programmer-inaccessible realm, where a simple instruction is expanded into a sequence of smaller instructions in a lower-level machine language. This is my personal interpretation, molded from reading [Patterson83] before I reached high school. My firmware does not contain any "sequences" to speak of, and it never loops. For any instruction, the  $\alpha$ ,  $\beta$ , and  $\gamma$  layer RAMs are read at most once, and all instructions take exactly the same number of cycles. These to me are more suggestive of the word *firmware* than *microcode*. It's the case that the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers are read one at a time, suggestive of microcoding, but their order of



use is always the same, suggesting firmware.

In my architecture, CPU instructions are always 36 bits, and the nine most significant bits are always the *opcode*, a unique identification of the instruction to execute. In the case of ALU instructions, which read from two registers and write a result to one register, this instruction format is:

opcode	dest. register	left register	right register
bits 35–27	bits 26–18	bits 17–9	bits 8–0

This format is stated for the reader’s understanding only. The ALU doesn’t care about the instruction format or the size of its fields, nor does it ever know what opcode it is executing. Instead, the ALU needs to know the function select inputs that decide the operation each of its SRAMs needs to compute. Figure 5.4 (p. 79) may help the reader recall this relationship. Each opcode is translated—via SRAM, of course—to a few dozen control bits, certain of which specify the operations for the participating SRAMs. For example, the **AW** “add with wrap” opcode is expanded by the control decoder SRAMs to operations named  $\alpha$ .add.uu (add pairs of unsigned tribbles in the  $\alpha$  layer),  $\beta$ .id (do nothing in the  $\beta$  layer),  $\gamma$ .add (apply carry decisions in the  $\gamma$  layer),  $\theta$ .add (compute carry decisions in the  $\theta$  RAM based on propagate and carry information from  $\alpha$ ), and  $\zeta$ .wrap (set flags as appropriate for the **AW** instruction). The incoming T(emporal) flag is forcibly zeroed, because **AW** must not include a previous carry in its sum.

Programming languages include a few operators and functions that my ALU can do quickly, but not as a single instruction. Counting the number of 1s in a 36-bit word requires two instructions, for example. Assemblers for this architecture should not require the programmer to remember these esoteric instructions, when the step they are trying to program is a simple popcount. So in addition to describing *operations* that individual SRAMs do, and *opcodes* that are made of operations, this chapter defines names and behavior for more than 60 *macros* that are made of *instructions*.

Although the core of every instruction is an opcode, most macro definitions require additional information, such as temporary registers or specific left or right operands, in order to execute correctly. So although it would be easier to remember that macros are made of opcodes made of operations, it is more precise to say that macros are made of instructions that each contain exactly one opcode made of operations.

A macro may be as short as one instruction. Single-instruction macros can improve the legibility of an assembly language. For example, a programmer who wishes to reverse the order of bits in a word can do so, using the PAIT opcode with the “reflected identity” permutation from table 7.8 for the rows and columns of figure 5.3 (p. 77) simultaneously. This requires a right operand of 131313131313‘o. Simply using the MIR “mirror bits” macro from table 7.18, which can work as if it is just another opcode, is more convenient for the programmer.

My assembler as of October 2022 implements almost none of the macros described in this dissertation, due only to limits on my available time. The mechanism by which this may work, as well as whether the mechanism will be extensible to support traditional assembler macros written for ordinary programs, remains to be decided.

The next two sections are corequisites. The section describing available opcodes and their implementations is placed first, because assembly-language programmers will likely rely more on this section. The section on SRAM operations appears second, and is useful for understanding the opcode implementations.

## 7.2 ALU opcodes and their implementations

Table 7.1 summarizes opcodes and macros this ALU implements by category. Many opportunities exist to grow this list, such as single-instruction macros for increment, decrement, and negate, as well as opcodes to support division, floating-point arithmetic, and faster long multiplication. Tables 7.2, 7.3, 7.4, 7.5, 7.6, and 7.10 provide additional detail and implementation specifics. The columns of these tables are:

**Table 7.1:** Principal opcodes and macros for a 36-bit, 3-layer ALU.

### Additive

A	add
S, RS	subtract, reverse subtract
...C	with carry
...W	with wrap
...WC	with wrap and carry

### Compare

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

### Shift and rotate

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

### Multiply

ML	tribble multiply, low result
MH	tribble multiply, high result
MHNS	MH without left shift
DSL	double-register left shift
SWIZ	copy tribble to all positions

### Bit permute

PAT	permute across tribbles
PIT	permute inside tribbles
PAIT	simultaneous PIT and PAT

### Mix

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

### Miscellany

CRF	clear R(ange) flag
HAM2	2 <sup>nd</sup> instr. of popcount macro
NUDGE	substitute rightmost bits
TXOR	transposing XOR

### Bitwise boolean

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

### Leading and trailing bit macros

BO, BZ ...	brighten ones/zeros
EO, EZ ...	erase ones/zeros
FO, FZ ...	find one/zero
GO, GZ ...	grow one/zero
LO, LZ ...	light ones/zeros

### More macros

ABS	absolute value
CLO	count leading ones
CLZ	count trailing ones
CTO	count trailing ones
CTZ	count trailing zeros
CX	check and extend
FABS	fast absolute value
LFSR	linear feedback shift register
MIR...	mirrored increment/decrement
PARTY	parity
POPC	popcount
PR...	prepare to rotate by non-const.
PS...	prepare to shift by non-const.
RTG...	rotate through T(emporal) flag
STG...	shift into T(emporal) flag
XPOLY	XOR polynomial if T set

<b>family</b>	The assembler mnemonic for the instruction. A given mnemonic can generate several different opcodes, depending on the signedness of its arguments and results. For example, the mnemonic <b>A</b> (add) in table 7.2 includes overflow checking that requires awareness of the signedness of both arguments and the result, and this awareness is supplied by using eight different opcodes for <b>A</b> .
<b># of</b>	The total number of opcodes corresponding to the mnemonic in the family column. The assembler automatically selects the correct opcode based on the declared signages of the registers involved. A mechanism exists for the programmer to override this automatic selection.
$\alpha$	The operation specified in the function select input to the $\alpha$ RAMs. See table 7.11.
$\beta$	The operation specified in the function select input to the $\beta$ RAMs. See table 7.12.
$\gamma$	The operation specified in the function select input to the $\gamma$ RAMs. See table 7.13.
$\theta$	The operation specified in the function select input to the $\theta$ RAM. See table 7.14.
$\zeta$	The operation specified in the function select input to the $\zeta$ RAM. See table 7.15.
<b>T</b>	Handling of the incoming temporal flag. <b>T</b> means pass unchanged. <b>!T</b> means pass inverted. <b>0</b> means force to logic 0. <b>1</b> means force to logic 1.
<b>description</b>	A brief explanation of what the mnemonic does.

### 7.2.1 Additive opcodes

Table 7.2 shows the basic addition and subtraction opcodes. The  $\alpha$  and  $\gamma$  layers provide carry-skip addition, with  $\beta$  not participating. Carry decisions based on  $\alpha$ 's propagate and carry outputs are made by  $\theta$  and distributed to  $\gamma$ . If “with carry” is specified in the opcode, the incoming **T**(emporal) flag is also reflected in carry decisions.

**Table 7.2:** Additive instructions and their implementations.

family	# of	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
A	8	[+]	id	add	add	[+us]	0	add
S	8	[−]	id	add	add	[+us]	1	subtract
RS	8	[r−]	id	add	add	[+us]	1	reverse subtract
AC	8	[+]	id	add	add	[+us]	T	add with carry
SC	8	[−]	id	add	add	[+us]	!T	subtract with carry
RSC	8	[r−]	id	add	add	[+us]	!T	reverse subtract with carry
AW	1	add.uu	id	add	add	wrap	0	add with wrap
SW	1	sub.uu	id	add	add	wrap	1	subtract with wrap
RSW	1	rev.uu	id	add	add	wrap	1	reverse subtract with wrap
AWC	1	add.uu	id	add	add	wrap	T	add with wrap, carry
SWC	1	sub.uu	id	add	add	wrap	!T	subtract with wrap, carry
RSWC	1	rev.uu	id	add	add	wrap	!T	rev. subtr. w. wrap, carry

**signedness choices for  $\alpha$  and  $\zeta$ :**

[+]	one of:	add.uu	add.us	add.su	add.ss
[−]	one of:	sub.uu	sub.us	sub.su	sub.ss
[r−]	one of:	rev.uu	rev.us	rev.su	rev.ss
[+us]	one of:	add.u	add.s		

Addition and subtraction are stratified by signedness, causing the 12 mnemonics to assemble to 54 distinct opcodes, depending on the signedness of the arguments and result. Listing 7.1 shows how the 36 subtraction instructions look in assembly language; addition works comparably. This stratification permits not only mixed-sign arithmetic; e.g., add a signed register to an unsigned register, but also alerts the CPU as to the signedness of the destination register so that overflow is detected correctly. Section 7.3.1 offers more specifics as to how these checks work. The “with wrap” opcodes suppress overflow detection, which may be useful when the R(ange) flag is being monitored for a surrounding computation.

```

unsigned au bu cu
signed   as bs cs

```

```

; ordinary      ; subtract      ; reverse      ; reverse subtract
; subtract      ; with carry    ; subtract    ; with carry

cu = au - bu    cu = au -- bu    cu = au ~- bu    cu = au ~-- bu
cu = au - bs    cu = au -- bs    cu = au ~- bs    cu = au ~-- bs
cu = as - bu    cu = as -- bu    cu = as ~- bu    cu = as ~-- bu
cu = as - bs    cu = as -- bs    cu = as ~- bs    cu = as ~-- bs
cs = au - bu    cs = au -- bu    cs = au ~- bu    cs = au ~-- bu
cs = au - bs    cs = au -- bs    cs = au ~- bs    cs = au ~-- bs
cs = as - bu    cs = as -- bu    cs = as ~- bu    cs = as ~-- bu
cs = as - bs    cs = as -- bs    cs = as ~- bs    cs = as ~-- bs

```

; Signedness is ignored for the following:

```

<wrap> cs = au - bs      ; subtract with wrap
<wrap> cu = as -- bu     ; subtract with wrap and carry
<wrap> cs = au ~- bs     ; reverse subtract with wrap
<wrap> cu = as ~-- bs    ; reverse subtract with wrap and carry

```

**Listing 7.1:** 36 subtraction opcodes. (Some tildes may look like minus signs.)

## Reverse subtraction

An extra subtract operation is offered with the arguments reversed, a special provision for CPUs with fast hardware multipliers. The reason is that fast multiplication breaks an important register file invariant. In ordinary subtraction,

$$-L + R = R - L,$$

allowing a simple reversal of the left and right operands in an instruction word to perform a reverse subtraction. But because the physical architecture needs to obtain the left and operands simultaneously, separate left and right copies of the register file are kept. The copies are ordinarily identical, as must be the case for the above identity to hold.

On a machine with a hardware multiplier, the 72-bit result of a multiplication needs to be split between the 36-bit register copies in order to be written to a register in one clock cycle. This will cause a register to evidence different values, depending on its position in subsequent instruction words. Because addition is commutative, a subsequent addition can involve either the high or low word of a result based on whether it comes via the left or right operand. But subtraction is not commutative, and therefore would force use of the left or right operand to obtain the high or low word as required. If that operand is misplaced for the subtraction desired, the sign of the result will be incorrect, necessitating an extra negation instruction that eats at the speed that was sought in building a hardware multiplier in the first place. By offering an explicit reverse subtraction operation, the architecture is able to preclude wasting an instruction on negation between multiplication and subtraction.

Section 8.3.1 will return to the topic of reverse subtraction.

## Smaller SRAMs

An earlier version of the firmware for  $64\text{Ki} \times 16$  asynchronous SRAMs used a  $128\text{Ki} \times 16$  SRAM for  $\alpha_5$  only for signedness stratification. The  $\alpha_0$ – $\alpha_4$  SRAMs in the present implementation have four identical slots for the four left and right operand signedness cases for A, S, RS, AC, SC, and RSC, because only  $\alpha_5$  needs to change with signedness. A lot of firmware memory can be conserved if a design is needed for smaller SRAMs.

## 7.2.2 Bitwise boolean opcodes

There exist sixteen bitwise boolean functions with two inputs, and as table 7.3 shows, this ALU implements all of them. Interestingly, the individual layers only implement a handful of these.  $\alpha$  supplies the common AND and OR operations.  $\beta$  is very limited in its flexibility to assist, because it's operating on a transposed word. The only change it can safely make is to invert all bits, thereby extending what  $\alpha$  can do to include NL, NAND, and NOR.  $\gamma$  doesn't have enough space to offer many operations,

**Table 7.3:** Bitwise boolean instructions and their implementations.

opcode	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
IGF	boo	id	kil	t	logic	1	ignorant false
IGT	boo	id	mux	t	logic	1	ignorant true (1)
XL	boo	id	mux	t	logic	0	exactly left
XR	boo	id	mux	t	logic	1	exactly right
NL	boo	not	add	t	logic	0	not left
NR	boo	id	xor	t	logic	1	not right (2)
AND	and	id	add	t	logic	0	AND
NAND	and	not	add	t	logic	0	NAND
OR	or	id	add	t	logic	0	OR
NOR	or	not	add	t	logic	0	NOR
XOR	boo	id	xor	t	logic	1	XOR
XNOR	boo	not	xor	t	logic	1	XNOR
LANR	or	id	xor	t	logic	1	left and not right
RANL	and	id	xor	t	logic	1	right and not left
LONR	and	not	xor	t	logic	1	left or not right
RONL	or	not	xor	t	logic	1	right or not left

(1) The assembler supplies an all-ones right argument for IGT.

(2) The assembler supplies an all-ones left argument for NR.

because its carry input cuts in half the available space for their implementation.<sup>1</sup> But by XORing what arrives from  $\beta$  with the original right operand,  $\gamma$  adds another five operations.  $\gamma$ 's multiplex and kill operations manage to fill out the remaining constants and identities.

Although the opcodes of table 7.3 technically implement the boolean functions listed, not all do so semantically. For instance, the boolean function **NR** (“not R,” or bitwise invert) function theoretically ignores its left input, but this ALU cannot both ignore its left input and invert its right input due to SRAM space conservation. Instead, the assembler has to substitute all ones in place of the left input register

---

<sup>1</sup>At the time these opcodes were decided,  $64\text{Ki} \times 16$  asynchronous SRAMs were anticipated. There is more space today, but the implementation is left in a manner to support the smaller RAMs.



**Table 7.4:** Comparison instructions and their implementations.

family	# of	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
MAX	8	[cmp]	id	mux	lt	[+us]	0	maximum
MIN	8	[cmp]	id	mux	gt	[+us]	0	minimum
CMP	4	[sub4]	id	add	add	cmp	1	compare (1)
BOUND	2	[sub2]	id	add	add	bound	1	check array index (2)

**signedness alternatives for  $\alpha$  and  $\zeta$ :**

[cmp]	one of:	cmp.uu	cmp.us	cmp.su	cmp.ss
[+us]	one of:	add.u	add.s		
[sub2]	one of:	sub.uu	sub.us		
[sub4]	one of:	sub.uu	sub.us	sub.su	sub.ss

- (1) CMP causes the N and Z flags to reflect the sign of L−R.
- (2) BOUND does nothing if  $0 \leq L < R$ . Otherwise, an exception is raised.

specified in the source code. This doesn't present a problem for users, compilers, or assembly language programmers, but it departs from feng shui. It also ties up a register in order to provide an all-ones immediate value.

### 7.2.3 Compare opcodes

Table 7.4 contains opcodes that do comparisons. The maximum and minimum operations are unusual in CPUs: traditionally a compare and branch is used, but we accomplish these in one instruction. Not only this, but the two operands and results can be of any signedness. MAX and MIN are implemented by having  $\alpha$  and  $\beta$  pass the left argument unchanged, and  $\gamma$  select between the output of  $\beta$  and the right operand depending on whether the carry decision bits are all ones or all zeros.  $\theta$  is able to compute which argument is larger on the basis of tribble comparisons made by the  $\alpha$  RAMs and delivered via the propagate and carry wires.

The BOUND operation is intended for array subscripts. Although BOUND's im-

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

The **CMP** instruction does a simple comparison and is identical to subtraction, except the control unit skips the step where the result is written to a register. It likewise does not check to see if the difference fits in any particular format, and will never set the T(emporal) or R(ange) flags.

## 7.2.4 Shift and rotate opcodes

The shift opcodes in table 7.5 have unconventional definitions. *Arithmetic shift* means multiplication or division by a power of two, rounding towards  $-\infty$  in the case of division. In traditional forums, arithmetic shift means that a signed number is being shifted. But my position is that it's moot whether the number is unsigned or signed; what is important is whether the intent is to perform arithmetic, and if this is the intent, the result must be tested for overrange irrespective of signedness. A shift in

---

<sup>2</sup>Anyone who can afford a  $2^{36}$ -word SRAM data memory would likely also want a 64-bit, 3-layer ALU built from  $1\text{Mi} \times 18$  SRAMs. **BOUND** would then give correct results up to  $2^{63} - 1$ .

**Table 7.5:** Shift and rotate instructions and their implementations.

opcode	signedness	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
ASL	$u \rightarrow u$	e36	shl	rol	t	asl.uu	0	arithmetic shift left
ASL	$u \rightarrow s$	e35	shl	rol	t	asl.us	0	arithmetic shift left
ASL	$s \rightarrow u$	e36	shl	rol	t	asl.su	0	arithmetic shift left
ASL	$s \rightarrow s$	e35	shl	rol	t	asl.ss	0	arithmetic shift left
ASR	$u \rightarrow u$	asr	shr	rol	t	asr.uu	0	arithmetic shift right
ASR	$u \rightarrow s$	asr	shr	rol	t	asr.us	0	arithmetic shift right
ASR	$s \rightarrow u$	asr	shr	rol	c5	asr.su	0	arithmetic shift right
ASR	$s \rightarrow s$	asr	shr	rol	c5	asr.ss	0	arithmetic shift right
LSL	any	boo	shl	rol	t	logic	0	logical shift left
LSR	any	boo	shl	rol	t	logic	0	logical shift right
ROL	any	boo	rol	rol	t	logic	0	rotate left (1)

(1) Right rotations are implemented as their left complements.

the absence of any intent to multiply or divide is called in this dissertation a logical shift. Logical shifts are not tested for overflow, because the bits involved do not represent quantities.

The logarithmic shifter is implemented by the  $\beta$  and  $\gamma$  layers. The shift and rotate operations require the number of bit positions to be encoded into every tribble of the right argument. These identical copies can be provided by the compiler, assembler, or programmer if the shift amount is fixed. Otherwise, one of the following macros may be used to check range and replicate the shift amount into each tribble:

macro	stacked unary	description
PRL	su.rlprep	prepare to rotate left
PRR	su.rrprep	prepare to rotate right
PSL	su.slprep	prepare to shift left
PSR	su.srprep	prepare to shift right

The  $\gamma$  layer's participation in the logarithmic shifter is expressed in terms of leftward rotation only, because RAM in that layer may be scarce. This means that

all shift and rotation operations are normalized to a left perspective, so to shift right one position, the argument supplied must indicate a left rotation of 35 positions. The stacked unary operations that prepare shift and rotate arguments account for this and do range checking on the shift or rotate amount. Rotations in excess of 63 bits require extra instructions due to the modulo-36 division needed.

Negative shift amounts aren't directly supported by this ALU, because a signed tribble can only represent numbers in the range  $-32 \dots 31$ , while the range of possible shifts is at least  $-35 \dots 35$ . If a program needs to shift by a signed variable amount, a branch is required to perform negative shifts correctly.

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

## 7.2.5 Multiply opcodes

Although this ALU isn't specified with a hardware multiplier, the **ML**, **MH**, **MHNS**, **DSL**, and **SWIZ** opcodes significantly accelerate unsigned multiplication in software. Their implementation is included in table 7.6. For short multiplication, the right operand must be a factor in the range of 0 to 63, and replicated across all six tribbles. **ML** (multiply low) and **MH** (multiply high) multiply across the tribbles of the left and right operands pairwise, with **ML** returning the low six bits of each tribble's result, and **MH** returning the high six bits. **MH** throws in a six-bit left shift to align with **ML**, so the results can be directly added for the final product. So if  $a$ ,  $b$ ,  $c$ , and  $t$  are unsigned registers,  $0 \leq b \leq 63$ , and  $a \times b < 2^{36}$ , we compute  $c = a \times b$  in four instructions:

```

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

```

Short multiplication correctly detects the overrange condition, in which case the

**Table 7.6:** Other ALU instructions and their implementations.

opcode	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
ML	ml	id	add	t	asr.uu	0	multiply low (1)
MH	mh	rtl	add	t	asr.us	0	multiply high
MHNS	mh	id	add	t	asr.uu	0	multiply high no shift
DSL	dsl	rtl	add	dsl	logic	T	double shift left (2)
PIT	boo	id	pit	t	logic	0	permute inside tribbles
PAT	boo	pat	add	t	logic	0	permute across tribbles
PAIT	boo	pat	pit	t	logic	0	permute across, inside tribbles
MIX	mix	mix	mix	t	logic	0	mix (3)
XIM	xim	xim	xim	t	logic	1	undo mix
CRF	boo	id	kil	1	clear.r	1	clear R(ange) flag
HAM2	boo	ham	ham	t	logic	1	second instr. of popcount macro
NUDGE	nud	id	mux	2r	logic	1	nudge (4)
SWIZ	boo	swz	add	t	logic	0	swizzle
TXOR	boo	txo	add	t	logic	0	transposing XOR

- (1) ML, MH, and MHNS do short multiplication across 6-bit tribbles.  
(2) Reduces the number of instructions needed for 36-bit multiplication.  
(3) Substitution-permutation network derived from  $\sqrt{2}$ .  
(4) Replaces 0 to 35 rightmost bits of L with the same number of bits from R.

result is of little value and might best be considered undefined. Short multiplication when b is constant only needs three instructions, because the compiler, assembler, or programmer can provide the initial replication.

Long multiplication can be emulated in software. For 36-bit unsigned integers with 72-bit results, 47 instructions are needed, not including two probable **CALL** and **RETURN** instructions for code reuse. An assembly subroutine showing long multiplication appears as listing 7.2. Unfamiliar language elements to the reader may include registers named by the programmer and declared to be unsigned, infix notation of alphabetic opcodes such as **SWIZ** and **MHNS**, an ‘o’ suffix for octal constants, and a ++ operator for the **AC** (add with carry) instruction.

Two opcodes in listing 7.2 that are not used for short multiplication are **MHNS**

```

; Multiply d:c = a * b

unsigned a b      ; inputs
unsigned c d      ; outputs
unsigned m t      ; scratch

t = b swiz 050505050505'o
c = a mhns t
d = 0 dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0

t = b swiz 040404040404'o
m = a mhns t
c = c + m
d = d dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0                                ; continued after column 1

t = b swiz 030303030303'o
m = a mhns t
c = c + m
d = d dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0

t = b swiz 020202020202'o
m = a mhns t
c = c + m
d = d dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0

t = b swiz 010101010101'o
m = a mhns t
c = c + m
d = d dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0

t = b swiz 000000000000'o
m = a mhns t
c = c + m
d = d dsl c
c = c lsl 060606060606'o
m = a ml t
c = c + m
d = d ++ 0

```

**Listing 7.2:** Assembler code for 36-bit multiplication. 47 instructions are executed.

**Table 7.7:** Frequently-used swizzle operations for  $\beta$ .swz.

octal	description
00	6 copies of tribble 0
01	6 copies of tribble 1
02	6 copies of tribble 2
03	6 copies of tribble 3
04	6 copies of tribble 4
05	6 copies of tribble 5
06–63	reserved

(“multiply high no shift”), which multiplies tribbles pairwise but without shifting the result 6 bits to the left, and DSL (“double shift left”). The DSL instruction is complex, because it effectively adds the T(emporal) flag to the left operand before shifting left six bits. The six least significant bits of the result are copied from the high tribble of the right operand. This instruction was designed specifically to combine two operations from an older multiplication subroutine that took 53 instructions instead of 47. One intricacy of DSL is that the shift is completed by  $\beta$  before  $\gamma$  can finish adding the T(emporal) flag, so  $\theta$  offers a special rotated output to align the increment to the  $2^6$  place value.

Also in the listing is SWIZ, which selects a tribble from the left operand and replicates it six times. The SWIZ opcode is a general call to any of the swizzles of table 7.7, where the first six operations happen to copy the six tribbles by position.

Except that we might not have CPU microcode that executes multiplication as a single instruction, this performance is not far out of line from the original Intel 80486, which took between 18 and 42 cycles at 16 MHz to multiply 32 bits depending on the operands. This minimum-speed i486 could do 32-bit multiplications at 380 952 to 888 889 per second. If my architecture can sustain 16 MIPS, it can do 36-bit multiplications at 326 530 per second, including overhead for CALL and RETURN.<sup>3</sup> For

---

<sup>3</sup>I here compared an i486 with a 16 MHz clock to my architecture with an 80 MHz clock. The illustration is to compare my architecture’s throughput to the top of the line as of 1989.

short multiplication, my architecture beats the minimum-speed i486 with four million per second rather than 888 889. (At the time of its discontinuation in 2007, the i486 could be clocked at 100 MHz and do 5 555 556 short multiplications per second.)

### 7.2.6 NUDGE instruction

The **NUDGE** instruction of table 7.6 adjusts the rightmost portion of the left operand, which could be a pointer, to conform to a supplied template in the right operand. This template consists of a start bit, indicating the number of bits to replace, and the replacement bits themselves. For example, 0111010 **NUDGE** 0010110 replaces the rightmost four bits to give the result 0110110 in one instruction. Without **NUDGE**, this process would require two instructions and would happen as (0111010 **AND** 0001111) **OR** 0000110.

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

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

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



```

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

```

With this format, `*parent` can directly access the `left` or `right` field of the parent node instead of the parent node as a whole, allowing us to null the dereferenced word within the parent without investigating which child `n` used to be. The difficulty is that `parent` is only useful for deletions, because it doesn't know where within the parent node it's pointing. The **NUDGE** opcode eliminates this problem for structures with power-of-two sizes. If this C program can access the **NUDGE** opcode, we can in one instruction locate any member of the parent by nudging the parent pointer's alignment within its known power of two. More generally, we can compute the address of a structure member in one instruction from the address of another member—even if we haven't kept track of which member the address calculation will start from.

**NUDGE** isn't new, in the sense the same operation can already be done at half the speed by using **AND** and **OR**. Moreover, the alignment behavior an implementer chooses for memory allocation on an SRAM CPU may differ from conventional processors. GNU `malloc`'s alignments to multiples of 8 or 16 are in fact just multiples of two machine words, because many traditional CPUs address individual bytes. But byte addressing and need for aligned memory boundaries might not apply to an SRAM-based CPU. Similarly, the cache lines we ordinarily try to consider are unlikely to be present on SRAM machines. So **NUDGE** might not become a big trend, but it does serve as an example of overloading a carry decision mechanism for other uses. It's also a rare example of left-to-right carry propagation, where  $\alpha$  and  $\theta$  need to scan the template from left to right to locate the leading 1 bit.

## 7.2.7 Bit permute opcodes

Table 7.6 shows how bit permutation opcodes PIT, PAT, and PAIT are provided in the  $\beta$  and  $\gamma$  layers.<sup>4</sup> Most software doesn't need to permute bits, except for ordinary shifts and occasional endianness changes, and only newer instruction sets offer much beyond these. So when an arbitrary rearrangement of a word is needed, historically a lot of code and clock cycles are expended doing this rearrangement. A swap of even two arbitrary bits requires at least five instructions: isolate the bits to swap, mirror all bits in that word,<sup>5</sup> shift to realign correctly, AND the bits out of the original word, and OR the replacements in. An arbitrary permutation of a 36-bit word would take up to 108 instructions, most of which are repetitions of (shift, AND, OR).

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

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

At present, I offer a hierarchy of claims about how many instructions are needed

---

<sup>4</sup>A reviewer who found this section interesting asked me to clarify how much of it is original. I am not knowledgeable enough in combinatorics to claim that any new mathematical knowledge is offered. What I can say is that I derived everything in this section independently.

<sup>5</sup>Many architectures do not offer a MIR instruction or macro, although mine does.

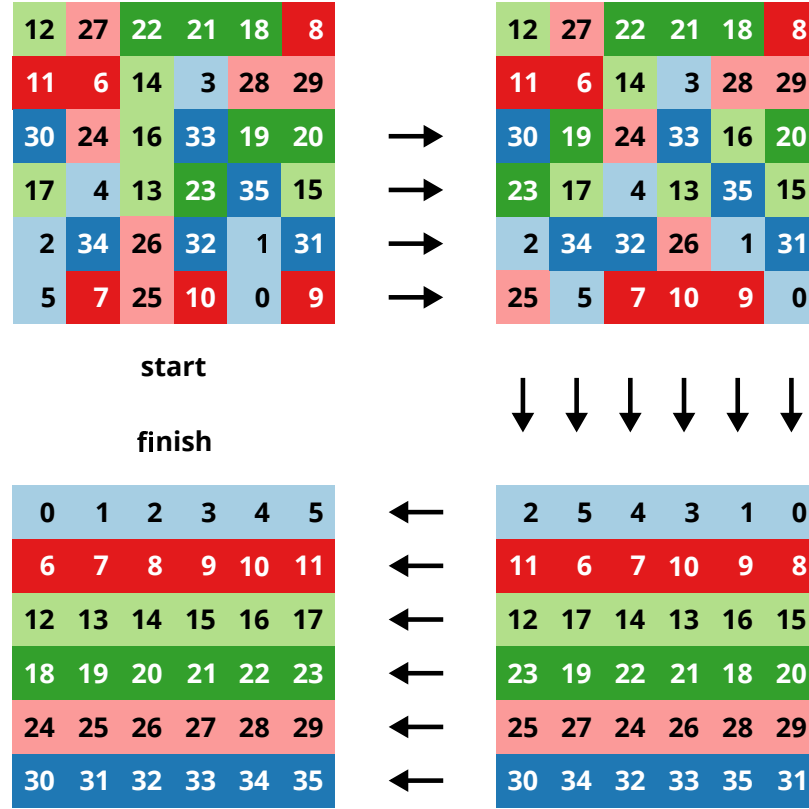
for 36-bit permutations. The operands to the instructions that produce these permutations are tedious and ordinarily not suitable for manual computation. Instead, an algorithm converts the desired permutation into the sequence of instructions which accomplishes it. I can offer you algorithms that succeed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

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

PIT is implemented in the  $\gamma$  layer. PAT uses the  $\beta$  layer and is simply a transposition of PIT. It's helpful to regard a 36-bit word as not a row of bits, but as a  $6 \times 6$  matrix as shown in figure 5.3 (p. 77). Any corner may be used as the origin; the point is that each row of the matrix represents one tribble of the machine word. Figure 7.1 shows how such a matrix can be operated on. The matrix is the left argument to PIT, which operates on the rows of the matrix, or PAT, which operates on the columns. The operation wanted for each row or each column is taken from the tribbles of the right argument, meaning we can invoke six different permutations on the six rows in one PIT, or six different permutations on the six columns in one PAT. So the good news is: a lot of things can happen at once simultaneously. The first part of the bad news is: we can only move within rows or within columns during one instruction.

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

Figure 7.1 shows how six bit permutation instructions combine to produce arbitrary permutations of full words. The three steps are indicated by horizontal arrows



**Figure 7.1:** Decomposition of a random 36-bit permutation.

(two PITs), and vertical arrows (two PATs), with the colored matrices showing the states before and after each step. A randomly-selected permutation appears in the upper left “start” matrix. The numbers in each box show that box’s eventual destination as its content—a zero or one—moves through each step. Every six consecutive numbers is given a different background color, allowing visual differentiation of the subword each bit will end within.

In the first transition, drawn as four arrows, each row is rearranged such that no color appears in the same column twice, or equivalently, every column contains every color. This guarantees that the second transition, drawn with vertical arrows, will be able to move every bit to its destination row (tribble) without conflict. The algorithm for the first transition works one row at a time without backtracking. The first row never requires adjustment. The second row of this example had no color

conflict with the first row. The third row contained three color conflicts, requiring relocation of the 16, 19, and 24 bits. The relocation algorithm is simply to try all 720 permutations, and use the first permutation that does not conflict with any preceding row. Eventually all six rows are decided, and the six permutations chosen are encoded into two PIT instructions.

A question appears as to how we know, for each tribble, which two permutations selected from the 64 implemented are needed for any of the 720 possible permutations. We know either by trial and error, with a maximum of 4 096 trials for each tribble, or from a  $720 \times 12$ -bit table in the compiler or assembler.

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

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

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

do we have a five-instruction method (although I have not proven that it will always succeed), but it even has a simpler algorithm to determine the instructions needed.

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

If we can find 4-instruction random permutations of 36 bits, we must strictly alternate PITs with PATs, where we are bounded by  $2 \times 64^{24} \approx 4.5 \times 10^{43}$ . This is sufficiently above the number of 36-object permutations to offer, on average, 120 different ways of finding each. But this does not guarantee that every permutation is reachable, let alone offer a fast algorithm for producing the four instructions.

We do have an algorithm to try to find four instructions. It's applied twice for completeness: PIT PAT PIT PAT and PAT PIT PAT PIT in either order. There is one transition for each instruction, so if figure 7.1 were redrawn for this operation, five matrices would be shown. The first transition is *carte blanche*: arbitrarily select any of the  $64^6 = 2^{36}$  PIT or PAT arguments. In practice, we would loop through all of them until the algorithm succeeds. The second, third, and fourth transitions are identical to the three transitions on figure 7.1, except we strictly limit to 64 alternatives per tribble at each step. Time complexity goes like this: the first transition always succeeds for its  $2^{36}$  variants, and the second transition succeeds an average of 820 times for each of the first transitions. This estimate comes from simulation with 10 000 trials. So this algorithm enters the third transition a total of  $2 \times 2^{36} \times 820$  times, which is good because the combined odds of success in the third and fourth transitions are

presumably  $(720 \div 64)^{12}$  to one against us, less than one in four trillion. So if we run the entire experiment to completion for many random permutations, on average we would expect to find around 27 sequences of four instructions that can produce each permutation.<sup>6</sup> This isn't a guarantee there aren't permutations which no series of four instructions can reach, but the chances of succeeding often at least appear pretty good. But the problem is CPU time: if you tell an assembler or compiler "I want the permutation \_\_\_\_\_, figure it out and do it in four instructions," it has to come out of the second transition successfully to try the third and fourth about four trillion times. This computation would take decades of CPU time on average to find the first working sequence of four instructions. Although the search is easily parallelized, we will ordinarily settle for five instructions.

The PAIT opcode combines PAT followed by PIT with the same right operand. Its most foreseeable use is to reverse the order of a 36-bit word (the MIR macro, which is PAIT with a right operand of 131313131313'o). There may be other uses.

### **Which 64 of the 720 permutations of six bits should be included?**

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

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

---

<sup>6</sup>This expected value of 27, which relies on multiple assumptions of uniformity, is within an order of magnitude of the preceding paragraph's expected value of 120, which is more dependable.

The permutations chosen for this ALU appear in table 7.8. From the above list, only criteria 1 and 3 were used when these permutations were selected. The middle requirement was left to chance, although there is more to say about that shortly. The order of items in the table is roughly the order I dreamed them up. Bit ordering in this table is not the conventional 543210 that would be used for arithmetic, but 012345 in order to make these permutations more spatially understandable by programmers. So the identity permutation is 012345, and exactly reversing the bits is 543210. Table 7.8 was produced iteratively: I would think of permutations to add to the set, and then I would use a greedy algorithm to add as few permutations as it could to make the set “720 complete.” The minimum size needed for this completeness grew slowly. The more permutations I added to the subset, the fewer the algorithm needed to add to reach 720 by composing two permutations that were already in the set. So I might find I had 20 left to choose, and after choosing 10, I’d find there were still 15 left to choose. In the end I was able to choose 58 of the permutations by the seat of my pants, and the algorithm only needed to choose six. But I couldn’t have chosen these last six without automation.

If someone had to design an unusually crowded ALU, possibly using the same function select inputs for permutations and swizzles, perhaps there wouldn’t be room to include 64 permutations. How small a subset would still work? As  $26 + 26^2 = 702$  and will not cover 720 permutations of six bits, it’s clear that a minimal set will have at least 27 elements. I haven’t succeeded at computing a set as small as 27 elements, although I spent more than 18 months of CPU time trying. But there wasn’t a lot of human effort expended. The algorithm I wrote had a fast implementation in C, but a better-considered algorithm might have a faster implementation.

During the search, I found four subsets of 30 permutations that can reach all 720 within two instructions. The first of these appeared in just hours, before I was even looking at the program output consistently before clearing the screen. I was stunned when I saw it, because I knew by that time that a set that small is very difficult



**Table 7.8:** Directly-available tribble permutations.

oct.	perm.	derivation	oct.	perm.	derivation
00	012345	identity	40	031524	count out of circle by 3s
01	123450	rotated identity	41	142035	count out of circle by 3s
02	234501	rotated identity	42	253140	count out of circle by 3s
03	345012	rotated identity	43	304251	count out of circle by 3s
04	450123	rotated identity	44	415302	count out of circle by 3s
05	501234	rotated identity	45	520413	count out of circle by 3s
06	432105	reflected 05	46	035124	count out by 3s backward
07	321054	reflected 04	47	140253	count out by 3s backward
10	210543	reflected 03	50	251304	count out by 3s backward
11	105432	reflected 02	51	302415	count out by 3s backward
12	054321	reflected 01	52	413520	count out by 3s backward
13	543210	reflected identity	53	524031	count out by 3s backward
14	102345	pair swap	54	102354	2-pair rotation
15	210345	pair swap	55	315042	2-pair rotation
16	312045	pair swap	56	542310	2-pair rotation
17	412305	pair swap	57	513240	2-pair rotation
20	512340	pair swap	60	021435	3-pair rotation
21	021345	pair swap	61	034125	3-pair rotation
22	032145	pair swap	62	043215	3-pair rotation
23	042315	pair swap	63	103254	3-pair rotation
24	052341	pair swap	64	240513	3-pair rotation
25	013245	pair swap	65	453201	3-pair rotation
26	014325	pair swap	66	521430	3-pair rotation
27	015342	pair swap	67	534120	3-pair rotation
30	012435	pair swap	70	120534	symmetric half rotation
31	012543	pair swap	71	201453	symmetric half rotation
32	012354	pair swap	72	034512	algorithmically selected
33	452301	movement in pairs	73	035421	algorithmically selected
34	014523	movement in pairs	74	105243	algorithmically selected
35	230145	movement in pairs	75	130542	algorithmically selected
36	024135	$2 \times 3$ , $3 \times 2$ transposes	76	254310	algorithmically selected
37	031425	$2 \times 3$ , $3 \times 2$ transposes	77	510432	algorithmically selected

**Table 7.9:** Compact set of tribble permutations for two instructions.

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

to compute, and the search of that time ran 275 times slower than it would after I ported it from Python to C. Table 7.9 shows this set of 30 permutations.

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

### 7.2.8 Mix opcodes

Table 7.6 (p. 112) also includes the MIX and XIM opcodes for S-box operations. MIX maximizes the ALU's capacity to use S-boxes quickly for hashing and pseudorandom number generation. This ALU contains  $64 \times 18 = 1,152$  different S-boxes; that is, 64 S-boxes for each RAM in the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers. They are applied to the left operand by the MIX or XIM instruction, and selected by the tribbles of the right operand. This implements a key-dependent mixing or unmixing operation for 36 bits in one instruction. The mixing may be iterated for a number of instructions with different key material supplied as the right argument.

MIX is invertible for the left operand only. For any key  $k$  and inputs  $a$  and  $b$ ,

$$a = b \iff a \text{ MIX } k = b \text{ MIX } k. \quad (7.1)$$

The inverse of the MIX opcode is XIM . I pronounce this as *zim*, because *ksim* has awkward phonemes. XIM has the property

$$(a \text{ MIX } k) \text{ XIM } k = (a \text{ XIM } k) \text{ MIX } k = a. \quad (7.2)$$

The S-boxes for XIM derive easily by inverting the MIX S-boxes, and swapping between the  $\alpha$  and  $\gamma$  layers. If a multiple-word key is used for a sequence of MIX instructions, these words must be provided correctly in reverse order to XIM to recover the original input.

Every bit of the 36-bit output from MIX and XIM depends on every input bit. When exactly one input bit changes to the left operand and the key is held constant, an average of 15.375 out of 36 bits change in the output. When exactly one input bit changes to the key and the left operand is held constant, an average of 16.474 bits out of 36 change in the output. These figures come from 14 300 000 trials of the MIX instruction for the two cases using `/dev/urandom` as source material. No measurements were made with the unmix instruction XIM .

When two MIX instructions are used sequentially with the same right operand, the average number of bits changed as a result of a single input bit changing improves to approximately 17.9997 bits out of 36, which is very near to the desired measurement of 18 bits. This figure comes from 398 000 000 trials of the MIX instruction. No measurements were made with the unmix instruction XIM .

### **Application: Hash function for associative arrays**

Either MIX or XIM can be used as a very fast keyed hash function for associative arrays, using just one instruction per word hashed. Listing 7.3 shows a code snippet

```

state = secret.key xim word.0
state = state xim word.1
state = state xim word.2
output = state xim word.3

```

**Listing 7.3:** Using the XIM instruction to hash a four-word object.

to hash a block of four words. The `xim` token here is the XIM opcode, which is written infix. There would be no problem using `mix` instead, although the computed hashes would differ from `xim`. The period in the register name `secret.key` is legal syntax. Because the assembly language does not have any structures at this time, periods are permitted in identifiers so that variables can be named into structure-like groups.

To evade hash flooding attacks on machines that may encounter an adversary, the key should be secret and certainly not hardcoded. I did not further assess the security of this hash function, and I do not warrant that it is free of possible exploits.

I did not evaluate this function’s speed against popular contemporary hash functions such as SipHash [Aumasson12] or xxHash [Collet21]. Such evaluations are complex on account of differences in target architectures. For example, one of the processors discussed in the SipHash paper has three ALUs, and benchmark results are given in terms of CPU cycles rather than instructions. Although the xxHash family is not specified for 36 bits at this time, its inner loop apparently uses two multiplications, an add, and a rotation for each word hashed. This seems to me like four instructions,<sup>7</sup> in contrast with one instruction for a MIX or XIM hash.

I did not evaluate the function of listing 7.3’s collision or distribution properties, although I am curious as to how they would measure against other hash functions. The SMHasher test suite [Appleby16], supported by comparative data from other functions, may be helpful for such an evaluation.

---

<sup>7</sup>Most hash functions that are written for speed avoid multiplication. Here also is an architectural difference making comparative assessments more difficult: my ALU does not offer 36-bit multiplication as a standard instruction.

```

        counter = seed1      ; must be nonzero
        output = seed2      ; can be anything
loop:
        counter = lfsr counter
        output = output mix counter
        ...
        jump loop

```

**Listing 7.4:** This two-instruction PRNG withstands all Dieharder tests.

### Application: Pseudorandom number generator

History has shown there are many wrong ways to design pseudorandom number generators (PRNGs). Warning was given forty years ago in [Knuth81]. The GNU Scientific Library (GSL) contains a large collection of historic PRNGs, of which many are poor [Galassi21]. A simple way for someone who has computing skills but perhaps not mathematical skills to weed out defective PRNGs is to use a statistical testing package for PRNGs such as Dieharder [Brown20]. One of the Dieharder’s best features is that all of the GSL PRNGs are compiled into it. Not only can the user evaluate the PRNGs using the Dieharder tests, but the user can also evaluate the Dieharder tests using the GSL PRNGs. I ran a full set back in 2008, and I thought the results were very interesting.

Using just Dieharder to test even a few PRNGs that use my ALU’s `MIX` and `XIM` instructions would make for a lengthy paper. Instead, I present a single PRNG in listing 7.4 that Dieharder gives a clean bill of health. I do not have the mathematical knowledge to begin to consider the suitability of this PRNG, but I will mention a couple of points.

Listing 7.4 shows a PRNG that uses just two instructions per 36-bit word output. The PRNG has 72 bits of state that can be initialized via `seed1` and `seed2`. 36 bits of this state is directly observable in the output stream, so if someone, perhaps an adversary, wanted to predict this generator’s future output on the basis of output that

has already appeared, at most  $2^{36} - 1$  trials would be needed to fully “crack” this generator. So although this generator may be adequate for simulations and casual use, it is not cryptographically secure.

An eight-tap linear feedback shift register (LFSR) is implemented by a single-instruction assembler macro that expands to stacked unary operation `su.lfsr` (section 7.3.7). This LFSR has two cycles: the trivial cycle of all zeros with period 1, and another cycle with all the other integers and period  $2^{36} - 1$ . For the PRNG to work, it’s critical that the LFSR’s seed be nonzero. If this condition is met, the right operand to `MIX` will have period  $2^{36} - 1$ , and the left operand to `MIX` will have the same period as the PRNG as a whole. Because every output bit of `MIX` depends on every input bit to both the left and right operands, the period of this generator will be  $2^{36} - 1$  at worst. The exact period will be a function of the two seeds, and will not exceed  $2^{72} - 2^{36}$  on account of the number of possible internal states.

The left operand to the `MIX` instruction is fed by the PRNG’s own output. Had the left operand been a constant, the PRNG’s period would have been fixed at  $2^{36} - 1$ , and there would be no duplicates in the output words within a period. Absence of duplicates is not random behavior. By using the output stream to give the `MIX` instruction more input than the counter alone, output words will tend to repeat more in the manner of a random variable.

Listing 7.5 shows the initial output of an October 2022 test, where the virtual machine of [Abel22b] simulates the PRNG of listing 7.4 and pipes the output to `Dieharder`. The PRNG seed values were taken from `/dev/urandom`. The Bash command line was:

```
$ ./vm -s u | dieharder -g 200 -a
```

The `Dieharder` documentation and virtual machine source code can explain the command line options.

Although `Dieharder` catches many known bad PRNGs, its capabilities are infinitesimal compared to the variety of ways a PRNG may differ from a true random

```

#=====#
#      dieharder version 3.31.1 Copyright 2003 Robert G. Brown      #
#=====#
  rng_name      |rands/second|   Seed   |
stdin_input_raw|  2.45e+06  |1455060622|
#=====#
      test_name  |ntup| tsamples |psamples|  p-value |Assessment
#=====#
  diehard_birthdays|  0|      100|      100|0.06637487| PASSED
    diehard_operm5|  0|    1000000|      100|0.66165512| PASSED
  diehard_rank_32x32|  0|      40000|      100|0.09510301| PASSED
    diehard_rank_6x8|  0|      100000|      100|0.97071373| PASSED
  diehard_bitstream|  0|    2097152|      100|0.30879029| PASSED
    diehard_opso|  0|    2097152|      100|0.44448456| PASSED
    diehard_oqso|  0|    2097152|      100|0.84436191| PASSED
    diehard_dna|  0|    2097152|      100|0.54635854| PASSED
  diehard_count_1s_str|  0|      256000|      100|0.06723551| PASSED
  diehard_count_1s_byt|  0|      256000|      100|0.59711400| PASSED
  diehard_parking_lot|  0|       12000|      100|0.56023159| PASSED
    diehard_2dsphere|  2|        8000|      100|0.61233368| PASSED
    diehard_3dsphere|  3|        4000|      100|0.16715580| PASSED
    diehard_squeeze|  0|      100000|      100|0.62235389| PASSED
    diehard_sums|  0|        100|      100|0.18303812| PASSED
    diehard_runs|  0|      100000|      100|0.77239037| PASSED
    diehard_runs|  0|      100000|      100|0.94154880| PASSED
    diehard_craps|  0|      200000|      100|0.56317191| PASSED
    diehard_craps|  0|      200000|      100|0.46969494| PASSED
  marsaglia_tsang_gcd|  0|    10000000|      100|0.78545646| PASSED
  marsaglia_tsang_gcd|  0|    10000000|      100|0.36912472| PASSED
    sts_monobit|  1|      100000|      100|0.71230580| PASSED
      sts_runs|  2|      100000|      100|0.98030602| PASSED
      sts_serial|  1|      100000|      100|0.33726371| PASSED
      sts_serial|  2|      100000|      100|0.54329962| PASSED
      sts_serial|  3|      100000|      100|0.59017244| PASSED
      sts_serial|  3|      100000|      100|0.50313766| PASSED
      sts_serial|  4|      100000|      100|0.25174796| PASSED
      sts_serial|  4|      100000|      100|0.59153234| PASSED
      sts_serial|  5|      100000|      100|0.03029913| PASSED
      sts_serial|  5|      100000|      100|0.66420369| PASSED
      sts_serial|  6|      100000|      100|0.04175049| PASSED
      sts_serial|  6|      100000|      100|0.19005336| PASSED

```

**Listing 7.5:** First portion of Dieharder output evaluating the PRNG of listing 7.4.

variable. If Dieharder were perfect, its tests would fail all PRNGs (identify them as non-random), and pass all true random number generators. At best, the pseudorandom number generator of listing 7.4 strikes an impressive ratio, compared with other PRNGs, of output quality to number of instructions per output word, and does so in a completely transparent hardware implementation. But more study by qualified investigators should be done to further validate or if necessary deprecate this PRNG.

The eight taps used by the LFSR instruction are special, in that identical taps can implement 72-, 108-, and 104-bit LFSRs using the XPOLY instruction. Listing 7.6 contains background as to how the taps were selected.

### **Application: Round function for a 36-bit cipher**

MIX and XIM have potential for use in cryptography, provided that experts specify and validate the ciphers. This is a specialization I am not qualified in, but here are a couple of problems with these operations. First, I have specified S-blocks that are not optimized in any manner against differential cryptanalysis. This is somewhat on purpose: optimizing for a specific attack could break a cryptosystem with respect to some other, unanticipated attack. Most efforts to design attacks against block ciphers and S-blocks are done in secret. So although the S-blocks specified are possibly terrible—no one has checked—with respect to differential cryptanalysis, I can't charge forward as if differential cryptanalysis is the only concern. In any event, more work is called for, because differential cryptanalysis safeguards are considered mandatory for new cipher submissions to standards bodies.

The other problem with MIX and XIM for secrecy is the 36-bit block size is too small and cannot be enlarged easily. The key would need to be changed with just about every packet sent or received due to the potential for birthday attacks, and collisions in real-world protocols have been demonstrated for considerably larger block ciphers [Bhargavan16]. Although special cipher modes have been designed to expand the birthday bound [Iwata06], 36-bit blocks are still too small for these new modes



```

/*
The minicomputer firmware offers fast, well-mixed pseudorandom
number generation via a fast linear feedback shift register (LFSR)
followed by a substitution-permutation network (SPN). If a period
of 2**36 is long enough, it takes just two instructions to
generate a "good" random number. But we need a polynomial to
define the LFSR taps, and here it is.

The following polynomials are all primitive and use the same taps
in the 36 most sig. bits. Therefore, we conceivably can use the
same 8 tap positions for LFSRs of 36, 72, 108, 144. This is the
ONLY tap set suitable for all four sizes that has 8 or fewer taps.
It cannot support 180, 216, 252, or 288 bits.

There is no penalty for using more than 8 taps, but this requires
CPU time and potentially a Mathematica license over that time to
search. Finding this 8-tap solution already took 90 minutes of CPU
time, and further solutions may require far more time.


$$x^{36} + x^{31} + x^{13} + x^7 + x^6 + x^5 + x^3 + x^2 + 1$$


$$x^{72} + x^{67} + x^{49} + x^{43} + x^{42} + x^{41} + x^{39} + x^{38} + 1$$


$$x^{108} + x^{103} + x^{85} + x^{79} + x^{78} + x^{77} + x^{75} + x^{74} + 1$$


$$x^{144} + x^{139} + x^{121} + x^{115} + x^{114} + x^{113} + x^{111} + x^{110} + 1$$


const long lfsr_taps =
    1L << 35 | 1L << 30 | 1L << 12 | 1L << 6
    | 1L << 5 | 1L << 4 | 1L << 2 | 1L << 1;
*/
#define lfsr_taps 0x840001076L

```

**Listing 7.6:** An explanation of the LFSR and XPOLY polynomials.

to be useful. But implementing a cipher for larger blocks requires that we choose between many more instructions per word encrypted, or a much larger number of RAMs in the ALU.

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

### **How S-boxes for MIX were selected**

Listing 7.7 shows a Python 3 specification for the MIX opcode’s suggested S-boxes. The boxes are filled from a so-called “nothing up my sleeve number,” giving a simple, transparent process for computing S-boxes. The rationale is that by using an algorithm that takes almost no input from me, I would have great difficulty trying to build a backdoor into the S-boxes produced by this algorithm. I opted to use an easy-to-compute irrational number, rather than requiring storage for a long number, or having people look up a number online or compute it with some utility.

The square root of one half  $\approx 0.7071$  is possibly the easiest irrational of all to write code to compute, and it converges quickly using Newton’s method. Listing 7.7 does this using Python’s built-in arithmetic for large integers. The program takes only a few moments to compute a 341 000-bit approximation of  $\sqrt{2} \div 2$ . This turns out to be enough bits to fill all of the S-boxes without reusing any data.

The  $\sqrt{2} \div 2$  bits are regarded as a fixed-point real number, where bit 341 000 would have place value 1. Initially, there is a 0 in place value 1, because only the bits through 340 999 are filled, representing a real number  $r$  such that  $0 \leq r < 1$ . Information is extracted from these bits by presuming  $r$  is a uniformly-distributed random variable. If a random integer between (for example) 0 and 4 inclusive is needed, the integer representation of  $r$  can be multiplied by 5. The integer part of the product will be in bits 341 000–341 002, and will be 0, 1, 2, 3, or 4. The fractional

```

#!/usr/bin/python3

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

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

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

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

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

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

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

```

**Listing 7.7:** Python 3 specification of S-boxes for the MIX and XIM opcodes.

part, still in bits 0–340 999, becomes the uniformly-distributed value of  $r$  for the next random integer extraction.

The program generates 6-bit S-boxes by shuffling the numbers 0 through 63 inclusive, using successive multiplications to pick from 64 remaining numbers, 63 remaining numbers, all the way down to the last remaining number. The precision of the arithmetic on  $r$ , 341 000 bits, needs to be adequate to uniquely describe any contents of the S-boxes that must be filled. Each  $\alpha$ ,  $\beta$ , and  $\gamma$  SRAM contains 64 S-boxes, and there are a total of 18 of these RAMs, so 1 152 S-boxes are needed. For each S-box, there are  $64!$  different contents possible, so a minimum of  $\lceil 1\,152 \times \log_2 64! \rceil = 340\,987$  bits is sufficient to fill all of them. The hardcoded 341 000 bits in listing 7.7 exceeds this computed minimum. If there are enough bits, the same set of S-boxes will be output regardless of how many bits were used for the computation. If there are not enough bits, the S-boxes obtained will change according to the number of bits supplied.

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

## 7.2.9 Simple unary instructions

The ALU opcodes of sections 7.2.1–7.2.8 all implement *binary operators*, meaning functions with two inputs. For opcodes that return a result, the assembler syntax places either the opcode or operator symbol between the left and right arguments. Unsurprisingly, the ALU also has opcodes for implementing *unary operators*, single-input functions that depend solely on the left operand.

**Table 7.10:** Unary instructions and their implementations.

opcode	$\alpha$	$\beta$	$\gamma$	$\theta$	$\zeta$	T	description
UN.A	uny	id	add	t	logic	0	simple unary, alpha layer
UN.B	boo	uny	add	t	logic	0	simple unary, beta layer
UN.G	boo	id	uny	t	logic	0	simple unary, gamma layer
STUN.A	stu	stu	stu	add	add.u	1	stacked unary support
STUN.B	stu	stu	stu	gt	logic	1	stacked unary support
STUN.C	stu	stu	stu	t	asl.uu	0	stacked unary support
STUN.D	stu	stu	stu	2r	logic	1	stacked unary support
STUN.E	stu	stu	stu	2l	logic	0	stacked unary support
STUN.F	stu	stu	stu	rev	t.adj	1	stacked unary support
STUN.G	stu	stu	stu	t	t.adj	0	stacked unary support
STUN.H	stu	stu	stu	t	t.adj	T	stacked unary support
STUN.I	stu	stu	stu	add	t.adj	1	stacked unary support

Rather than ignore the right operand for unary operators, my architecture uses the right operand to select the unary function being implemented. Any “slot,” or combination of function select bits, offers enough SRAM address space to implement either one binary operator or 64 unary operators (figure 5.4).<sup>8</sup>

In a *simple unary instruction*, sometimes shortened to *unary instruction*, only one layer— $\alpha$ ,  $\beta$ , or  $\gamma$ —participates in the computation. The non-participating layers pass their inputs through unchanged. Because whichever of the three layers is active operates on disjoint tribbles (transposed in  $\beta$ ’s case), the computation is very simple and is threaded across the tribbles. The simple unary instructions are implemented by the first three opcodes of table 7.10: UN.A, UN.B, and U.G for the  $\alpha$ ,  $\beta$ , and  $\gamma$  versions respectively.

The available operations are in table 7.16 (p. 161), which still has room for expansion. The right operand’s six-bit specifier as to which unary function to implement must be replicated across the right operand. For example, to reverse the bits within each tribble of a word, operation 22’o from table 7.16 is applied to all six tribbles:

---

<sup>8</sup>Another interpretation is that a binary operator may use its right operand to select among and emulate 64 unary operators.

```

word = 111100_010101_000010_111111_011001_011111'b
word' = word un.a 222222222222'o
; word' is now
;      001111_101010_010000_111111_100110_111110'b

```

Because the right operand's tribbles are isolated to one participating SRAM each, it is possible to mix-and-match unary operations among the tribbles. For instance, the following code counts the number of zeros in the three most significant tribbles using operation 11'o, but counts the number of ones in the three least significant tribbles using operation 14'o:

```

word = 111100_010101_000010_111111_011001_011111'b
word'' = word un.a 111111141414'o
; word'' is now
;      000010_000011_000101_000110_000011_000101'b

```

With respect to real-program usefulness, simple unary operations are something of a solution looking for a problem. Rarely do programmers ever think about six-bit subwords, let alone seek to thread unary operations across them. Rather than being a design feature, simple unary operations are more like an architectural oddity that may find infrequent application as a result of being supported.

### 7.2.10 Stacked unary instructions

A *stacked unary instruction* employs the entire ALU to compute a function of the left operand, with the function itself selected by the right operand. Because of  $\beta$ 's transpositions,  $\theta$ 's summarizing capability, and other interconnections, stacked unary operations are not constrained to operate within tribbles. The result is that stacked unary instructions are much more capable than their simple unary counterparts, and would find use in many programs.

Stacked unary instructions engage the  $\alpha$ ,  $\beta$ , and  $\gamma$  layer simultaneously, with each layer having the ability to offer a different operation during the instruction. But there is one catch: the three layers share the same right operand, so if  $\alpha$  is

doing whatever it does for operation 25‘o,  $\beta$  and  $\gamma$  must be doing their versions of operation 25‘o. The ALU does not support  $\alpha$  doing operation 25‘o while  $\beta$  does operation 07‘o. Therefore, any stacked unary operation is actually written as a set of three unary operations, one each for the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers.

Table 7.10 lists nine stacked unary instructions, presently named STUN.A–STUN.I. Their names and implementations are likely not stable yet. Although the right operand supplies all information to  $\alpha$ ,  $\beta$ , and  $\gamma$  as to how they are to function for a particular operation, the incoming T(emporal) flag, flags RAM  $\zeta$ , and carry decision RAM  $\theta$  need their functioning to match the instruction’s objective. In practice, the assembler’s macro feature will generate stacked unary instructions with specific STUN opcodes (controlling  $\theta$ ,  $\zeta$ , and the incoming T(emporal) flag) and matching right operands (controlling  $\alpha$ ,  $\beta$ , and  $\gamma$ ). Nearly all of the macros in table 7.1 will expand to one or more stacked unary instructions.

Table 7.17 (p. 162) lists the stacked unary operations themselves—that is, their purposes and right operands. As is ordinarily done with simple unary operations, the octal code for the operation must be replicated to all six tribbles of its right operand.

## 7.3 ALU operations and their implementations

Section 7.2 described the ALU’s principal opcodes, and in six tables specified the operations and other control signals that implement each opcode. This section is about the operations themselves. As a reminder, the difference between an ALU *opcode* and an ALU *operation* is that opcodes are implemented by the ALU as a whole, but operations are implemented by individual SRAMs. Assembly language programs specify the sequence of opcodes a program uses, but it is the firmware that determines the operations that implement each opcode.

None of the ALU operation tables are near full at this time, assuming the ALU is constructed from  $256\text{Ki} \times 18$  SRAMs. For example, only 29 out of 64 available  $\alpha$

slots are specified, so space remains available to add more operations to the  $\alpha$  layer.

Although the ALU operations of this section are described in terms of calculations and steps, their SRAM implementations are simple table lookups. The complex operations described below aren't accomplished by manipulating a RAM to perform them, but using a C program to compute tables to distribute as firmware.

### 7.3.1 $\alpha$ layer operation

An alphabetical list of the  $\alpha$  (alpha) layer operations appears as table 7.11. Each of the six  $\alpha$  RAMs accepts a six-bit left operand L and six-bit right operand R, so the operations themselves are small and simple. Which operation to perform is determined by six operation select input bits. The  $\alpha$  layer portion of table 5.2 shows how these inputs map to SRAM address bits.

Each  $\alpha$  RAM sends a six-bit result to the transposed  $\beta$  layer. Each  $\alpha_i$  also sends two bits, nominally named  $p_i$  and  $c_i$  (propagate and carry), to the RAM  $\theta$ . The use of these two bits depends on the operation and is not always propagate and carry information. The most significant  $\alpha$  RAM,  $\alpha_5$ , sends five bits to RAM  $\zeta$ , nominally named *encoded range*. The use of these five bits depends on the operation, but ordinarily supports detection of arithmetic overflow. The  $\alpha$  layer portion of table 5.3 (p. 87) shows how these outputs map to SRAM address bits.

#### Additive operations: $\alpha$ .add, $\alpha$ .sub, $\alpha$ .rev

The additive operations do base-64 addition, subtraction, and reverse subtraction. The  $\alpha$ .add variants add in the customary manner. In addition, the propagate bit is set if the result is exactly 63, and the carry bit is set if the result is more than 63.

The  $\alpha$ .sub variants do a complement-and-add rather than a true subtract. After the right operand is replaced with its bitwise NOT, everything is exactly the same as for  $\alpha$ .add. Because this is a one's complement rather than a two's complement, the computed difference is off by one. The ALU corrects this by setting or inverting the



**Table 7.11:** ALU  $\alpha$  layer operations.

name	description
$\alpha$ .add.uu	add tribbles unsigned/unsigned
$\alpha$ .add.us	add tribbles unsigned/signed
$\alpha$ .add.su	add tribbles signed/unsigned
$\alpha$ .add.ss	add tribbles signed/signed
$\alpha$ .and	bitwise AND
$\alpha$ .asr	sign extension and zero detection for arithmetic shift right
$\alpha$ .cmp.uu	magnitude compare tribbles unsigned/unsigned
$\alpha$ .cmp.us	magnitude compare tribbles unsigned/signed
$\alpha$ .cmp.su	magnitude compare tribbles signed/unsigned
$\alpha$ .cmp.ss	magnitude compare tribbles signed/signed
$\alpha$ .dsl	replace 6 most significant bits of L with those of R
$\alpha$ .e35	check encroachment of bit 35 or 36 for arithmetic shift left
$\alpha$ .e36	check encroachment of bit 36 for arithmetic shift left
$\alpha$ .mh	multiply tribbles unsigned, high tribble of products
$\alpha$ .mix	$\alpha$ 's forward S-boxes
$\alpha$ .ml	multiply tribbles unsigned, low tribble of products
$\alpha$ .nud	help locate leftmost 1 bit for NUDGE instruction
$\alpha$ .or	bitwise OR
$\alpha$ .rev.uu	reverse subtract tribbles unsigned/unsigned
$\alpha$ .rev.us	reverse subtract tribbles unsigned/signed
$\alpha$ .rev.su	reverse subtract tribbles signed/unsigned
$\alpha$ .rev.ss	reverse subtract tribbles signed/signed
$\alpha$ .stu	"stacked unary" functions that engage $\alpha$ , $\beta$ , and $\gamma$ together
$\alpha$ .sub.uu	subtract tribbles unsigned/unsigned
$\alpha$ .sub.us	subtract tribbles unsigned/signed
$\alpha$ .sub.su	subtract tribbles signed/unsigned
$\alpha$ .sub.ss	subtract tribbles signed/signed
$\alpha$ .uny	R selects simple unary functions on 6-bit tribbles of L
$\alpha$ .xim	inverses of $\gamma$ 's forward S-boxes

incoming T(emporal) flag, which causes RAM  $\theta$  and the  $\gamma$  layer to correct the result. This T(emporal) flag specification can be seen in table 7.2 (p. 104).

The  $\alpha$ .rev variants work exactly like  $\alpha$ .sub, except that the left and right operands are swapped first.

RAM  $\alpha_5$  participates in overrange checking, but in order to do this, it needs to know whether its inputs are signed or unsigned. If L is signed, its most significant bit has place value  $-(2^{35})$ . Otherwise L is unsigned, and its most significant bit has place value  $2^{35}$ . The same is true for R. From the perspective of  $\alpha_5$ , which only sees the most significant tribble of L and R, the place values are  $-(2^5)$  and  $2^5$ .  $\alpha_5$  knows the signage of L and R, because the control decoder stratifies their operation by signage. These are indicated by .uu, .us, .su, and .ss suffixes in table 7.11 for  $\alpha$ .add,  $\alpha$ .sub, and  $\alpha$ .rev—there are a total of twelve operations. The first u or s indicates that L is unsigned or signed respectively, and the second u or s does the same for R.

How does the control decoder know which signage to request for an  $\alpha$  operation as a program is running? The answer is, the A, S, RS, AC, SC, or RSC opcodes are also stratified by signage, not only for the left and right operands, but for the result also. This can be seen in table 7.2.

How does the assembler know what signage to attach to additive opcodes? The answer is, the assembly language declares all registers names with their signage. Here is an example:

```
; Allocate and name two unsigned and one signed registers.
unsigned a c
signed b

; Below generates opcode A.usu, which the control decoder
; expands at runtime to alpha.add.us and zeta.add.u.
c = a + b
```

$\alpha_5$  participates in range checking, but it can't make the final decision as to whether overflow has occurred. It's missing two pieces of information. First, it's not known whether an incoming carry will arrive in the  $2^{30}$  place value, in which case  $\alpha_5$ 's

result would be one less than the true result. Second, in order to limit the number of operations the  $\alpha$  layer must implement (that is, in order to conserve memory), the  $\alpha$  layer does not know whether the destination register is unsigned or signed. Instead, RAM  $\zeta$  will determine if overflow occurred, using carry summary information from RAM  $\theta$  and an *overflow estimate* called *encoded range* from  $\alpha_5$ .

Although RAM  $\alpha_5$  truncates the sum it calculates to six bits, it also categorizes where the true sum lies prior to truncation. Assuming that *no carry* will arrive in the  $2^{30}$  place value, five scenarios exist that can be numbered from 0 through 4:

0	$\text{sum} < -32$	always underflows
1	$-32 \leq \text{sum} < 0$	underflows if result is unsigned
2	$0 \leq \text{sum} < 32$	never overflows or underflows
3	$32 \leq \text{sum} < 64$	overflows if result is signed
4	$64 \leq \text{sum}$	always overflows

The same information is needed under the assumption that a carry *will* arrive in the  $2^{30}$  place value. This causes the boundaries to move for the five scenarios, which are now numbered in increments of five:

0	$\text{sum} < -33$	always underflows
5	$-33 \leq \text{sum} < -1$	underflows if result is unsigned
10	$-1 \leq \text{sum} < 31$	never overflows or underflows
15	$31 \leq \text{sum} < 63$	overflows if result is signed
20	$63 \leq \text{sum}$	always overflows

By having  $\alpha_5$  evaluate both scenarios and add their representations together, this information RAM  $\zeta$  will need to determine if an addition or subtraction result does not fit in a particular destination register is encoded in a number between 0 and 24, which fits in five bits. This is the encoded range output of RAM  $\alpha_5$ .

An astute reader may note that the two range scenarios can be compressed from 25 cases to 9 cases by evaluating the sum relative to  $-32$ ,  $-32$ ,  $-1$ ,  $0$ ,  $32$ ,  $33$ ,  $63$ , and  $64$ . This would seem to allow the encoded range to fit in four bits instead of five. It turns out that the **MAX** and **MIN** instructions need five bits anyway, because the two scenarios to encode are not whether or not a carry occurred, but which operand is larger or smaller. Because the operands to **MAX** and **MIN** are independent, there will

be 25 cases to encode that cannot be reduced in number. So the encoded range field is kept at five bits, and the encoding mechanism is harmonized as much as possible between the additive opcodes and **MAX** and **MIN**.

### Bitwise boolean operations: $\alpha$ .and, $\alpha$ .or

The operations  $\alpha$ .and and  $\alpha$ .or respectively compute the bitwise AND and OR of the left and right operands. The result is sent to the transposed  $\beta$  layer. The propagate, carry, and encoded range outputs are not used.

### Compare operation: $\alpha$ .cmp

The  $\alpha$ .cmp operation is used by the **MIN** and **MAX** opcodes, but not by the **CMP** opcodes. The six-bit left and right operands are compared for each RAM. If the right operand is greater, the carry bit is set. If the left operand is greater, the propagate bit is set. If the operands are equal, neither bit is set.

$\alpha$ .cmp requires stratification by signage for the  $\alpha_5$  RAM only, where the encoded range must anticipate that either operand may become the result:

0	$L < -32$	always underflows
1	$-32 \leq L < 0$	underflows if result is unsigned
2	$0 \leq L < 32$	never overflows or underflows
3	$32 \leq L < 64$	overflows if result is signed
4	$64 \leq L$	always overflows
0	$R < -32$	always underflows
5	$-32 \leq R < 0$	underflows if result is unsigned
10	$0 \leq R < 32$	never overflows or underflows
15	$32 \leq R < 64$	overflows if result is signed
20	$64 \leq R$	always overflows

$\alpha$ .cmp passes its left operand unchanged to the transposed  $\beta$  layer.

### Shift and rotate operations: $\alpha$ .asr, $\alpha$ .e35, $\alpha$ .e36

The  $\alpha$ .asr operation is used by the **ASR** (arithmetic shift right) instructions, passing its left operand unchanged to the transposed  $\beta$  layer.  $\alpha_5$ 's most significant bit is copied

to its carry output. If the left operand is signed, RAM  $\theta$  will replicate that to all carry decision bits to  $\gamma$  so that sign extension for the right shift can occur during the  $\gamma$  layer. The same bit is copied to RAM  $\zeta$  using one of the encoded range bits, because the left operand's sign is relevant to overflow decisions. Another encoded range bit alerts  $\zeta$  when the right operand is zero, information that also affects overflow outcomes.

The  $\alpha.e35$  operation is used by the ASL (arithmetic shift left) instructions for range checking when the left operand is signed. The information required is whether or not a given shift amount, indicated by the right operand at all  $\alpha_i$ , would cause a 1 to be shifted into or through the sign bit, which is the  $-(2^{35})$  place value of the eventual output for a given  $L_i$  (left operand tribble), in which case propagate output bit  $p_i$  is set. If any  $p_i$  is set but the sign bit was initially clear, negative overflow has occurred. Also, if the shift amount would cause a 0 to be shifted into or through the sign bit, carry output bit  $c_i$  is set. If any  $c_i$  is set but the sign bit was initially set, positive overflow has occurred. RAM  $\alpha_5$  transmits the initial sign bit to RAM  $\zeta$  via one of the encoded range bits.

The  $\alpha.e36$  operation is used by the ASL (arithmetic shift left) instructions for range checking when the left operand is unsigned. Its implementation is identical to  $\alpha.e35$ , except that  $2^{36}$  place value is monitored instead of  $-(2^{35})$ . Unlike  $\alpha.e35$ , the downstream logic after  $\alpha.e36$  in RAMs  $\theta$  and  $\zeta$  only looks for positive overflow.

### **Multiply operations: $\alpha.dsl$ , $\alpha.mh$ , $\alpha.ml$**

$\alpha.dsl$  passes the left operand unchanged except for the most significant tribble, which is replaced with the right operand's most significant tribble. The propagate, carry, and encoded range outputs are not used.

$\alpha.mh$  multiplies the tribbles pairwise and passes the most significant tribble (bits 6–11) of each result to the transposed  $\beta$  layer. If  $\alpha_5$ 's result is nonzero, two encoded range bits are set to advise RAM  $\zeta$  that a MH instruction may be in the process of overflowing a short multiplication. The propagate and carry outputs are not used.

$\alpha.ml$  multiplies the tribbles pairwise and passes the least significant tribble (bits 0–5) of each result to the transposed  $\beta$  layer. The propagate, carry, and encoded range outputs are not used.

### **Nudge operation: $\alpha.nud$**

The NUDGE operation from section 7.2.6 begins with  $\alpha.nud$ , which presumes that for every tribble, the leftmost 1 bit of the right operand indicates the start bit for the NUDGE. That tribble’s output bits are the right operand’s least significant bits up to but not including the start bit, and the left operand’s most significant bits from the start bit upward. The carry output is set. If the right operand is all zeros, the carry output is not set, and the output is the left operand without changes.  $\theta$  and  $\gamma$  will assure that only the actual start bit is treated as such; all tribbles right of the tribble that contains the actual start bit will be overwritten by the  $\gamma$  layer with the right operand.

### **Mixing operations: $\alpha.mix$ , $\alpha.xim$**

The  $\alpha.mix$  and  $\alpha.xim$  operations implement the  $\alpha$  layer’s S-boxes for the MIX and XIM instructions. The left operand is the input word, and the right operand contains the key material (which S-box to apply) for each tribble. The forward mix is  $\alpha.mix$  and has inverse  $\gamma.xim$ ;  $\alpha.xim$  is the inverse of  $\gamma.mix$ .

$\alpha.mix$  and  $\alpha.xim$  only supply output to the transposed  $\beta$  layer. The propagate, carry, and encoded range outputs are not used.

### **Unary operations: $\alpha.uny$ , $\alpha.stu$**

Up to 64 unary operations, selected by the right operand, are provided by  $\alpha.uny$ . These operations can be mixed and matched across tribbles, so up to six different unary functions can be applied simultaneously during one instruction. Table 7.16 (p. 161) lists some anticipated operations which, depending on the ALU function

select lines, might or might not be used in combination with the other layers. Some might be given assembler macros in the future. The propagate, carry, and encoded range outputs are not used. See also section 7.2.9.

The  $\alpha$ .stu operation implements the  $\alpha$  layer of the stacked unary operations of section 7.2.10 and table 7.17 (p. 162). Output will be to the transposed  $\beta$  layer, propagate and carry output to RAM  $\theta$ , and/or encoded range to RAM  $\zeta$  as the operation requires. Some implementation descriptions for the stacked unary operations appear in section 7.3.7. For greater detail, the file `unary.c` in the firmware implementation is the best source of information.

### 7.3.2 $\beta$ layer operation

An alphabetical list of the  $\beta$  (beta) layer operations appears as table 7.12. Each of the six  $\beta$  RAMs accepts a six-bit left operand transposed from the  $\alpha$  layer and a six-bit untransposed right operand R. The operations themselves are small and simple. Which operation to perform is determined by six operation select input bits. The  $\beta$  layer portion of table 5.2 (p. 86) shows how these inputs map to SRAM address bits. The  $\beta$  RAMs each send a six-bit result to the untransposed  $\gamma$  layer. The  $\beta$  layer portion of table 5.3 (p. 87) shows how these outputs map to SRAM address bits.

#### Bitwise boolean operations: $\beta$ .id, $\beta$ .not, $\beta$ .txo

The  $\beta$ .id operation passes the left operand to the output unchanged. It is used for opcodes that the  $\beta$  layer does not participate in, such as `AND`, `MAX`, `PIT`, and `NUDGE`.

The  $\beta$ .not operation is a bitwise `NOT` of the left operand, with the right operand ignored. It is used alone and in combination with bitwise boolean operations from other layers to product opcodes such as `NOT`, `NAND`, `NOR`, `XNOR`, `LONR`, and `RONL`.

The  $\beta$ .txo operation is a bitwise `XOR` of the left operand, which is transposed, with the right operand, which is not transposed. It forms the basis of the `TXOR` opcode, which is interesting but would probably find rare use. `TXOR` can be used with

**Table 7.12:** ALU  $\beta$  layer operations.

name	description
$\beta$ .ham	swizzle for HAM2 instruction
$\beta$ .id	identity (pass all tribbles unchanged)
$\beta$ .imm	swizzle immediate argument out of CPU instruction
$\beta$ .mix	$\beta$ 's forward S-boxes
$\beta$ .not	bitwise NOT
$\beta$ .pat	R selects permutations of transposed tribbles
$\beta$ .rol	rotate transposed tribbles left
$\beta$ .rtl	rotate one tribble (6 bits of untransposed word) left
$\beta$ .shl	shift transposed tribbles left
$\beta$ .shr	shift transposed tribbles right
$\beta$ .stu	“stacked unary” functions that engage $\alpha$ , $\beta$ , and $\gamma$ together
$\beta$ .swz	frequently-used transposed tribble swizzle operations
$\beta$ .txo	bitwise XOR of transposed tribbles with non-transposed R
$\beta$ .uny	R selects simple unary functions on transposed tribbles
$\beta$ .xim	inverses of $\beta$ 's forward S-boxes

a zero left operand to compute the transposition of its 36-bit right operand.

### **Shift and rotate operations: $\beta$ .rol, $\beta$ .rtl, $\beta$ .shl, $\beta$ .shr**

The  $\beta$ .rol operation provides the first stage of a logarithmic shifter, supporting full-word rotations by the number of bits in each tribble of the right operand. Because the  $\beta$  layer's input and output are transposed,  $\beta$ .rol moves bits between tribbles, retaining the same position within any particular tribble. See also figure 4.9 (p. 65) and section 4.6. The  $\gamma$ .rol operation will complete this rotation.

The  $\beta$ .rtl (rotate tribble left) operation ignores the right operand and rotates the left operand one bit position left. Because the  $\beta$  layer is transposed, the effect of this is to rotate the entire 36-bit word six bit positions to the left. This operation supports the multiplication opcodes MH and DSL.

The  $\beta$ .shl operation is an augmented version of  $\beta$ .rol, in that the most significant  $n$  bits, where  $n$  is the rotation amount, are zeroed prior to the rotation. The net effect



is a left shift, which is completed by the  $\gamma$ .rol operation.

The  $\beta$ .shr operation is an augmented version of  $\beta$ .rol, in that the least significant  $n$  bits, where  $n$  is the rotation amount had it been expressed as a right rotation, are zeroed prior to the rotation. The net effect is a right shift, which is completed by the  $\gamma$ .rol operation.

### **Bit permute operation: $\beta$ .pat**

The  $\beta$ .pat operation implements permutations on the transposed tribbles of the  $\beta$  layer. The bits to permute come from the left operand, and the permutation is selected by the right operand according to table 7.8 (p. 124).  $\beta$ .pat supports the PAT and PAIT opcodes.

### **Mixing operations: $\beta$ .mix, $\beta$ .xim**

The  $\beta$ .mix and  $\beta$ .xim operations implement the  $\beta$  layer's S-boxes for the MIX and XIM instructions. The left operand is the transposed input word, and the right operand contains the key material (which S-box to apply) for each tribble.  $\beta$ .mix and  $\beta$ .xim are inverses of each other.

### **Unary operations: $\beta$ .uny, $\beta$ .stu**

Up to 64 unary operations, selected by the right operand, are provided by  $\beta$ .uny. These operations can be mixed and matched across tribbles, so up to six different unary functions can be applied simultaneously during one instruction. Table 7.16 (p. 161) lists some anticipated operations which, depending on the ALU function select lines, might or might not be used in combination with the other layers. Some might be given assembler macros in the future. The propagate, carry, and encoded range outputs are not used. See also section 7.2.9.

The  $\beta$ .stu operation implements the  $\beta$  layer of the stacked unary operations of section 7.2.10 and table 7.17 (p. 162). Output is to the untransposed  $\gamma$  layer. More in-

formation about stacked unary operations appears in section 7.3.7. For greater detail, the file `unary.c` in the firmware implementation is the best source of information.

### Swizzle operations: $\beta$ .swz, $\beta$ .imm, $\beta$ .ham

The  $\beta$ .swz operation enables the transposed  $\beta$  layer to replicate and/or relocate among the six subwords of a CPU word. Table 7.7 (p. 114) gives a list of the currently defined swizzles.

The  $\beta$ .imm operation is used in conjunction with `ff m` (the immediate argument flip-flop of figure 8.6 on p. 200) to transfer the immediate argument field from the current CPU instruction word to the  $\gamma$  layer. This swizzle is from the CPU word:

opcode	dest. register	immediate value
bits 35–27	bits 26–18	bits 17–0

to two copies of the immediate value:

immediate value	immediate value
bits 35–18	bits 17–0

The  $\gamma$  layer will either retain both immediate values, replace the right copy with zeros, replace the left copy with zeros, or replace the left copy with ones (sign extend a negative immediate value).

The  $\beta$ .ham operation moves bits from (untransposed) positions 6, 7, 8, 12, 13, 14 to (untransposed) positions 0–5 respectively. The remaining positions 6–35 are cleared.  $\beta$ .ham’s use for population counting via the HAM2 instruction is discussed in section 7.3.7.

### 7.3.3 $\gamma$ layer operation

An alphabetical list of the  $\gamma$  (gamma) layer operations appears as table 7.13. Rather than having six function select input bits as the  $\alpha$  and  $\beta$  layers do, the  $\gamma$  RAMs have

**Table 7.13:** ALU  $\gamma$  layer operations.

name	T	description
$\gamma$ .add		add carry decisions from $\theta$ to tribbles
$\gamma$ .ham	1	shift tribble 2 left 3 bits for HAM2 instruction
$\gamma$ .imh	1	place 18-bit immediate value in upper half of 36-bit word
$\gamma$ .imn	1	sign-extend 18-bit negative immediate value to 36 bits
$\gamma$ .imp	0	sign-extend 18-bit positive immediate value to 36 bits
$\gamma$ .kil		kill (set to zero) tribbles identified by $\theta$
$\gamma$ .mix	0	$\gamma$ 's forward S-boxes
$\gamma$ .mux		replace tribbles identified by $\theta$ with tribbles from R
$\gamma$ .pit	0	R selects permutations of tribbles
$\gamma$ .rol		rearrange partially-rotated tribbles into correct order
$\gamma$ .stu		“stacked unary” functions that engage $\alpha$ , $\beta$ , and $\gamma$ together
$\gamma$ .uny	0	R selects simple unary functions on tribbles
$\gamma$ .xim	1	inverses of $\alpha$ 's forward S-boxes
$\gamma$ .xor	1	bitwise XOR

Many of these require  $\theta.t$  and the indicated T(emporal) flag input.

five function select input bits that are common to all six RAMs. Each RAM  $\gamma_i$  also has its own carry decision bit  $d_i$  from the  $\theta$  RAM. So rather than having 64 memory “slots” to support different operations, the  $\gamma$  layer has 32 slots that are twice as large, because of their carry decision input. Each of the 32 slots can hold one operation that uses the  $d_i$  bits, or two operations that do not use the  $d_i$  bits, but are differentiated by them.

Table 7.13 lists the  $\gamma$  layer operations. Column T indicates those operations that use only one-half slot, in which case the  $d_i$  must be set as marked in column T. The  $d_i$  are set to 0 or 1 by forcing the incoming T(emporal) flag to 0 or 1, and using the  $\theta.t$  operation to distribute it to the six  $\gamma$  RAMs. Where column T is blank, the operation uses the  $d_i$  decisions in its computation and requires a full slot.

The  $\gamma$  sections of tables 5.2 and 5.3 (pp. 86 and 87) specifically list the input and output bits of the  $\gamma$  RAMs. Each  $\gamma$  RAM delivers two six-bit copies of its result, in order that the result can be written to the left and right register file SRAMs

simultaneously. Because the data wiring at the register file is bidirectional and both copies must be read simultaneously, separate wiring is needed for the two register copies. This is why  $\gamma$  cannot use the same output bits for both copies.

Certain of the  $\gamma$  layer's output bits work identically across all operations. These are the tribble zero bits, one per  $\gamma$  RAM, which are set when the six principal outputs are all zero. Supplying these bits from  $\gamma$  rather than computing them outside speeds and simplifies the eventual zero computation for the 36-bit word. There is also a third copy of the  $Y_{35}$  bit. Although two copies are already available in the output to the two register files, those circuit board tracks are heavily loaded. The third copy, needed for N(egative) flag computation, is separated in order to not further increase capacitance and delay on the other two.

### **Additive operation: $\gamma$ .add**

The  $\gamma$ .add operation ignores the right operand, and adds the carry decision  $d_i$  to the left operand. The output is allowed to wrap modulo  $2^6$ .

### **Bitwise boolean operation: $\gamma$ .xor**

The  $\gamma$ .xor operation computes the bitwise XOR of the left and right operands.

### **Tribble boolean operations: $\gamma$ .imh, $\gamma$ .imn, $\gamma$ .imp, $\gamma$ .kil, $\gamma$ .mux**

The tribble boolean operations provide many-to-one bitwise boolean logic. The “many” here are the six bits of the left operand, and in the case of  $\gamma$ .mux only, the right operand also. The “to-one” either comes from a constant 0 or 1 depending on the tribble position, or comes from the  $d_i$  carry decisions.

The  $\gamma$ .imh (immediate high) operation passes the 18 most significant bits unchanged, but zeros the 18 least significant bits. This supports the IMH (load immediate high) CPU opcode of section 8.6.3. Because the  $d_i$  inputs are not used, this operation fits in a half slot.

The  $\gamma$ .imn (immediate negative) operation sets the 18 most significant bits to 1s, but passes the 18 least significant bits unchanged. This supports the **IMN** (load immediate negative) CPU opcode of section 8.6.3. Because the  $d_i$  inputs are not used, this operation fits in a half slot.

The  $\gamma$ .imp (immediate positive) operation clears the 18 most significant bits, but passes the 18 least significant bits unchanged. This supports the **IMP** (load immediate positive) CPU opcode of section 8.6.3. Because the  $d_i$  inputs are not used, this operation fits in a half slot.

The **IMB** (load immediate both) CPU opcode does not have  $\gamma$  layer operation of its own, because the  $\beta$  layer has already replicated the 18-bit constant from the instruction into both halves of the 36-bit word.

The  $\gamma$ .kil (kill) operation passes the left operand unchanged for those tribbles where  $d_i = 0$ , and returns all zeros for those tribbles where  $d_i = 1$ . This supports opcodes such as **IGF**. Because the  $d_i$  inputs are used, this operation requires a full slot.

The  $\gamma$ .mux (mux, or multiplex) operation passes the left operand unchanged for those tribbles where  $d_i = 0$ , and passes the right operand unchanged for those tribbles where  $d_i = 1$ . This supports opcodes such as **MIN** and **MAX**. Because the  $d_i$  inputs are used, this operation requires a full slot.

### **Shift and rotate operations: $\gamma$ .rol**

The  $\gamma$ .rol completes a right rotation by providing the second stage of a logarithmic shifter. The  $\beta$  layer has already rotated all bits into their correct tribbles, but their position within the tribbles requires correction.  $\gamma$ .rol applies this correction. Although this operation is a left rotation, a right rotation can be accomplished using  $36 - (\text{rotation amount})$  as the right operand.

For the **ASR** (arithmetic shift right) opcodes,  $\gamma$ .rol provides sign extension in addition to completing the rotation. The  $\alpha$ .asr and  $\theta$ .c5 operations copy the most

significant bit of the original left operand—its so-called sign bit—to all of the  $d_i$  bits. Upon finding  $d_i$  set, the  $\gamma$  RAMs will OR in 1 bits at the correct place values based on the right operand to achieve the sign extension. Because the  $d_i$  inputs are used to support the **ASR** opcode,  $\gamma.\text{rol}$  requires a full slot.

### **Bit permute operations: $\gamma.\text{pit}$ , $\gamma.\text{ham}$**

The  $\gamma.\text{pit}$  operation implements permutations within tribbles in the  $\gamma$  layer. The bits to permute come from the left operand, and the permutation is selected by the right operand according to table 7.8.  $\gamma.\text{pit}$  supports the **PIT** and **PAIT** opcodes. Because the  $d_i$  inputs are not used, this operation fits in a half slot.

The  $\gamma.\text{ham}$  operation moves three bits from positions 12, 13, 14 to positions 15, 16, 17. The positions moved from are replaced with zeros, and bits overwritten are lost.  $\gamma.\text{ham}$ 's use for population counting via the **HAM2** instruction is discussed in section 7.3.7. Because the  $d_i$  inputs are not used, this operation fits in a half slot.

### **Mixing operations: $\gamma.\text{mix}$ , $\gamma.\text{xim}$**

The  $\gamma.\text{mix}$  and  $\gamma.\text{xim}$  operations implement the  $\gamma$  layer's S-boxes for the **MIX** and **XIM** instructions. The left operand is the input word, and the right operand contains the key material (which S-box to apply) for each tribble. The forward mix is  $\gamma.\text{mix}$  and has inverse  $\alpha.\text{xim}$ ;  $\gamma.\text{xim}$  is the inverse of  $\alpha.\text{mix}$ . Because the  $d_i$  inputs are not used, these operations each fit in a half slot.

### **Unary operations: $\gamma.\text{uny}$ , $\gamma.\text{stu}$**

Up to 64 unary operations, selected by the right operand, are provided by  $\gamma.\text{uny}$ . These operations can be mixed and matched across tribbles, so up to six different unary functions can be applied simultaneously during one instruction. Table 7.16 (p. 161) lists some anticipated operations which, depending on the ALU function select lines, might or might not be used in combination with the other layers. Some

**Table 7.14:** ALU  $\theta$  RAM operations.

name	description
$\theta.2l$	$d_i \iff \exists j > i : c_j = 1$
$\theta.2r$	$d_i \iff \exists j < i : c_j = 1$
$\theta.add$	$d_i$ = carry-skip decisions based on $\alpha$ 's propagates and carries
$\theta.dsl$	causes the T(emporal) flag to be added at place value $2^6$ (1)
$\theta.gt$	$d = 111111'b$ if $L > R$ , else $d = 000000'b$
$\theta.lt$	$d = 111111'b$ if $L < R$ , else $d = 000000'b$
$\theta.rev$	supports addition of bit-reversed words (2)
$\theta.t$	$d_i = T(\text{emporal})$ flag

(1) The DSL instruction effectively adds any previous carry prior to shifting, but  $\beta$  completes the shift before  $\gamma$  can increment the tribbles.  $\theta.dsl$  computes rotated carry decisions in their correct place values for  $\gamma$ .

(2) This is the same as  $\theta.add$ , except propagating carries from left to right.

might be given assembler macros in the future. See also Section 7.2.9. Because the  $d_i$  inputs are not used, the  $\gamma.uny$  operation fits in a half slot.

The  $\gamma.stu$  operation implements the  $\gamma$  layer of the stacked unary operations of section 7.2.10 and table 7.17 (p. 162). More information about stacked unary operations appears in section 7.3.7. For greater detail, the file `unary.c` in the firmware implementation is the best source of information. Because the  $d_i$  inputs are available to support some of the stacked unary operations,  $\gamma.stu$  requires a full slot.

### 7.3.4 $\theta$ operation

The  $\theta$  (theta) RAM operates while the  $\beta$  layer is active, accepting one propagate bit  $p_i$  and one carry bit  $c_i$  from each RAM  $\alpha_i$ . Able to see all 12 of these bits as well as the modified incoming T(emporal) flag,  $\theta$  computes one *carry decision* bit  $d_i$  for each of the six  $\gamma_i$  RAMs. This computation usually implements carry propagation, but on occasions implements some other kind of information propagation to or between subwords.  $\theta$  can also compute a two-bit *carry summary* for RAM  $\zeta$  to use when

determining flags. The  $\theta$  sections of tables 5.2 and 5.3 (pp. 86 and 87) list all of RAM  $\theta$ 's input and output bits.

### **Additive operations: $\theta$ .add, $\theta$ .rev**

The  $\theta$ .add operation is the ordinary (right-to-left) carry propagation mechanism:

$$d_i = \begin{cases} \text{incoming T(emporal) flag} & \text{when } i = 0, \\ c_{i-1} \vee p_{i-1} \wedge d_{i-1} & \text{when } i > 0. \end{cases}$$

The  $\theta$ .rev operation offers reverse (left-to-right) carry propagation for weird macros like IBRI (increment bit-reversed integer):

$$d_i = \begin{cases} \text{incoming T(emporal) flag} & \text{when } i = 5, \\ c_{i+1} \vee p_{i+1} \wedge d_{i+1} & \text{when } i < 5. \end{cases}$$

### **Compare operations: $\theta$ .gt, $\theta$ .lt**

The  $\theta$ .gt and  $\theta$ .lt operations support the MAX and MIN opcodes.  $\theta$ .gt sets all  $d_i$  to 1s if the right operand is greater than the left operand. Otherwise, all  $d_i$  are 0s.  $\theta$  can determine this, because for each subword,  $\alpha$ .cmp sets  $c_i$  where  $R_i > L_i$ , and sets  $p_i$  where  $L_i > R_i$ .

$\theta$ .lt sets all  $d_i$  to 1s if the left operand is greater than the right operand. Otherwise, all  $d_i$  are 0s.

Both  $\theta$ .gt and  $\theta$ .lt provide RAM  $\zeta$  a copy of what the  $d_i$  are set to.

### **Multiply operation: $\theta$ .dsl**

The  $\theta$ .dsl operation supports the DSL (double shift left) opcode. This operation actually happens has nothing to do with a left shift, but implements an ADC instruction within DSL in order to speed long multiplication.

$\theta$ .dsl effectively adds the T(emporal) flag to the left operand prior to the left



shift. This addition is as if the right operand were zero—in practice, the right operand provides bits to the double shift. What’s tricky here is that before  $\gamma$  can apply the carry decisions,  $\beta$  has already shifted the left addend. So the implementation of  $\theta.dsl$  is to first to  $\theta.add$ , and then shift all  $d_i$  one tribble left. This causes the  $\gamma$  layer to add the T(emporal) flag at place value  $2^6$  instead of  $2^0$ .

### **Tribble scan operations: $\theta.2l$ , $\theta.2r$**

The tribble scan operation  $\theta.2l$  (to left) is not presently in use. It finds the minimum  $k$  such that  $c_k$  is set. If no  $c_i$  is set, then  $k = 5$ . Then

$$d_i = \begin{cases} 0 & \text{when } i \leq k, \\ 1 & \text{when } i > k. \end{cases}$$

The tribble scan operation  $\theta.2r$  (to right) supports the NUDGE opcode by finding the maximum  $k$  such that  $c_k$  is set. If no  $c_i$  is set, then  $k = 0$ . Then

$$d_i = \begin{cases} 0 & \text{when } i \geq k, \\ 1 & \text{when } i < k. \end{cases}$$

### **Temporal flag replication: $\theta.t$**

The  $\theta.t$  operation copies the incoming modified T(emporal) flag to all  $d_i$ . It is most often used to select either of two  $\gamma$  operations that only require a half slot, such as between  $\gamma.mix$  and  $\gamma.xim$ .

## **7.3.5 $\zeta$ operation**

The  $\zeta$  (zeta) RAM operates while the  $\gamma$  layer is active and updates the CPU flags N(egative), Z(ero), T(emporal overrange), R(ange), and I(nterrupt). Its inputs are the previous flags, the two-bit carry summary from RAM  $\theta$ , and the five-bit encoded range from  $\alpha_5$ . Hardware for the flags is described in section 5.3. Input and output

**Table 7.15:** ALU ζ RAM operations.

name	description
ζ.add.s	addition with signed result
ζ.add.u	addition with unsigned result
ζ.asl.uu	overflow check after unsigned to unsigned left shift
ζ.asl.us	overflow check after unsigned to signed left shift
ζ.asl.su	overflow check after signed to unsigned left shift
ζ.asl.ss	overflow check after signed to signed left shift
ζ.asr.uu	flags after unsigned to unsigned right shift
ζ.asr.us	overflow check after unsigned to signed right shift
ζ.asr.su	overflow check after signed to unsigned right shift
ζ.asr.ss	flags after signed to signed left shift
ζ.bound	set I(nterrupt) flag if array index out of bounds
ζ.clear.r	clear R(ange) flag after copying it to T(emporal) flag
ζ.cmp	update N(egative) and Z(ero) flags without changing T(emporal)
ζ.logic	update N(egative) and Z(ero) flags for non-numeric instruction
ζ.t.adj	support shifts and rotates through T(emporal) flag (1)
ζ.wrap	addition with wrapping allowed

(1) Shift or rotate through T is achieved in one stacked unary instruction.

bits for ζ appear in the ζ sections of tables 5.2 and 5.3 (pp. 86 and 87).

### Additive operations: ζ.add.s, ζ.add.u, ζ.wrap

The ζ.wrap operation supports the **AW**, **AWC**, **SW**, **SWC**, **RSW**, and **RSWC** opcode families. Flags are set as if the result and both operands are unsigned. The T(emporal) flag will indicate whether or not wrapping has occurred, but the R(ange) flag will be left unchanged. In other words, the **AW** opcode and its ilk can detect overflow, but it is not considered an error. The Z(ero) flag will be set if the true result, but not a wrapped result, is zero.<sup>9</sup> The N(egative) flag will be cleared.

The ζ.add.u operation supports the 24 non-wrapping additive opcodes of table 7.2 that have an unsigned result. If the true result does not fit in an unsigned word, the T(emporal) and R(ange) flags will both be set. Otherwise the T(emporal) flag will

---

<sup>9</sup>I question whether or not this reflects correct semantics. This behavior may have to change.

be cleared, and the R(ange) flag will not change. The N(egative) flag will be set if the true result is negative; note that because this is an out-of-range condition, the T(emporal) and R(ange) flags will also be set. If the true result is not negative, the N(egative) flag will be cleared. The Z(ero) flag will be set if the true result is zero and cleared otherwise.

The `ζ.add.s` operation supports the 24 non-wrapping additive opcodes of table 7.2 that have a signed result. If the true result does not fit in a signed word, the T(emporal) and R(ange) flags will both be set. Otherwise the T(emporal) flag will be cleared, and the R(ange) flag will not change. The N(egative) flag will be set if the true result is negative, otherwise the N(egative) flag will be cleared. The Z(ero) flag will be set if the true result is zero and cleared otherwise.

### **Bitwise boolean operations: `ζ.logic`**

The `ζ.logic` operation supports opcodes don't involve out-of-range conditions. The T(emporal) and R(ange) flags do not change. The N(egative) flag is a copy of the leftmost bit, irrespective of the destination register's signage. The reason for this preventable semantic faux pas is to allow an immediate conditional branch to be decided by the leftmost bit from a non-arithmetic instruction. The Z(ero) flag is set if all 36 result bits are zero, and cleared otherwise.

### **Compare operations: `ζ.cmp`, `ζ.bound`**

The `ζ.cmp` operation supports the `CMP` family of opcodes. Because none of these opcodes store a result to a register, it is impossible for them to produce out-of-range results. The T(emporal) and R(ange) flags are not changed by `ζ.cmp`, but the N(egative) and Z(ero) flags are set or cleared in relation to the true ordering of the left and right operands.

The `ζ.bound` operation supports the `BOUND` family of opcodes. The N(egative), Z(ero), T(emporal), and R(ange) flags do not change. If the index is out of range, the

I(nterrupt) flag will be set. Further semantics and implementation for the I(nterrupt) flag remain to be determined.

### **Shift and rotate operations: $\zeta$ .asl, $\zeta$ .asr, $\zeta$ .t.adj**

Four  $\zeta$ .asl operations are stratified by signedness and support the four **ASL** opcodes. The **ASL** right operand is always between 0 and 63 and is replicated across every tribble, but the left operand (the word to shift) may be unsigned or signed, and the destination may also be unsigned or signed. The  $\zeta$ .asl family sets the N(egative), Z(ero), T(emporal), and R(ange) flags to reflect the true result and any overrange situation. The logic to accomplish this is intricate, and the file `zeta.c` from the firmware source code is its best documentation.

Four  $\zeta$ .asr operations are stratified by signedness and support the four **ASR** opcodes. The **ASR** right operand is always between 0 and 63 and is replicated across every tribble, but the left operand (the word to shift) may be unsigned or signed, and the destination may also be unsigned or signed. The  $\zeta$ .asr family sets the N(egative), Z(ero), T(emporal), and R(ange) flags to reflect the true result and any overrange situation.<sup>10</sup> The logic to accomplish this is intricate, and the file `zeta.c` from the firmware source code is its best documentation.

The  $\zeta$ .t.adj operation uses the  $\alpha$  layer's  $p_i$  and  $c_i$  outputs to adjust the T(emporal) flag. This operation supports shifts into and rotates through the T(emporal) flag via the **RTGL**, **RTGR**, **STGL**, and **STGR** opcodes and their corresponding stacked unary operations **su.rcl** and **su.rcr**. The adjustment to the T(emporal) flag is as follows:

- If no  $p_i$  and no  $c_i$  are set, T is unchanged.
- If no  $p_i$  is set and any  $c_i$  is set, T is set.
- If any  $p_i$  is set and no  $c_i$  is set, T is cleared.
- If any  $p_i$  is set and any  $c_i$  is set, T is inverted.

---

<sup>10</sup>ASR results may not fit if the left operand and destination registers differ in signedness.

### Flag operation: $\zeta$ .clear.r

The  $\zeta$ .clear.r operation implements the CRF (clear range flag) instruction by copying the present value of the R(ange) flag to the T(emporal) flag, and then clearing the R(ange) flag.

## 7.3.6 Simple unary operations

Computations with only one input generally use the left argument, leaving the right argument free to specify which of 64 operations to compute. This affords SRAM ALUs a lot of freebies we wouldn't otherwise enjoy. The simplest cases occur when everything is done within 6-bit subwords, and the tribbles are completely independent of each other. These are merely lookup tables that are available in the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers using the UN.A, UN.B, and UN.G opcodes.

Table 7.16 provides a list of the simple unary operations that are specified to date. Note that the right operand is specified on a per-tribble basis, meaning that up to six operations from table 7.16 may be applied across a word during one instruction. Slots 00'o–10'o provide constants 0, 1, and all ones, pass-through and bit inversion, and tests for zero and nonzero with output alternatives. 11'o–16'o offer counts of zeros and ones with total, leading, and trailing options. Slots 17'o–20'o allow tribbles to be individually incremented and decremented. Slot 22'o reverses the bits of the tribbles.

Slot 21'o is currently the sole operation where the tribble outputs are not identical when given the same input. The left input is a number between 0 and 63, and must be provided across all six tribbles. The right input is 21'o (the operation to perform) in all tribbles. The result is the largest 36-bit word the left input can be multiplied by without overflow, assuming the product is unsigned. Another way to view this operation is as a fixed-point reciprocal of the input value, with all ones output when dividing by zero.

**Table 7.16:** ALU simple unary operations.

<b>name</b>	<b>octal</b>	<b>description</b>	(1)
un.zt	00	000000‘b	
un.ot	01	000001‘b	
un.oxt	02	111111‘b	
un.idt	03	identity	
un.nott	04	bitwise NOT	
un.iszt	05	000001‘b if tribble = 0 else 000000‘b	
un.isnzt	06	000001‘b if tribble $\neq$ 0 else 000000‘b	
un.iszxt	07	111111‘b if tribble = 0 else 000000‘b	
un.isnzxt	10	111111‘b if tribble $\neq$ 0 else 000000‘b	
un.czt	11	count zeros	
un.clzt	12	count leading zeros	
un.ctzt	13	count trailing zeros	
un.cot	14	count ones	
un.clot	15	count leading ones	
un.ctot	16	count trailing ones	
un.inct	17	increment with wrap	
un.dect	20	decrement with wrap	
un.recip	21	$\lfloor (2^{36} - 1) \div \max(\text{tribble}, 1) \rfloor$	(2)
un.revt	22	reverse bits	
	23–77	reserved	

(1) Throughout this table, six unary operations, selected by the tribbles of R, are done simultaneously on the six input tribbles.

(2) Reciprocal: Given  $t : 0 \leq t < 64$ , return largest  $m : mt < 2^{36}$ .  $t$  must be in all six tribbles of L. Result  $m$  is a whole word.

### 7.3.7 Stacked unary operations

Table 7.17 lists more complex unary operations. These require participation of multiple layers and sometimes need  $\theta$  to support operations across subwords. For this reason, the slots specified by the right operand are numerically consistent from layer to layer. The name “stacked unary” arises from the involvement of these lock-stepped layers. Table 7.18 shows how the stacked unary instructions would combine into easy-to-use assembler macros. Their reverse Polish notation (RPN) implementation

**Table 7.17:** ALU stacked unary operations (1).

name	octal	description
su.abs2s	00	assume $x < 0$ , return $-x$ , result is signed (2)
su.abs2i	01	assume $x \geq 0$ , return $x$ (identity) (2)
su.abs2u	02	assume $x < 0$ , return $-x$ , result is unsigned (2)
su.abs1s	03	if $x < 0$ then 000000't else 111111't (3)
su.abs1u	04	if $x < 0$ then 222222't else 111111't (3)
su.fabs	05	$ x $ , only works when $ x  < 2^{30}$ , range checked
su.signum	06	-1, 0, or +1 based on $x < 0$ , $x = 0$ , or $x > 0$
su.slprep	07	convert $x$ to left shift control word, range checked
su.srprep	10	convert $x$ to right shift control word, range checked
su.rlprep	11	convert $x$ to left rotate control word, range checked
su.rrprep	12	convert $x$ to right rotate control word, range checked
su.ham0	13	2-instruction popcount, count 0s, first instruction
su.ham1	14	2-instruction popcount, count 1s, first instruction
su.llz	15	light leading 0s for CLZ
su.llo	16	light leading 1s for CLO
su.ltz	17	light trailing 0s for CTZ
su.lto	20	light trailing 1s for CTO
su.parity	21	parity, result in $2^0$ position only
su.increv	22	increment bit-reversed word mod $2^{36}$ (4)
su.decrev	23	decrement bit-reversed word mod $2^{36}$ (4)
su.lfsr	24	36-bit LFSR
su.xpoly	25	XOR 410000 010166'o if T is set
su.rcl	26	shift/rotate one place left into/through T
su.rcr	27	shift/rotate one place right into/through T
su.cx	30	6 copies of tribble 0 with range check
	32-77	reserved

- (1) The  $\alpha$ ,  $\beta$ , and  $\gamma$  layers operate simultaneously to do these operations. The selected operation must be copied to all six tribbles of R.
- (2) Encoding must be consistent with su.abs1s and su.abs1u results.
- (3) Almost self-modifying code, but safely in a register. The result is the stacked-unary encoding that completes an absolute value computation.
- (4) The T(emporal) flag is not changed.

**Table 7.18:** Assembler macros that (mainly) use stacked unary operations.  
The firmware supports these as of September 2022, but not the assembler yet.

name	length	description	RPN implementation
ABS	2	absolute value	$x \ x \text{ su.abs1 su } (1)$
CLO	3	count leading ones	$x \text{ su.llo su.ham1 HAM2}$
CLZ	3	count leading zeros	$x \text{ su.llz su.ham1 HAM2}$
CTO	3	count trailing ones	$x \text{ su.lto su.ham1 HAM2}$
CTZ	3	count trailing zeros	$x \text{ su.ltz su.ham1 HAM2}$
CX	1	check and extend	$x \text{ su.cx}$
FABS	1	fast absolute value	$x \text{ su.fabs}$
LFSR	1	linear feedback shift register	$x \text{ su.lfsr}$
MIR	1	mirror the bits of a word	$x \text{ 131313131313' o PAIT}$
MIRD	1	mirrored decrement	$x \text{ su.decrev}$
MIRI	1	mirrored increment	$x \text{ su.increv}$
PARTY	1	parity	$x \text{ su.parity}$
POPC	2	popcount	$x \text{ su.ham1 HAM2}$
PRL	1	prepare to rotate left	$x \text{ su.rlprep}$
PRR	1	prepare to rotate right	$x \text{ su.rrprep}$
PSL	1	prepare to shift left	$x \text{ su.slprep}$
PSR	1	prepare to shift right	$x \text{ su.srprep}$
RTGL	1	rotate through T going left	$x \text{ su.rcl } (T_{\text{in}} \text{ intact})$
RTGR	1	rotate through T going right	$x \text{ su.rcr } (T_{\text{in}} \text{ intact})$
STGL	1	shift into T going left	$x \text{ su.rcl } (T_{\text{in}} = 0)$
STGR	1	shift into T going right	$x \text{ su.rcr } (T_{\text{in}} = 0)$
XPOLY	1	XOR polynomial if T is set	$x \text{ su.xpoly}$
	25–63	reserved	

(1) The first stacked unary operation determines the second.



is shown; however, their corresponding **STUN**-series opcodes remain to be finalized.

The best documentation for the stacked unary operations is their source code, which is the file `unary.c` in [Abel22b]. Unfortunately, that code too would benefit from lengthier explanations. Although running the firmware in the virtual machine can show that the stated operations work, it's not obvious *how* in some instances. Below are short descriptions of the three most complex operations thus far.

### **Absolute value in two instructions**

According to table 7.17, a stacked unary instruction with a right operand of `000000't` will negate its left operand. But if the right operand is `111111't`, it will return the left operand unchanged. There happens to be another stacked unary operation, `333333't`, that examines checks the most significant bit of the left operand and returns `000000't` if it is set and `111111't` if it is not set. So in two instructions, it's possible to find the absolute value of a number:

```
signed input output
unsigned either_or

input = ...
either_or = input stun.a 333333't
output = input stun.a either_or
```

### **Limited-domain absolute value in one instruction**

There is also a “fast” absolute value that uses just one instruction, and only works if the absolute value is less than  $2^{30}$ . Under this constraint, the rightmost six bits of the input are either all ones (the input is negative) or all zeros (the input is not negative). When the input is transposed for the  $\beta$  layer, each of the  $\beta$  RAMs receives one of these sign bit copies, allowing each of the  $\beta$  RAMs to invert their transposed subwords if the left operand was negative, or not invert if the left operand was positive. In the meantime,  $\alpha$ ,  $\theta$ ,  $\gamma$ , and the modified incoming T(emporal) flag are configured to increment the left argument by one, which with  $\beta$ 's bitwise NOT

would complete a two's complement negation. The NOT had a “kill switch” via the transposed sign bit copies in the event the left argument wasn't negative. The increment also needs a kill switch, and it's accomplished by  $\alpha_5$  and  $\theta$ . In the event  $\alpha_5$  doesn't find the most significant bit set, it sets both its propagate and carry output to 1s. In ordinary addition, either propagate or carry may be set, but never both by the same bitslice. When  $\theta$  sees a propagate and carry arrive from the same RAM, it interprets this as a “secret message” to leave all carry decisions zero, which prevents  $\gamma$  from doing the increment. So absolute value is implemented in a single instruction, on the precondition that the result will be less than  $2^{30}$ .

### Population count in two instructions

For many years, it was uncommon for traditional CPUs to include an instruction that counts the number of one bits in a word. Intel's POPCOUNT instruction did not appear until several years after x86 became available with 64 bits. Without such an instruction, a simple count of set bits in a word was cumbersome and inspired many creatively-written subroutines to avoid expensive loops.

My ALU can count the ones or zeros within a word in two instructions. Here's a code example, and an explanation follows.

```

unsigned word      ; will count 1s in this
unsigned t         ; temporary register
unsigned ones      ; output

word = ...
t = word stun.c 141414141414'o
ones = t ham2 t

```

The first instruction is the stacked unary operation `su.ham1`, so named because another term for popcount is Hamming weight. The  $\alpha$  layer counts the ones in each of the tribbles, leaving a count between 0 and 6 in the three low bits of each tribble. To obtain the total popcount, it suffices to somehow add these post- $\alpha$  bits together, assigning the following place values to each bit:

000421 000421 000421 000421 000421 000421

As these bits flow to the  $\beta$  layer, they are transposed. The weights are now:

000000 000000 000000 444444 222222 111111

The  $\beta$  layer counts the bits in the above tribbles, leaving a count between 0 and 6 in the three low bits of the three least significant tribbles. To obtain the total popcount, it suffices to somehow add these post- $\beta$  bits together, assigning the following place values to each bit, where  $G = 16$ :

000000 000000 000000 000G84 000842 000421

As these bits flow to the  $\gamma$  layer, they are untransposed. The weights are now:

000000 000000 000000 000G84 000842 000421

Six bits of interest did actually move, but their untransposed locations turn out to have the same weights after the move. What happened may be easier to see if the weights are given in matrix form—recall that the  $\alpha$ - $\beta$  and  $\beta$ - $\gamma$  wiring transpositions are simply reflections through the main diagonal:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	G	8	4
0	0	0	8	4	2
0	0	0	4	2	1

The  $\gamma$  layer now adjusts tribble 2 only, shifting left three bits in order to find lanes for a swizzle:

000000 000000 000000 G84000 000842 000421

This is the end of the first instruction. All the second instruction needs to do is sum the above bits with the given place values. A stacked unary instruction will not suffice: the task calls for an instruction that supports a right operand. This

instruction is called HAM2, so-named because it is the second of two steps to compute a Hamming weight. HAM2’s  $\alpha$  layer does nothing. The  $\beta$  layer does a quick swizzle, taking three bits each from subwords 2 and 1, and moving them to subword 0. Coming into the  $\gamma$  layer, the bit positions to sum are now these:

000000 000000 000000 000000 000000 G84842

Of the nine weighted bits that need to be added, six are available at  $\gamma_0$  via its left operand. The three missing bits are lost from the left side, but are present at  $\gamma_0$ ’s right operand with weights 000421 as required. It comes down to this single RAM to add these nine bits with the correct weights, and like all single-RAM computations, the task is a mere table lookup. The result will be in the range 0 through 36, which fits properly in the RAM’s six-bit output and indicates the correct population count. This tested successfully in October 2022 using [Abel22b] with one million words from `/dev/urandom`.

## 7.4 Leading and trailing bit manipulation

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

The full set of these manipulations could be implemented by my ALU as stacked unary operations for leading and trailing use, making them accessible via one instruction. But there are 40 of these: search for 0 vs. 1  $\times$  invert output or not  $\times$  leading vs. trailing  $\times$  five “what to do.” This would be a lot of address space to set aside for these rarely-used operations, so I didn’t. Instead, all 40 can be accomplished in

**Table 7.19:** Leading and trailing bit manipulation macros.

The firmware supports these as of October 2022, but not the assembler yet.

name	length	description	RPN implementation
BOL	2	brighten ones left	<i>x</i> su.decrev <i>x</i> XOR
BOLI	2	brighten ones left invert	<i>x</i> su.decrev <i>x</i> XNOR
BOR	2	brighten ones right	<i>x</i> 1 SW <i>x</i> XOR
BORI	2	brighten ones right invert	<i>x</i> 1 SW <i>x</i> XNOR
BZL	2	brighten zeros left	<i>x</i> <i>x</i> su.increv XOR
BZLI	2	brighten zeros left invert	<i>x</i> <i>x</i> su.increv XNOR
BZR	2	brighten zeros right	<i>x</i> <i>x</i> 1 AW XOR
BZRI	2	brighten zeros right invert	<i>x</i> <i>x</i> 1 AW XNOR
EOL	2	erase ones left	<i>x</i> <i>x</i> su.increv AND
EOLI	2	erase ones left invert	<i>x</i> <i>x</i> su.increv NAND
EOR	2	erase ones right	<i>x</i> <i>x</i> 1 AW AND
EORI	2	erase ones right invert	<i>x</i> <i>x</i> 1 AW NAND
EZL	2	erase zeros left	<i>x</i> su.decrev <i>x</i> OR
EZLI	2	erase zeros left invert	<i>x</i> su.decrev <i>x</i> NOR
EZR	2	erase zeros right	<i>x</i> 1 SW <i>x</i> OR
EZRI	2	erase zeros right invert	<i>x</i> 1 SW <i>x</i> NOR
FOL	2	find one left	<i>x</i> su.decrev <i>x</i> RANL
FOLI	2	find one left invert	<i>x</i> su.decrev <i>x</i> LONR
FOR	2	find one right	<i>x</i> 1 SW <i>x</i> RANL
FORI	2	find one right invert	<i>x</i> 1 SW <i>x</i> LONR
FZL	2	find zero left	<i>x</i> su.decrev <i>x</i> RANL
FZLI	2	find zero left invert	<i>x</i> su.decrev <i>x</i> LONR
FZR	2	find zero right	<i>x</i> 1 SW <i>x</i> RANL
FZRI	2	find zero right invert	<i>x</i> 1 SW <i>x</i> LONR
GOL	2	grow one left	<i>x</i> <i>x</i> su.increv OR
GOLI	2	grow one left invert	<i>x</i> <i>x</i> su.increv NOR
GOR	2	grow one right	<i>x</i> <i>x</i> 1 AW OR
GORI	2	grow one right invert	<i>x</i> <i>x</i> 1 AW NOR
GZL	2	grow zero left	<i>x</i> su.decrev <i>x</i> AND
GZLI	2	grow zero left invert	<i>x</i> su.decrev <i>x</i> NAND
GZR	2	grow zero right	<i>x</i> 1 SW <i>x</i> AND
GZRI	2	grow zero right invert	<i>x</i> 1 SW <i>x</i> NAND
LOL	1	light ones left	<i>x</i> su.llo
LOLI	2	light ones left invert	<i>x</i> su.llo NOT
LOR	1	light ones right	<i>x</i> su.lto
LORI	2	light ones right invert	<i>x</i> su.lto NOT
LZL	1	light zeros left	<i>x</i> su.llz
LZLI	2	light zeros left invert	<i>x</i> su.llz NOT
LZR	1	light zeros right	<i>x</i> su.ltz
LZRI	2	light zeros right invert	<i>x</i> su.ltz NOT

**Table 7.20:** Visual index to the leading and trailing bit manipulation macros.

name	from	to	category
BOL	11100010	→ 11110000	brighten ones
BOLI	11100010	→ 00001111	
BOR	01000111	→ 00001111	
BORI	01000111	→ 11110000	
BZL	00011101	→ 11110000	brighten zeros
BZLI	00011101	→ 00001111	
BZR	10111000	→ 00001111	
BZRI	10111000	→ 11110000	
EOL	11100010	→ 00000010	erase ones
EOLI	11100010	→ 11111101	
EOR	01000111	→ 01000000	
EORI	01000111	→ 10111111	
EZL	00011101	→ 11111101	erase zeros
EZLI	00011101	→ 00000010	
EZR	10111000	→ 10111111	
EZRI	10111000	→ 01000000	
FOL	00011011	→ 00010000	find one
FOLI	00011011	→ 11101111	
FOR	11101110	→ 00000010	
FORI	11101110	→ 11111101	
FZL	11100100	→ 00010000	find zero
FZLI	11100100	→ 11101111	
FZR	00010001	→ 00000010	
FZRI	00010001	→ 11111101	
GOL	11101110	→ 11111110	grow one
GOLI	11101110	→ 00000001	
GOR	00011011	→ 00011111	
GORI	00011011	→ 11100000	
GZL	00010001	→ 00000001	grow zero
GZLI	00010001	→ 11111110	
GZR	11100100	→ 11100000	
GZRI	11100100	→ 00011111	
LOL	11100010	→ 11100000	light ones
LOLI	11100010	→ 00011111	
LOR	01000111	→ 00000111	
LORI	01000111	→ 11111000	
LZL	00011101	→ 11100000	light zeros
LZLI	00011101	→ 00011111	
LZR	10111000	→ 00000111	
LZRI	10111000	→ 11111000	

two instructions by composing a bitwise boolean opcode from table 7.3 (p. 107) with an increment or decrement. For leading bit manipulation (on a word's left side), the increment or decrement operates on a bit-reversed word and carries from left to right. The `su.increv` and `su.decrev` stacked unary operations of table 7.17 supply these. For trailing bit manipulation (on a word's right side), an ordinary **AW** or **SW** (add or subtract with wrap) instruction provides the increment or decrement.

Table 7.19 shows the names, lengths, descriptions, and implementations of the 40 leading and trailing bit manipulation macros. The implementation is given in Reverse Polish Notation (RPN), where  $x$  denotes the input word, capital letters indicate opcodes, and stacked unary operations are given in lowercase.

- **Brighten.** Mark the leading or trailing run, plus one additional bit, with 1s. The remaining output is 0s.
- **Erase.** Invert the leading or trailing run. Remaining output copies the input.
- **Find.** Mark the first matching bit with a 1. The remaining output is 0s.
- **Grow.** Invert the first bit after the leading or trailing run. The remaining output copies the input.
- **Light.** Mark the leading or trailing run with 1s. The remaining output is 0s.
- **Left.** Leading bit manipulation: scan from most to least significant bit.
- **Right.** Trailing bit manipulation: scan from least to most significant bit.
- **Invert.** At the very end of the operation, invert all bits.

Table 7.20 is a visual index to the 40 bit manipulation macros.

## 7.5 A reference implementation

The firmware described in this chapter is available in [Abel22b], and comes with an assembler and virtual machine for evaluation. Regression tests are supplied for the additive, bitwise boolean, compare, shift and rotate, multiply, bit permute, and mix opcodes, as well as the `CRF` and `NUDGE` instructions. Everything is written in the C programming language, except for short test programs written in the assembly language of my architecture.

## 7.6 Future work

The entire operation of an SRAM ALU depends on its firmware, and a minicomputer architecture cannot be stable until its firmware revision process is placed under strong controls. A tiny change in flag semantics, renumbering of an available feature, re-implementation of a macro, or other disruption can invalidate the architecture's operating systems, toolchains, and applications all at once. In order to minimize such disruptions, I have done my best within the bounds of my time and ability to make the first edition of the firmware as complete and correct as possible. Here is a list of remaining firmware development work:

- The stacked unary operations of table 7.17 fully determine what happens in the  $\alpha$ ,  $\beta$ , and  $\gamma$  layers, but a published specification is still needed that states the  $\theta$  and  $\zeta$  RAM operations and incoming T(emporal) flag modification needed for each stacked unary operation. This will stabilization, cross-indexing, and possible renaming of the `STUN` family of opcodes of table 7.10 (p. 136).
- The virtual machine software includes regression tests for almost all binary (two-input) ALU operations. But almost all of the simple unary and stacked unary operations need regression tests written and their implementations validated.



- A small number of ALU regression tests use C language code to compose multiple ALU instructions. Now that an assembler has been written, these tests should be rewritten in assembly language for better legibility and serviceability. An example is the PRNG of listing 7.4 (p. 128), which was tested with hopefully-equivalent C calls into the ALU, but was not tested as written in the listing.
- The stacked unary operations need examined for consistency, completeness, and semantic usefulness. For example, I am coming to believe that `su.abs2s` and `su.abs2u` operations (table 7.17) can be consolidated into one operation, and that signage should be handled by  $\theta$ ,  $\zeta$ , and/or the incoming T(emporal) flag. The `su.abs1s` and `su.abs1u` would be similarly consolidated. The assembler should adapt the `ABS` macro to the signage of the destination register, and forbid `ABS` when the source register is unsigned. `ABS` overflow can only occur when its input is  $-(2^{35})$  and the result is signed. Such overflow should set the T(emporal) and R(ange) flags.
- The 36-bit multiplication routine of listing 7.2 uses 47 instructions. My notes contain ideas for multiplying in 41 or even 35 instructions. If they turn out to be workable, they will depend on additional firmware opcodes and operations.
- 36-bit signed and mixed-signage multiplication routines should be written. This work may identify opportunities for more firmware support.
- Integer quotient and remainder need semantics, planning, and implementations. I have not started at all on these. These will surely find opportunity for new firmware opcodes and operations.
- Subroutines for floating-point arithmetic are needed early on. No one should have to upgrade firmware later because early firmware does not support floating-point math well. This work is intricate due to the need to ensure floating-point

interoperability and consistency between architectures, with aid from prevailing standardization [IEEE19]. Unfortunately, IEEE’s Microprocessor Standards Committee did not accommodate basic formats that are not 32, 64, or 128 bits or interchange formats that are not a power-of-two number of bits. Decisions will be needed as to when the extended range and precision of 36-, 72-, and 144-bit formats merit breaking interoperability. Multiple implementations may eventually emerge, wherein some use SRAM architectures at their maximum capabilities, while others round identically to non-SRAM architectures.

- Several nontrivial assembler programming projects should be undertaken, in order to identify desirable opcodes and operations for future inclusion.



## 8

# A solder-defined CPU with protected memory

In design maturity, my architecture's central processing unit lags far behind its arithmetic logic unit. The ALU is fully operational not only in logical simulations of the computations it performs, but also computes correctly in discrete event simulations of its electrical signals using an actual netlist and circuit board component placement. Track lengths, load capacitances, and signal timings are taken into account. The principal obstacle to building a physical ALU immediately is, the external circuits that it requires do not yet have functioning designs.

One external circuit on which the ALU will depend is the central processing unit, the subject of this chapter. In electrical timing simulations, the CPU can presently increment an instruction pointer, fetch instructions from the code RAM, decode those instructions, fetch operands from the twin register files, use the operands in ALU calculations, write the correct result back to the register files, and continue with the next instruction. It can also transfer immediate values from CPU instructions into registers. Conditional and unconditional branches not only enable loops, but also allow the CPU detect and respond to overrange arithmetic. Listing 8.1 shows a program that does all of these things in the simulator.

All this is promising, but it is not enough. I had hoped to name this chapter

```

; Compute xth Fibonacci number, 0 <= x <= 53

    unsigned x            ; input to fib subroutine
    imm x 53              ; seeking 53rd Fibonacci number
    jump fib              ; go compute it
back:                        ; call and return don't work yet
    halt                  ; answer in last stored register

fib:
    unsigned answer       ; output from fib subroutine
    unsigned i            ; highest computed x
    unsigned next         ; temporary new answer
    unsigned one          ; constant 1
    unsigned prev         ; previous answer

    imm i 1                ; special case when x <= 1
    cmp x - i
    jump <= small

    imm one 1              ; constant 1
    imm prev 0             ; fib(0)
    imm answer 1           ; fib(1)

loop:
    next = prev + answer   ; fib(x+1) = fib(x) + fib(x-1)
    jump +t toobig        ; arithmetic overflow?
    i = i + one            ; computed one more term
    prev = answer          ; keep previous two terms
    answer = next
    cmp x - i              ; have xth term yet?
    jump != loop           ; no, keep going
    jump back              ; yes, done

small:
    answer = x             ; fib(x) = x when x <= 1
    jump back              ; done

toobig:
    imm answer 68719476735 ; out of range indication for test
    jump back              ; done

```

**Listing 8.1:** The 53rd Fibonacci number is 53 316 291 173. Larger inputs would exceed 36 bits and return  $2^{36} - 1$ . This code ran correctly in circuit-level simulation with a 15.514 ns clock period, which in the present design is 16.11 MIPS.

**Table 8.1:** These major subsystems remain for design and test.

- preemptive multitasking
- input and output
- firmware loader

“A solder-defined CPU with memory protection and preemptive multitasking.” But three major subsystems of the architecture are missing, and multitasking is one of them. The other two are a firmware loader and an I/O subsystem. These subsystems are called out in table 8.1, and chapter 9 describes my plan to include them.

In addition to the three missing subsystems, there are a few smaller gaps in functionality that is nearly complete. Little more than cross-referencing, typing, and testing is needed to close these gaps, which are best identified in terms of opcodes that firmware hasn’t been written for yet. Although hardware is already present to support these opcodes, it can’t be finalized until firmware is written and tested successfully. These gaps are **CALI**, **CALL**, and **RETURN**, nonprivileged load and store instructions **LD** and **STO** for data memory, and privileged load and store instructions **RCM1**, **RCM2**, and **WCM** for code memory, **RDM** and **WDM** for data memory, and **RPT** and **WPT** for page table memory.

This chapter describes the portions of the minicomputer that are designed to such an extent that an electrical (not just logical) simulation for testing is available now. Table 8.2 shows a list of the opcodes described in this chapter.

The chapters ahead describe the missing subsystems in chapter 9, and a description of the design and test environment along with test results in chapter 11.

**Table 8.2:** Non-ALU instructions.

<b>opcode</b>	<b>nonprivileged instruction</b>
CALL	call subroutine
IMB	load immediate both (18 bits high, same 18 bits low)
IMH	load immediate high (18 bits high, 18 zeros low)
IMN	load immediate negative (18 ones high, 18 bits low)
IMP	load immediate positive (18 zeros high, 18 bits low)
JUMP	jump unconditionally
JUMP.EQ	jump if equal
JUMP.GE	jump if greater than or equal
JUMP.GT	jump if greater than
JUMP.LE	jump if less than or equal
JUMP.LT	jump if less than
JUMP.NE	jump if not equal
JUMP.NR	jump if R(ange) flag clear
JUMP.NT	jump if T(emporal) flag clear
JUMP.R	jump if R(ange) flag set
JUMP.T	jump if T(emporal) flag set
LD	load (read word from data memory via page table)
ORLD	OR and load (future instruction)
RETURN	return from subroutine
STO	store (write word to data memory via page table)
<b>opcode</b>	<b>privileged instruction</b>
CALI	faux call and initialize call stack
ORDM	OR and read data memory (future instruction)
RCM1	read code memory (first portion)
RCM2	read code memory (second portion)
RDM	read data memory, bypass page table
RPT	read page table
WCM	write code memory
WDM	write data memory, bypass page table
WPT	write page table

**Table 8.3:** Bill of materials for a simulated CPU that ran listing 8.1.

count	description	mfr.	part number
21	256 Ki $\times$ 18 sync. SRAM	GSI	GS840Z18CGT-250I
5	128 Ki $\times$ 36 sync. SRAM	GSI	GS840Z36CGT-250I
3	1 Mi $\times$ 36 sync. SRAM	GSI	GS8320Z36AGT-250I
1	512 Ki $\times$ 18 sync. SRAM	GSI	GS880Z18CGT-250I
43	16-bit D flip-flop	TI	SN74AUC16374DGGR
10	16-bit buffer	TI	SN74AUC16244DGGR
39	dual 2-input AND	TI	SN74AUC2G08DCUR
23	D flip-flop	TI	SN74AUC1G74DCUR
12	dual 2-input NAND	TI	SN74AUC2G00DCUR
20	dual 2-input XOR	TI	SN74AUC2G86DCUR
7	dual 2-input OR	TI	SN74AUC2G32DCUR
4	dual 2-input NOR	TI	SN74AUC2G02DCUR
3	dual buffer	TI	SN74AUC2G34DCKR
2	dual inverter	TI	SN74AUC2G04DCKR
1	80 MHz crystal osc.	Kyocera	KC7050K80.0000C1GE00
26	8-resistor array, 10 k $\Omega$	Bourns	CAY17-103JALF

## 8.1 Physical characteristics of the CPU

It may help early on to understand the scale of what I propose can be built. The CPU this chapter describes fits on a  $220 \times 220$  mm single-sided circuit board that holds 220 other components. Table 8.3 shows a list of these parts, which cost \$414.73 plus shipping when I bought them in early 2021. The circuit has 6 678 pins to solder in 1 645 distinct nets. If only two flying probes were available for a full continuity test, 1 358 868 touches would be needed.

A circuit board floorplan appears as figure 8.1. The drawing is split into two facing sheets and appears at 1:1 scale. To show exactly where the cut was made, the parts marked 374.28 and 374.3 appear on both sheets. Pin numbering proceeds counterclockwise on all components, with pin 1 shown enlarged in red. An unsplit floorplan for PDF viewing is in figure 8.2.

All components used are surface-mount and have leads, and although the task



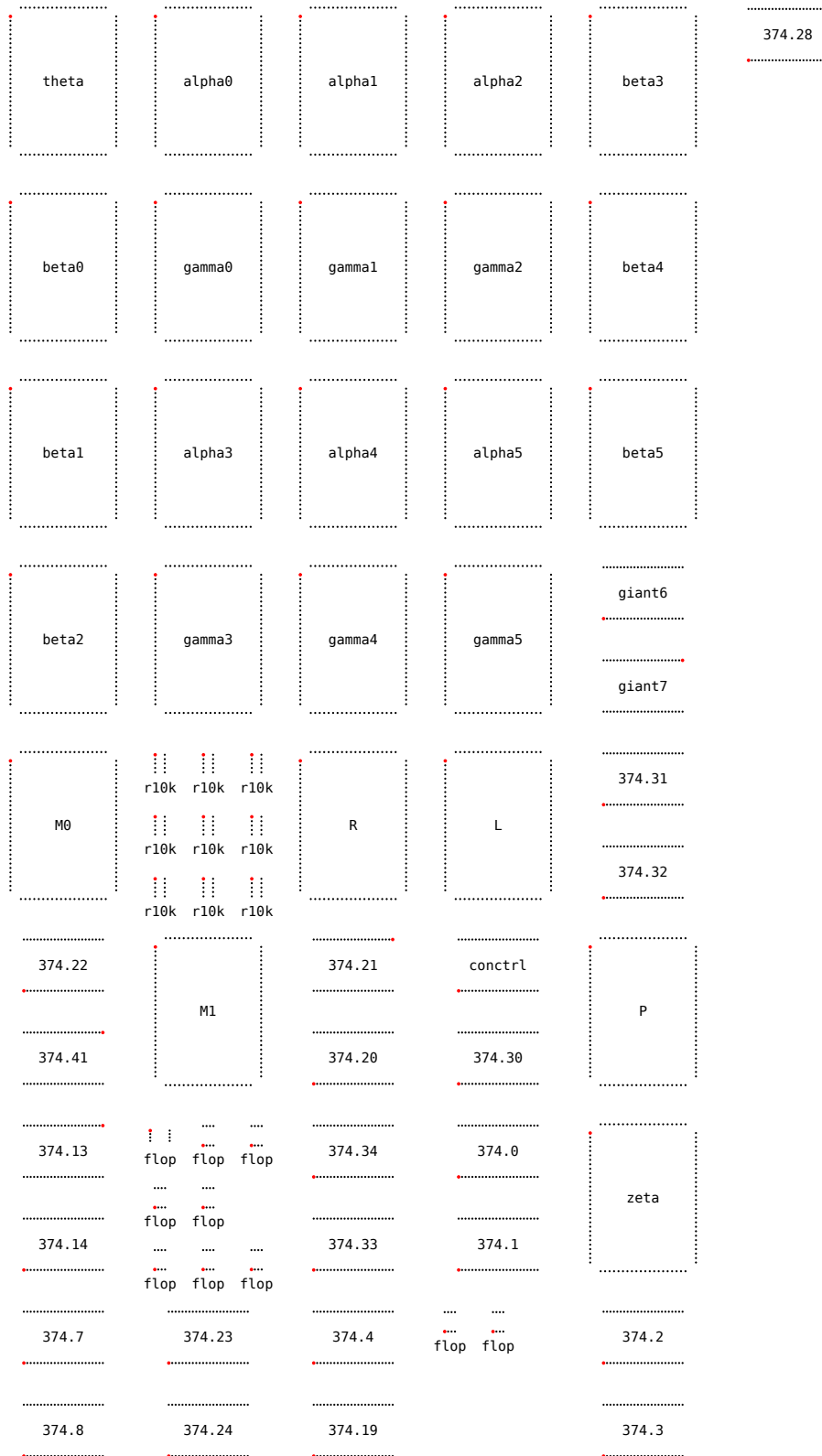
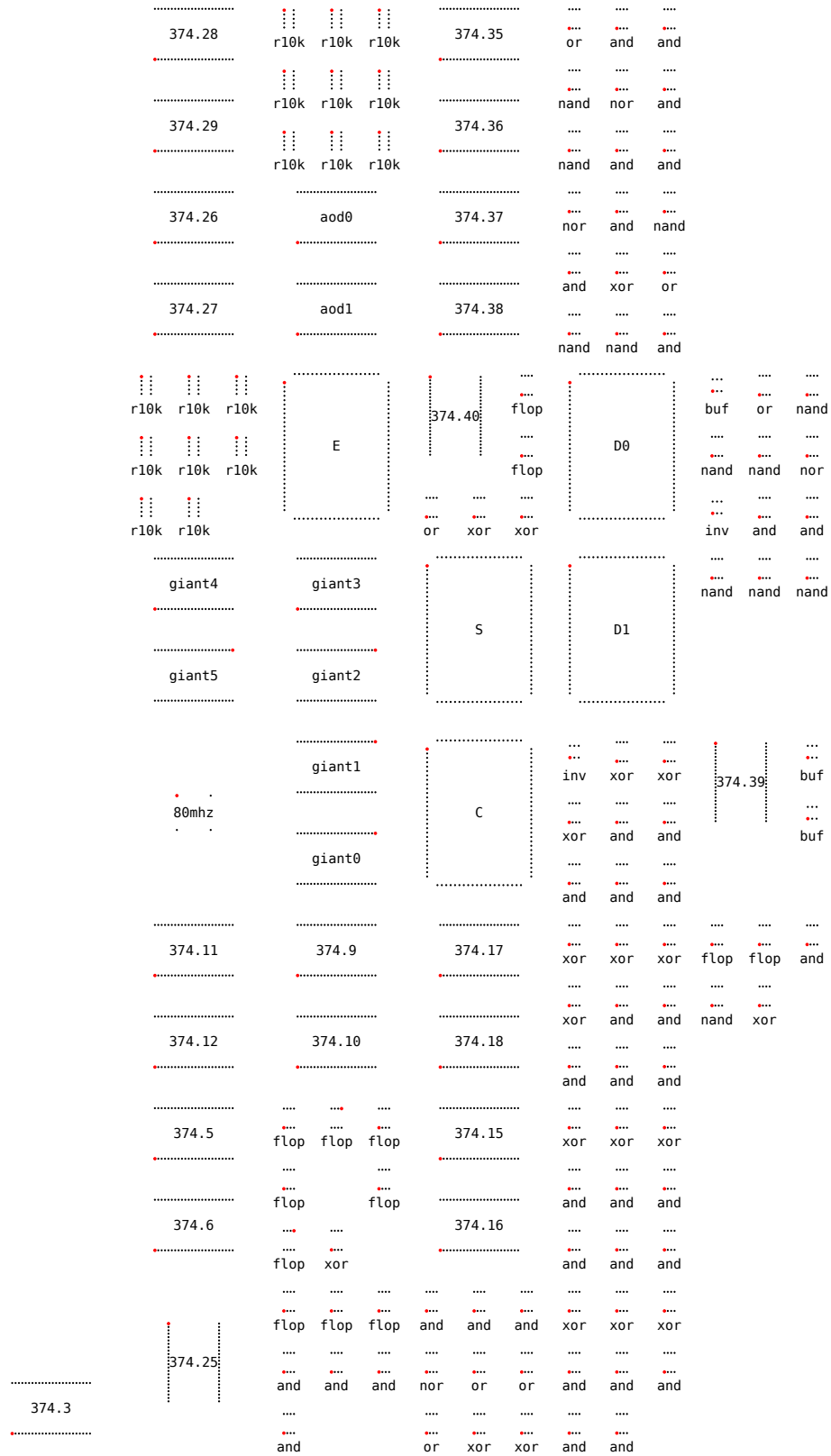


Figure 8.1: Left half of CPU floorplan. Actual size.



**Figure 8.1:** Right half of CPU floorplan. Actual size.



may look impossible to someone who has only soldered through-hole components in the past, everything can be soldered by hand. This is why there are no ball grid array components, and one of several reasons for the generous separation between ICs. The extra space also makes room for the hundreds of bypass capacitors to be added later, as well as increases the realism or difficulty of the timing simulations.

Three component sizes are prominent in the floorplan. The large ICs are synchronous SRAMs, with 30 pins on each long side and 20 on each short side, for a total of 100 pins each. The medium ICs have 48 pins with tighter spacing than the SRAM pins. These are 16-bit flip-flops and 16-bit buffers. All of these large and medium ICs have three-state outputs, a capability that is essential to the CPU's design. The small parts are resistor arrays, glue logic ICs, and a crystal oscillator. None of the small ICs have three-state outputs.

The crystal oscillator is near the center of the board. Next to it are eight buffer ICs that double the oscillator signal six times to attain 64 parallel outputs. The outputs at each level of the tree are shorted together, with the goal of having a single high-current net clock the entire minicomputer. As mentioned in section 3.4, I am more nervous about this clock than any other part of the computer. I am only somewhat knowledgeable in RF design, and the skew tolerances are very tight.

The resistor arrays are used to pull unused RAM data lines to a power rail. The RAM datasheets do not indicate that these pins can float, but because they are sometimes outputs, they must not be soldered directly to  $V_{DD}$  or ground.

The ICs marked 374... are 16-bit D flip-flops with output enable. Most of these are used to occasionally bypass the usual instruction datapath, either to support non-ALU instructions such as WCM (write code memory), or for loading firmware at power-up. Although the present design does not include a *firmware loader* that initiates and coordinates transfer of firmware into the ALU, control decoder, and so on, the flip-flops that electrically transfer the firmware into their destination RAMs are already in the netlist and on the board as drawn. This means that the capacitance

**Table 8.4:** Non-ALU RAMs visible in CPU floorplan.

legend	purpose
C	<b>C</b> ode memory
D0	control <b>D</b> ecoder bits 0–35
D1	control <b>D</b> ecoder bits 36–71
E	call stack <b>dE</b> pth
L	<b>L</b> eft register file
M0	data <b>M</b> emory
M1	data <b>M</b> emory (optional)
P	<b>P</b> age table
R	<b>R</b> ight register file
S	return address <b>S</b> tack

and track length overhead of requiring parts to load firmware into the CPU is already accounted for in the design, and is already measured in electrical simulations.

The floorplan has two “nests” of glue logic ICs on the right page. AND gates are common to both nests, but the remaining glue logic clusters somewhat according to nest. The bottom nest is characterized by having many XOR gates. It is the instruction pointer incrementer described in section 3.5.3 and partially drawn in figure 3.3.

The top nest has NOR, NAND, and OR gates that are not found in the incrementer. Many of these gates combine to make *node lockouts*, physical logic controls that prevent SRAMs and 16-bit D flip-flops from exceeding their maximum output current ratings by oppositely driving the same nets (section 8.7.3). Although short circuits can be avoided by using carefully-written firmware without further protection, I wanted safeguards to assure makers and maintainers that a misunderstood table entry in their firmware won’t permanently disable their machine. These lockouts are also intended to lessen opportunities for maliciously-written firmware to inflict hidden or delayed harm on soldered components.

In the floorplan, 20 SRAM ICs have Romanized Greek letter legends and are

easily recognized as the 36-bit ALU of chapter 5. The ten remaining SRAM ICs are introduced in later sections of this chapter. Table 8.4 offers a succinct list of what their legends in the drawing mean.

## 8.2 Machine word structure

The CPU word size is 36 bits. The rightmost bit is bit 0 and has place value  $2^0$ . The leftmost is bit 35 with place value  $2^{35}$  or  $-(2^{35})$  for unsigned or signed words respectively. Bit positions may also be written as base-36 characters, with z and 0 as the left and right extrema.

Individual arithmetic logic unit components are too small for 36-bit operands. Instead, they operate on 6-bit subwords called *tribbles*. For this reason, many assembler instructions process their left and right operands in pairs of tribbles. Tribble 5 is leftmost, and tribble 0 is rightmost.

As words pass through the ALU, they undergo a self-inverse transposition two times. These transpositions are simply a matter of wiring and do not use any active components. A word is transposed by feeding the  $i$ th bit of tribble  $j$  to the  $j$ th bit of tribble  $i$  for all  $i, j \in \{0...5\}$ . The written notation uses the top superscript: if  $\mathbf{w}$  is a word, then  $\mathbf{w}^\top$  is its transpose, and  $\mathbf{w}^{\top\top}$  is  $\mathbf{w}$  transposed twice, which is simply  $\mathbf{w}$ .

The bit positions of  $\mathbf{w}$  and  $\mathbf{w}^\top$  can also be written as  $6 \times 6$  square matrices. The transpose operation is the ordinary reflection through the main diagonal:

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

A linear notation for the transpose operation is this: If

zyxwvu tsrqpo nmlkji hgfedc ba9876 543210 are the bits of some word  $\mathbf{b}$ , then  
ztnhb5 ysmga4 xrlf93 wqke82 vpjd71 uoic60 are the bits of  $\mathbf{b}^\top$ .

## 8.3 Register organization

To lessen bottlenecks in the CPU, two copies of the register file are maintained in separate RAMs: one supplies the left argument of an operation, and one supplies the right. When writing a result to a register, the CPU updates both copies simultaneously. For almost all of this document, any reference to a register (or a number of available registers) ignores fact that there are two electrical copies, as that fact is simply an implementation detail.

The smallest widely-available, 36-wide synchronous SRAMs have 17 address bits, allowing a single IC to hold 131 072 registers. Of the 17 bits, 9 are taken from the CPU instruction word, meaning that a program can use up to 512 registers without spilling. The other 8 bits specify one of 256 programs that are ready to run. I call these ready-to-run programs *users* instead of tasks, programs, processes, or threads. I will reserve these other words to be defined by the operating system design. In order to switch from one user to another, the CPU has almost no work to do, because no registers need to be saved or restored. To switch from using 512 registers to 512 different registers, all that is needed is to clock the new user into an 8-bit D flip-flop.

The registers in this CPU are fully orthogonal, meaning that all registers have identical capabilities. In an instruction where any register can be used, truly *any* register owned by that program can be used. If you remember the distinct functions of the Intel 8086's registers, which have an accumulator, a counter, an extended accumulator, index registers, base pointer, stack pointer, several segment registers, and more, they can all be forgotten here. My CPU has one kind of register, and 512 are available at a time.

Users cannot share registers in this architecture. One consequence is that a user that is part of the operating system, notwithstanding privileges, cannot directly read or write any registers of another user that is running. But the operating system can—indirectly—access another user's registers by storing instructions that would access

those registers in the code RAM, and then branching to that location as that user. This indirect capability is the mechanism by which an operating system can initialize a user's 512 registers to zeros before invoking a program.

### 8.3.1 Register splitting and reverse subtraction

The left and right copies of a register are identical in the current architecture. If fast hardware multiplication is added to a future architecture, the 72-bit product will be stored simultaneously with the 36 most and least significant bits in the left and right register files respectively. Operations that follow hardware multiplication will need to write their code to retrieve the 36-bit halves of the product via the correct operand. For addition this is easy: the operation `product + 0` retrieves the most significant half, and `0 + product` retrieves the least significant half.

Subtraction works differently, because `5 - product` retrieves the least significant half and subtracts it from 5, but how to subtract 5 from the register instead? `product - 5` doesn't work, because that's five fewer than the *most* significant 36 bits. There are easy workarounds that use require two instructions per subtraction, but one instruction per subtraction would be preferable.

The solution is a *reverse subtract* arithmetic operation, earlier described in section 7.2.1. The syntax `5 ~- product` takes the right copy of the register file and then subtracts 5 from it. The tilde symbol `~` expresses that the arguments “reverse” before the operation. Although hardware multiplication is not available as an option yet, I included reverse subtraction at the outset for the benefit of future continuity.

## 8.4 Memory organization

The CPU's program, data, and stack memory are stored on physically separate SRAM chips and have distinct address spaces. This separation is very helpful to speed, simplicity, and security. The only drawback is that when the system has unused



memory for one of these three purposes, it cannot be repurposed for the other two.

The CPU has no cache memory for three compelling reasons. Most importantly, there are no stalls or wait states within the CPU or its memory subsystem, so there is no speed to gain using a cache. Second, a cache will not help alleviate the slow access times of DRAM, because there is no DRAM anywhere in the machine. Third, a cache will not speed the CPU's access to memory by putting a cache on the same die as the CPU, because this architecture is soldered together from separate ICs.

### 8.4.1 Data memory organization

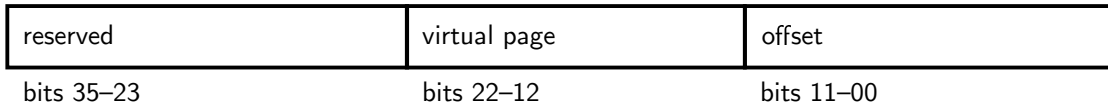
Data memory is *indirect memory* that programs can read from and write to for temporary storage as they run. Data memory is 36 bits wide and is addressable only as words, so location  $n + 1$  is 36 bits past location  $n$ , and both locations are word-aligned.

Data memory is allocated in pages of 4096 words and managed via a page table. This page table resolves virtual addresses as shown in figure 8.3 through the mapping of figure 8.4 to become physical addresses as seen in figure 8.5. The physical address format supports 28 bits each for up to two SRAM ICs. Twenty-eight bits will be ample for some time because the largest ICs on the market only use 22 address bits.<sup>1</sup>

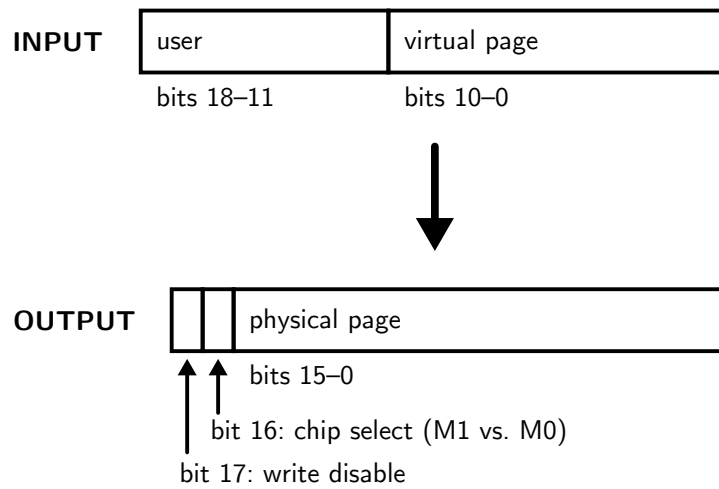
The reason for having the chip select bit separate from the physical page number is, the board supports six available SRAM sizes between  $128\text{Ki} \times 36$  and  $4\text{Mi} \times 36$ . Because the size actually used is decided by soldering the chosen SRAM into place, the board has no method of knowing which address bit should be used to switch between RAMs M0 and M1. It is not a problem that the physical addresses of the two ICs are not contiguous, because the page table can easily make their virtual addresses contiguous. The operating system can determine the installed RAM sizes by checking how addresses wrap around. The two RAM ICs do not need to be the

---

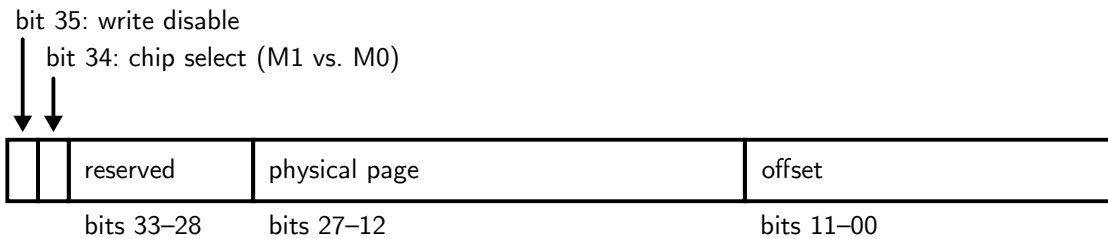
<sup>1</sup>An October 2022 catalog check search  $4\text{Mi} \times 36$  SRAMs for \$243.83 each. To fill out figure 8.5 without tapping the reserved bits, 128 SRAM ICs would be needed at a discounted cost of \$27 121.92.



**Figure 8.3:** Virtual address format for data memory. The virtual page field’s width matches figure 8.4, and can grow in the future from the reserved field.



**Figure 8.4:** Page table RAM’s input and output bit assignments.



**Figure 8.5:** Physical address format for data memory. The reserved field can be reallocated as SRAM densities improve to expand the number of physical pages.

same size.

Supporting a selection of SRAM sizes for the data memory needs careful attention given to address pin order. For any fixed memory size, both the manufacturer’s data sheet and JEDEC specification indicate that all address pins are interchangeable. But they are not interchangeable if more than one size is supported. For example, when expanding from  $128\text{Ki} \times 36$  with  $2^{15}$  rows to  $256\text{Ki} \times 36$  with  $2^{16}$  rows, the selected JEDEC packaging adds a new address bit at pin 49. For the address space to be contiguous across all supported RAM sizes, pin 49 must attach to the  $2^{15}$  address place value. Several more address pins representing further SRAM size increases have similar assignment constraints.

The page table input and output fields are shown in figure 8.4. These fields may expand in width should larger SRAM ICs for data memory become available, but they are wide enough now to support two  $4\text{Mi} \times 36$  SRAMs, which is the largest size with leads in this series today. The page table SRAM IC itself is  $512\text{Ki} \times 18$ , which is the smallest commonly-available, 18-wide size.

There is no out-of-bounds exception for memory reads or writes. Instead, out-of-bounds memory accesses are real operations on live memory to whatever IC responds modulo its size. The operating system is responsible for ensuring that all page table entries for a running program, including virtual memory that is not in use, point to a physical page that does not belong to another user. Unused table entries will have a many-to-one relationship with a physical memory page that has been cordoned off. In this manner, wayward programs will not have access to memory owned by other programs. There are no RowHammer-class security attacks against SRAM chips, so a rogue program’s ability to cause trouble is strictly limited to the memory the operating system permits it to access.

As figure 8.5 shows, a provision exists for “write-protected physical addresses.” Because the write disable bit does not connect to an SRAM address line, any physical word of memory can be reached by via both a write-disabled physical address and a

write-enabled physical address. Reads and writes via either are fully allowed, except that writes to the write-disabled addresses have no effect. By having the page table map a virtual page to a write-protected physical page, the operating system can prevent a user from writing to that page.

The CPU supports memory sharing between users at the operating system's discretion. Granting a user permission to access a page is a simple matter of adding a new page table entry for that user. The write disable bit need not be the same for the same physical page as seen through different users' page tables, so it is possible to have pages that some users can read and write, but other users can only read.

The physical address format does not include a read disable bit, which if included would have let a user write to a page for which the user does not also have read permission. I did not try hard to find a compelling use for such a feature.

I considered an alternative for memory protection that involved locking a variable number of address bits instead of having a page table. The locked-bit scheme would require all of a user's memory to be contiguous and have a number of words that is a power of two. The consequences of such a scheme are not ideal. For example, no user program would be able to use more than half of the installed data memory, because the next size up would be *all* of the data memory, leaving no memory for an operating system. Also, increasing the size of a user's memory could require relocating the memory of other users in order to provide a contiguous block.

## 8.4.2 Code memory organization

*Code memory* contains CPU instructions that are fetched, decoded, and executed in order to run a program. Code memory can also be called *program memory*. Other than this fetch-decode-execute process, which happens transparently, users have no access at all to their code memory. Enforcement of this restriction works very simply. Code memory can only be accessed via the **WCM** (write code memory) instruction and **RCM** (read code memory) instruction family. The operating system will refuse to load

these instructions in application programs. The rigid machine instruction format (section 8.5) makes it straightforward to exclude specific opcodes.

The architecture does not require, and does not offer, any further hardware protection or page table scheme for code memory. By using a relocating program loader to finalize addresses immediately prior to program execution, programs can be scanned by the program loader to ensure that all `CALL` and `JUMP` destinations are permitted for the user running. Because these instructions only come in fixed-address formats, there is no means to evade this filtering. The drawback to this approach is that pointers to functions, `setjmp`-like schemes in C, and the like are not available in this architecture. The simplicity and security gained will greatly outweigh these drawbacks in most cases.

Operating systems can easily offer linking to library code that shares the same resources and privileges as the user running. All that is necessary is to have the code in memory, and the application can simply `CALL` it directly. The program loader is responsible for allowing specific program-external addresses to be called, and disallowing calls to other addresses. These external routines cannot in themselves gain privileges, registers, or memory access that their user does not already have, but they can make calls to the operating system on the user's behalf, and the operating system may proxy additional privileges or memory access as needed.

At one point, there was a plan to segment the code memory into pages of 8192 words. Offsets within pages would be advanced by a linear feedback shift register (LFSR), and switches between pages would require `JUMP` instructions. This was because I did not think a fast enough program counter could be built using available glue logic ICs. Mercifully, I was able to design a fast counter (section 3.5.3 and figure 3.3), and the code memory organization is flat.

The code memory address space is 27 bits, and the implementation only supports one SRAM IC for code. This 27-bit limit is on account of the instruction word size for `JUMP` and `CALL` instructions, which have this format:

opcode	code address for <b>JUMP</b> , <b>CALI</b> , or <b>CALL</b>
bits 35–27	bits 26–0

The single-SRAM IC constraint presently limits code memory to 4Mi words per machine. Although this does not sound like a lot of code to run a complete system, I believe that many systems should not have more code than this. I think it would be much easier to shrink a system to fit within 4Mi words (18 Mibyte) of code than to efficaciously audit a larger system and warrant that it is free of defects.

### 8.4.3 Stack memory organization

A  $128\text{Ki} \times 36$  SRAM contains the stack memory for the CPU. Only  $64\text{Ki} \times 27$  bits of this IC are used. Its rows are accessed using the 8-bit user, indicating which program is executing, and an 8-bit call stack depth. The 27 columns correspond to the code memory's 27-bit address space. The stack memory contains return addresses only.

Rather than count in numerical order, the call stack depth changes via the successor and predecessor operations of an eight-bit linear feedback shift register (LFSR) as drawn in figure 3.2 (p. 45). The LFSR can adjust the stack depth faster and using less circuitry than an up/down counter can, due to the limited variety of fast glue logic ICs on the market. The all-zeros state of the LFSR is not valid, because its successor and predecessor are also all zeros. The nonzero states form a cycle of 255 call depths, which permit each user up to 255 nested subroutine calls at a time.

The LFSR sequence has no beginning and no end, and the bottom position of any user's call stack is not monitored and almost entirely left to chance. The two necessary requirements are (1) the stack begin at a nonzero LFSR state, and (2) no more than 255 calls are active in the same program at any time.

Exactly three instructions affect the call stack:

- **CALL** adjusts the stack depth to its successor, and then writes the address of the instruction that immediately follows **CALL** to the stack. As these are happening,

the 27-bit destination specified in the `CALL` instruction becomes the current instruction pointer.

- `CALI` (call and initialize) is the same as `CALL`, except the control decoder strobes the “preset depth” bit (figure 3.2 on p. 45) in order to guarantee the LFSR will land in a nonzero state, thereby initializing the `CALL` stack to a sane condition. `CALI` is a privileged instruction (not allowed in user code), because its return address may not be initialized and therefore may not belong to the address space of the program. Although `CALI` requires a valid destination because it does branch, `CALI` should not have a matching `RETURN`.
- `RETURN` causes a branch to the address at the top of the user’s stack, and then adjusts the stack depth to its predecessor.

With `CALI` banned from user programs (the program loader enforces this), only four possibilities exist for a program to branch via `CALL` or `RETURN`:

1. The program can `CALL` a subroutine at some code address. The program loader must enforce that the destination is either within the program that is running, or is otherwise expressly permitted (library code, operating system call, shared unprivileged code, etc.).
2. The program can `RETURN` to the instruction after an earlier `CALL` from code that the program was authorized to execute. The program loader must ensure that `CALL` is not the last instruction within an authorized block for some user. Otherwise, its matching `RETURN` could land in unauthorized, privileged code.
3. The program can `RETURN` to an address that no longer belongs to its user. The operating system must ensure that code memory is never taken away from an unprivileged program without terminating the program first.

```

    cali cali.done          ; 1. Initialize call stack.
cali.done:
    call user.program       ; 2. Run the user's program.
    jump os.terminate.program ; 3. Prevent stack underflow.
user.program:

    ; ===== INSERT UNPRIVILEGED USER PROGRAM HERE. =====

    jump os.terminate.program ; 4. Don't overrun allowed memory.

```

**Listing 8.2:** Four “bookend” instructions for call stack safety.

4. The program can underflow the call stack by using an unmatched `RETURN`, possibly into unauthorized, privileged code. The program loader must insert code to catch an unmatched `RETURN`.

Listing 8.2 shows “bookend” assembly code that the program loader can place around users programs to prevent call stack abuse.

Nothing has been said about call stack overflow. Although stack overflow is very bad in terms of program execution, it is not a security concern in the sense that it cannot cause a program to branch outside of a user’s permitted code.

## 8.5 Machine instruction format

The architecture implements four instruction formats, named after the number of fields used. For example, Format III contains three fields. The underlying basic format is that all fields are a multiple of nine bits, and the opcode field is always the nine most significant bits.

### Format I

opcode	ignored
bits 35–27	bits 26–0



Format I instructions include **CRF** and **NOP**. The CPU is guaranteed to ignore the “ignored” portion of the instruction. The virtual machine, on the other hand, sometimes uses this extra space to provide diagnostic capability such as **SAY** that is not available on the hardware. (**SAY** is implemented as a parameterized **NOP**.)

### Format II

opcode	branch target in code memory
bits 35–27	bits 26–0

Format II is for **CALL**, **CALI**, and **JUMP**. All branch targets are physical addresses: code memory is not protected by hardware and does not use a page table.

Note that code memory addresses cannot exceed 27 bits using this format. Most systems built for this architecture are not expected to approach this much RAM in the near term. Per section 8.4.2, the present implementation uses 22 bits at most.

### Format III

opcode	dest. register	immediate value
bits 35–27	bits 26–18	bits 17–0

Format III is used for moving immediate values to registers. This is done 18 bits at a time, with a worst case that 36 arbitrary bits would need three CPU instructions: **IMH**, **IMP**, and **OR**. Several cases can be done in one instruction, one of **IMB** (immediate both), **IMH** (immediate high), **IMN** (immediate negative), or **IMP** (immediate positive). See also section 8.6.3.

### Format IV

opcode	dest. register	left register	right register
bits 35–27	bits 26–18	bits 17–9	bits 8–0

Format IV is for most CPU instructions, especially ALU operations requiring a 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  $c = a + b$ .

### 8.5.1 Alternative instruction formats

For my CPU, I chose that all instructions would have the same length and stand by themselves. Having no modifying prefixes or other “instruction state” to remember between fetches makes context switching between users easier to implement. An important feature for ease of construction is that the CPU can be free to switch users after any instruction.<sup>2</sup> For SRAM minicomputer designs, I believe that this instruction homogeneity would be a near-universal feature.

I also decided that the instruction word size would match the ALU word size. This was more discretionary, and many good reasons exist to consider wider instructions. Even the four-bit advantage that 36 bits offers over 32 bits is very significant, because these four bits double the number of program-accessible registers and double the number of possible opcodes. It would be easy to modify my CPU for a larger instruction word size.

#### Reasons to expand instructions beyond 36 bits

- A user (running program) could address more registers. This would be at the expense of the number of permitted users.
- A user could address all 131 072 registers in the machine, allowing great flexibility for register handling and exchanging data between users. This need not make the system less secure, because the operating system can allocate registers to users on an as-needed basis. The only registers the hardware will let

---

<sup>2</sup>I propose an exception for reading code memory, because the situation is easy to manage. See section 8.6.2.

any program address are the registers specified in the simple CPU instruction format, so it is easy to constrain code to use only certain registers. And because all registers are orthogonal, the operating system can defer assigning register numbers to a program until it is to be loaded into code memory.

- The number of opcodes could be increased beyond 512, and/or opcode modifier bits could be supported.
- The register file could be expanded to more than two copies, so that ALU instructions could offer more operands. With four copies of the register file, an ALU instruction could have a left operand, plus a separate right operand for each ALU layer ( $\alpha$ ,  $\beta$ , and  $\gamma$ ).
- Instructions could already be partially or fully decoded at the time they are fetched. The present CPU can't add an immediate value to a register, because by the time the instruction is decoded, operands that would need to come from the instruction word have already been fetched from registers and delivered to the ALU. If the need for an immediate value could be determined by simply fetching the instruction, the ALU could receive that value on time.
- Instructions could use a greater variety of decodings.
- “New” instructions could be synthesized from existing operations without any firmware changes needed. This would involve doing much of the instruction decoding in the compiler or assembler instead of in the decoder RAM.
- An expanded instruction word size can offer backwards compatibility with non-expanded instruction words, by having the program loader expand narrow programs as they are copied to code memory. This would allow owners of “souped-up” CPUs run unmodified code that was built for ordinary CPUs.

### Reasons to *not* expand instructions beyond 36 bits

- Expanded instructions increase the firmware interface, creating more parts of the firmware that should not change between releases.
- Expanded instructions could enable unforeseen privilege escalation attacks.
- Expanded instructions will require more code memory.
- Expanded instructions could require more circuit board space.
- Expanded instructions would slow down program loading.
- Expanded instructions would require more complex assemblers and compilers.
- Expanded instructions are not assured to add enough benefit to justify use.
- Expanded instructions could make analysis of binary programs more difficult. Not difficult enough to offer new security assurances, but difficult enough to be a nuisance to programmers.
- Use of expanded instructions could increase documentation needs.

## 8.6 CPU topology and instruction cycle

Figure 8.6 is a high-level diagram of the principal data paths of the CPU. Control signals, ALU specifics, most of the return address stack mechanism, and more are not drawn. Some explanation of the drawing symbols is needed.

The lines represent numbered *nodes* in the CPU, buses that move data between sections. Some nodes are uniquely identified with the digits 0 through 5. The rectangular and triangular boxes represent components that are selectively active as computation proceeds.

### letter codes for flip-flops

**A**ddress for code reads and writes

**B**ypass page table

**C**all (save return address)

**D**estination register

**F**rom incrementer

**I**nput from i/o

**J**ump and call destinations


**iM**mediate argument

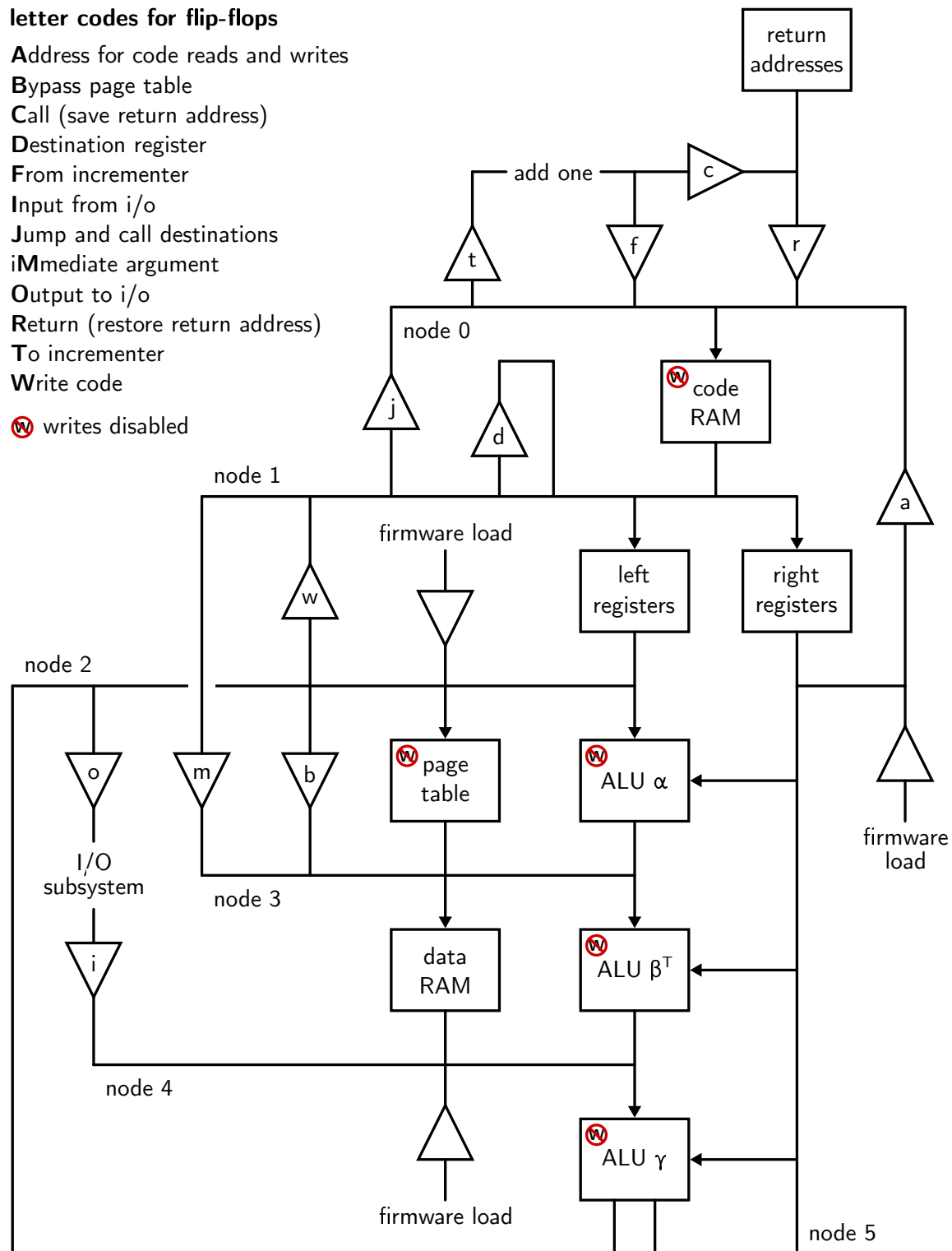
**O**utput to i/o

**R**eturn (restore return address)

**T**o incrementer

**W**rite code

 writes disabled



**Figure 8.6:** Principal data paths of the CPU.

The rectangular boxes indicate SRAM ICs. The  $\alpha$ ,  $\beta$ , and  $\gamma$  boxes contain six SRAMs each as explained in section 5.1, and the data RAM box may contain one or two ICs. The remaining boxes are singleton ICs. Arrowheads pointing into the top of each box mark address lines, indicating one-way information movement. The connections without arrowheads at the bottom of each box indicate D/Q (input/output) lines, indicating that the direction of information flow depends on timing. These directions are as follows.

- The ALU's  $\alpha$ ,  $\beta$ , and  $\gamma$  layers are used as logic elements only, and are marked with a “writes disabled” icon. These RAMs can only be written to by the firmware loader at power-up. No CPU instructions, whether privileged or not, are able to write to these ICs.
- To aid the understanding of normal program flow, the code RAM is marked “writes disabled.” But the privileged instruction **WCM** (write code memory) can write to the code RAM so that programs can be loaded to run.
- To aid the understanding of memory protection, the page table is marked “writes disabled.” But the privileged instruction **WPT** (write page table) can write to the page table to allocate memory pages to users.
- The data RAM can be written to or read from via the **STO** and **LD** instructions.
- Operands for ALU instructions are read from the registers at the beginning of each instruction, with the result of the computation written back at the end of each instruction.
- The return addresses, or *stack*, is written to and read from by the **CALL**, **CALI**, and **RETURN** instructions.

For simplicity of drawing and reading, as well as conformity with established schematic symbols, the triangular boxes can be regarded as buffers. Data enters the

buffers on the flat side and emerges at the pointy connection. As with the SRAMs, these “buffers” operate on demand by the control unit instead of continuously. In particular, the buffers have registered inputs and three-state outputs just as the SRAM ICs do. This is a roundabout way of saying the triangles are really D flip-flops with output enable, which had I used traditional schematic symbols for, would have made the drawing much harder to read.

From here, these buffers-that-are-more-than-buffers will be called flip-flops, and their usual purpose is to provide “wormholes” for out-of-path information transfers within the CPU. Ordinarily, a computation in progress proceeds from one RAM or layer of RAMs to the next in a four-clock cycle, while the flip-flops sit idly with high-impedance outputs and no rising clock edges to capture their inputs. But now and then something else may need to happen, such as a write to a code memory or input from a peripheral. Like valves in a complex network of pipes, these flip-flops allow information flow to be reconfigured as the control unit directs.

For convenience, the flip-flops have single-letter names that derive from their purpose, and are notated in writing as ff a, ff j, ff t, etc. A guide to the names is present in figure 8.6. Because the purpose of these flip-flops is to deviate from the “normal” datapath flow, a statement to describe normal flow precedes an explanation of the flip-flops and their functioning.

### 8.6.1 Instruction cycle for ALU opcodes

A majority of opcodes defined for my CPU are for ALU instructions, and they more or less have the same datapath flow though the CPU. This flow is what is meant by “normal,” and it’s only normal in the sense that it’s how most opcodes work.

The present design executes a CPU instruction every four clock cycles. To me, this high throughput seems dazzlingly efficient and elegant, but it is premature to know if four-clock instructions can be implemented reliably. The contingency, as mentioned in section 3.4, is clock skew. Here is the four-clock design in the meantime.

**Table 8.5:** A new CPU instruction starts every fourth clock cycle.

click	main activity	other activity
−3	deliver instruction pointer	
−2	fetch instruction	
−1	fetch operands from registers	decode instruction
0	ALU: $\alpha$ layer	
1	ALU: $\beta$ layer	ALU: $\theta$ propagates carries
2	ALU: $\gamma$ layer	ALU: $\zeta$ manages flags
3	write result to registers	further flag processing

The CPU counts clock cycles modulo four, with the remainder being called a *click*. The time divisions are named click 0, click 1, click 2, and click 3. For counting, click 0 begins with the rising clock edge that starts the ALU’s work at the  $\alpha$  layer. By the time click 0 is over, data will be ready to compute at  $\beta$ ’s address pins.

Table 8.5 shows the essential instruction cycle, although it may be too detailed. The short explanation is this.  $\alpha$ ,  $\beta$ ,  $\gamma$ , registers.  $\alpha$ ,  $\beta$ ,  $\gamma$ , registers. Over and over. 0 is  $\alpha$ , 1 is  $\beta$ , 2 is  $\gamma$ , 3 is registers. You try it now.  $\alpha$ ,  $\beta$ ,  $\gamma$ , registers. Look at figure 8.6.  $\alpha$ ,  $\beta$ ,  $\gamma$ , registers.

### Sleight of hand clocking the registers

There’s a little magic to the four-click CPU cycle. It’s easy to understand that  $\alpha$ ,  $\beta$ , and  $\gamma$  use one clock each, but the registers apparently *read and write* on the same rising clock edge using the same pins. How can one data pin be two different bits? There’s yet more magic, because the operand registers and destination registers are often not the same, so how can one address pin be two different bits? Section 3.2.2 explains a little bit. This is a common (but not universal) feature of synchronous SRAMs called *zero bus turnaround*, among other names.

With zero bus turnaround, two consecutive rising clock edges do the work. The address to write is electrically ready at the address pins—it comes from something



else in the circuit—prior to the first rising edge. With the first rising edge, the RAM captures its write address, but the RAM does nothing else yet. The address to read is electrically ready at the address pins—again coming from something else in the circuit—prior to the second rising edge. When the second rising edge arrives, five things happen almost simultaneously:

- The RAM captures its address to *read* at its address pins.
- The RAM captures the data to *write* at its data pins.
- Whatever circuit was driving the data pins (with information to write) goes high-impedance. (It's synchronized using the same clock as the RAM). Within a very few nanoseconds, the node will be free for the RAM to drive instead.
- The RAM initiates a read at the requested address.
- The RAM switches its data pins to low impedance. Within a very few nanoseconds, the node will contain the read data.

The above list isn't quite the whole story. There is still the matter of actually writing the data to the RAM. Internally, the RAM's memory is in a hurry to do the requested read, so the write operation is deferred using internal registers. There is also the possibility that the write and read addresses are the same, with the hazard that the data being read is not yet written. Here again, logic internal to the RAM IC detects this hazard and reproduces the still-to-be-committed data as output in the event the addresses match. So there's a lot of clever engineering in zero bus turnaround SRAMs—engineering that only involves the SRAM's interface, not the storage array itself. Zero bus turnaround adds almost no overhead to a chip.

### **Before an instruction can be executed**

Although the four-click machine cycle finishes a CPU instruction every fourth clock cycle, the start-to-finish time for any single instruction is more than four clicks. There

is some overlap. The first three rows of table 8.5 shows what happens prior to the ALU's involvement at click 0. Because the CPU counts clock cycles modulo four, clicks  $-3$ ,  $-2$ , and  $-1$  of a given instruction happen during clicks 1, 2, and 3 respectively of its immediate predecessor.

During click  $-3$ , or three clicks before the ALU begins its work, the final instruction pointer is delivered from one of three sources. This is the address at which the code memory will be read. Most often the address is one more than the previous instruction pointer. For this reason, some architectures use the term *program counter* to mean instruction pointer. But it may instead be the destination address of a conditional branch, or a subroutine return address that is popped from the call stack.

During click  $-2$ , the instruction to be executed is fetched from the code RAM. This will have one of the machine instruction formats of section 8.5.

During click  $-1$ , three pieces of information from the fetched instruction are looked up. First, control decoder RAMs D0 and D1 convert the opcode into the many control signals that are necessary to implement the opcode. Second, the left register number is looked up in the left copy of the register file, and its contents are retrieved for use as the ALU's left operand. Third, the ALU's right operand is similarly fetched.

### Branching as a potential bottleneck

One constraint on getting the four-click CPU cycle to work is that one of three approaches to branching (**JUMP** or **CALL**) may be necessary:

- **Branch approach 1.** Branches need to be taken so quickly that they behave like ordinary instructions.
- **Branch approach 2.** Branches are not taken fast enough to prevent executing the instruction after the branch, so the hardware must somehow cause any

spurious post-branch instruction to have no effect.

- **Branch approach 3.** Branches are not taken fast enough to prevent executing the instruction after the branch, so the programmer, assembler, or compiler must write all code in such a way that spurious post-branch instructions can do no harm.

My current implementation presently uses branch approach 1, which could either be viewed as having a fast enough branch implementation, or viewed as reducing the CPU speed during simulation to a point where branches work correctly. Although the code of listing 8.1 runs at 16.11 MIPS, not all `JUMP` opcodes work correctly at that speed. The problem is electrically motivated. Some branch calculations are five SN74AUC-series logic gates deep, and branch decisions have to be ready by clock 1. It remains to be seen whether the branch circuit can be improved to run at 20 MIPS without reconsidering what branching the architecture will support.

### Mitigating clock skew risk

My greatest worry about implementing a reliable machine is that clock skew and noise will be too large for the CPU's timing tolerance  $t_{KQX} - t_H$ , which is SRAM hold time subtracted from SRAM clock to data valid time (section 3.4). This tiny tolerance is 1.5 ns, and I have no prior experience or relevant study designing high-speed circuit boards. Here are four sensible approaches to this problem.

1. Secure competent assistance.
2. Model the clock tree and circuit board using an appropriate RF or mixed-signal simulator. This is discussed in section 11.4.5.
3. Increase track length from SRAM output pins to artificially increase  $t_{KQX}$ .
4. Alter the design so that when a rising clock edge causes an SRAM to read any input pins, the same rising clock edge does not cause any logic level or

impedance change at the input pins. This is likely to require changing the CPU cycle from having four clock cycles to having five.

There would be gains and losses if this option is chosen. All things being equal, the CPU would run slower and need a few more components. But all things may not be equal, because a fifth clock cycle may be able to eliminate some bottlenecks such as the branch decision calculation. If the right bottlenecks could be amortized over a fifth clock cycle, it might even be possible to increase the CPU a little beyond 80 MHz.

## 8.6.2 Memory access opcodes and routes

Opcodes that access memory, whether code, data, or page table, depart from the ALU opcodes in how data flows. Tables 8.6 and 8.7 describe the opcodes for reading and writing memory respectively and how the memory is electrically reached. Opcodes that access stack memory were described in section 8.4.3.

### Nonprivileged data memory: LD and ST0

The LD (load) and ST0 (store) instructions achieve memory protection through use of the page table. These instructions are very fast, because the page table and data RAM are co-located within the ALU and straddle the  $\alpha$  and  $\beta$  layers respectively. There wasn't another good place to put page table and data RAM anyway, because all instructions execute in just four clock cycles. The bad news is that the only addressing mode available is register indirect, because (1) the ALU is bypassed, and (2) there is no time in the CPU cycle to add an offset to an address. So if a register contains the base address of a structure, loading an arbitrary word within the structure would require a preliminary A (add) instruction to compute the address to LD.

In my dissertation research proposal, I suggested a base-plus-offset addressing mode for data memory. This mode required placing the page table after the  $\gamma$  layer of the ALU, and then the data RAM after the page table. I decided this made the

**Table 8.6:** Memory read opcodes and their circuitous routes. See figure 8.6.

opcode	description	route from and to registers
LD	load	left, page table & ff b, data, $\gamma$ , registers
RCM1	read code memory 1	right, ff a, code, ff m
RCM2	read code memory 2	(continued) ff m, $\beta^\top$ , $\gamma$ , registers
RDM	read data memory	left, $\alpha$ , data, $\gamma$ , registers
RPT	read page table	left, page table, $\beta^\top$ , $\gamma$ , registers

**Table 8.7:** Memory-write opcodes and their register-to-memory routes.

opcode	description	address route	data route
STO	store	left, page table & ff b, data	right, $\beta^\top$ , data
WCM	write code memory	right, ff a, code	left, ff w, code
WDM	write data memory	left, $\alpha$ , data	right, $\beta^\top$ , data
WPT	write page table	left, page table	right, $\alpha$ , page table

datapath too long, and that faster-executing instructions with less capability would result in a faster CPU on average. One factor I considered was the relatively infrequent need for data memory reads and writes due to the large number of available registers.

Because the offset within a page is the same for virtual and physical addresses, the page table does not process the offset. A mechanism is needed for the offset to bypass the page table so that it gets from node 2 to node 3 of figure 8.6. My current implementation uses ff b to move the offset, but I intend to eliminate ff b soon and use the ALU's  $\alpha_0$  and  $\alpha_1$  RAMs instead. This will require separating the  $\alpha$  RAMs' E1# (active-low enable) pins into two separate groups for  $\alpha_0$ – $\alpha_1$  and  $\alpha_2$ – $\alpha_5$ . In addition to eliminating a 16-bit flip-flop IC, this change will add a very limited addressing mode for reads, where the  $\alpha$ .or operation can be used with the 12 least significant bits. This will support a future ORLD (OR and load) opcode that has a base address aligned to some power of two in the left register, and an offset that is less than the lesser of that power of two and 4096 in the right register. On good days,

the new opcode may eliminate the **A** (add) instruction before certain LDs. Note that a matching **ORSTO** (OR and store) instruction will not be available, because it would need three operands (base, offset, value to write) on an architecture that supports only two operands. **ORLD** will have a privileged version **ORDM** that bypasses the page table.

Another change is coming to the data memory opcodes, because there is a prodigious blooper in figure 8.6 and table 8.7. The data for the **STO** (store) and **WDM** (write data memory) instructions originates with the right operand and is introduced via the  $\beta$  layer, but  $\beta$ 's output is transposed per figures 4.8 and 5.1 (pp. 63 and 75) in relation to the right operand and the data RAM, neither of which are transposed. This transposition makes all writes wrong. A software workaround is available, but it comes with a bloat and speed penalty. Every **STO** and **WCM** would need a **TXOR** (transposing XOR) first, changing code that looks like:

```
sto address = data
```

to read:

```
data' = 0 txor data
sto address = data'
```

In order to have **STO** and **WCM** work correctly, a 36-bit flip-flop must be added that can send the right operand to node 4 without engaging the transposed  $\beta$  RAMs. This wouldn't significantly add components, because node 4 would lose its isolation and no longer need a 36-bit flip-flop for loading firmware. So the big flip-flops are a wash. But performance may be hindered. Nodes 2 and 5 are the most connected in the datapath, with each net connecting to seven pins. The proposed change would increase node 5 to eight pins per net, which might break the CPU at higher clock speeds. Simulation will indicate whether this is a problem or not. Another possible difficulty is that repurposing the flip-flops from firmware loading (where speed may not be important) to system operation (where speed is important) may require them

to be located more centrally. Although these flip-flops are immediately adjacent to the ALU at present, a new distance constraint may make finishing the design harder when the layout needs updating.

### **Privileged data memory: RDM and WDM**

The RDM (read data memory) and WDM (write data memory) instructions work much like LD and STO, except that the page table is bypassed via  $\alpha$  so physical addresses are used directly.

### **Privileged code memory: WCM and RCM**

The privileged WCM (write code memory) instruction is used by the program loader to move instructions to execute into code memory.

I thought twice before specifying an RCM (read code memory) instruction, because confidentiality of code is not the only reason it needs to be privileged. The path is not short enough to complete in four clock cycles, so I split it into two instructions named RCM1 and RCM2. At the end of RCM1, the word has been read from the code RAM, but it's held up at ff m on its way to the registers. The principal use for ff m is the load immediate family of opcodes IMB, IMH, IMN, and IMP. These frequently-used opcodes would cause RCM2 to lose information if they separate it in time sequence from a prior RCM1. Thus RCM1 and RCM2 are only permissible in code that is not subject to preemptive multitasking. I anticipate that the only code the hardware will be able to protect from context switching will be the operating system main thread.

Other than memory testing, I do not foresee a need to read from code memory other than for instruction fetching. For this reason, I was willing to include this two-instruction instruction, the only of its kind at this time, in the architecture.

### Privileged page table memory: WPT and RPT

The WPT (write page table) instruction provides a simple implementation for the operating system to manage physical and virtual memory, including write protection of virtual pages. Operating system implementers need to remember that user programs can produce any virtual address, therefore *all* virtual pages for a user program (whether in use or not) must resolve to security-appropriate physical pages.

The RPT (read page table) instruction may be useful for memory testing, but perhaps not much else.

### 8.6.3 Flip-flops not involved in memory accesses

There are five more uses for flip-flops in figure 8.6 beyond the memory operation uses described in section 8.6.2 and its tables 8.6 and 8.7.

#### Immediate constant loading via ff m

Four CPU instructions that use Format III (section 8.5) are able to copy an 18-bit numeric constant from the instruction word to a destination register via ff m. This numeric constant is called an *immediate operand*, because it is “immediately” available from the instruction rather than requiring a register to obtain it. There are four natural things to do when an 18-bit constant is converted to a 36-bit word to store in a register:

- Use the 18 bits at face value as an unsigned integer. The 18 most significant bits will be zeros. The IMP (immediate positive) instruction implements this.
- Use the 18 bits as a signed and negative integer. Bits 18–35 will be ones. The IMN (immediate negative) instruction implements this.
- Copy the same 18 bits into both halves of a 36-bit word. The IMB (immediate both) instruction implements this.



- Use the 18 bits as the upper half of a 36-bit word. The 18 least significant bits will be zeros. The IMH (immediate high) instruction implements this. An arbitrary 36-bit integer can be placed in a register using a three-instruction sequence such as IMH, IMP, OR.

### **Input and output via ff i and ff o**

The CPU can output a word to the I/O subsystem via ff o, and receive a word back via ff i. It would be fine if an opcode is written that can do both in one instruction, but due to lack of clock cycles for the I/O subsystem to do anything, the word received back would not be in response to the word sent. Opcodes have not been named yet to support this communication, because new control signals with yet-unknown semantics and names would likely support those opcodes.

### **Branches via ff j, ff f, and ff r**

Flip-flops ff j, ff f, and ff r comprise the instruction pointer, where ff f handles straight runs of code, ff j supports CALL and JUMP, and ff r supports RETURN. This design is messy in the sense that the instruction pointer is not contained in any single register—doing so would add a fifth clock pulse to the instruction cycle, thus slowing down the machine. The instruction pointer incrementer is fed by ff t, and ff c is used to save the return address after CALL.

### **ALU destination register writes via ff d**

The address pins on the two register RAMs require operand register numbers at the beginning of an ALU instruction so that registers can be fetched, but destination register numbers at the end of the instruction so that results can be written back. This is the purpose of ff d, which saves two copies of the nine-bit destination register after the instruction is fetched. Further in the machine cycle, the code RAM outputs shut off, and ff d recalls the destination register so the ALU's result can be stored.

## Flip-flops for firmware initialization

The firmware loader needs to reach all address and data pins of the code and ALU ( $\alpha_i, \beta_i, \gamma_i, \theta, \zeta$ ) RAMs. This access is sometimes through a series of flip-flops instead of direct. Figure 8.6 shows three unlettered flip-flops for introducing firmware where no other route exists via flip-flops to their destinations.

## 8.7 Control unit

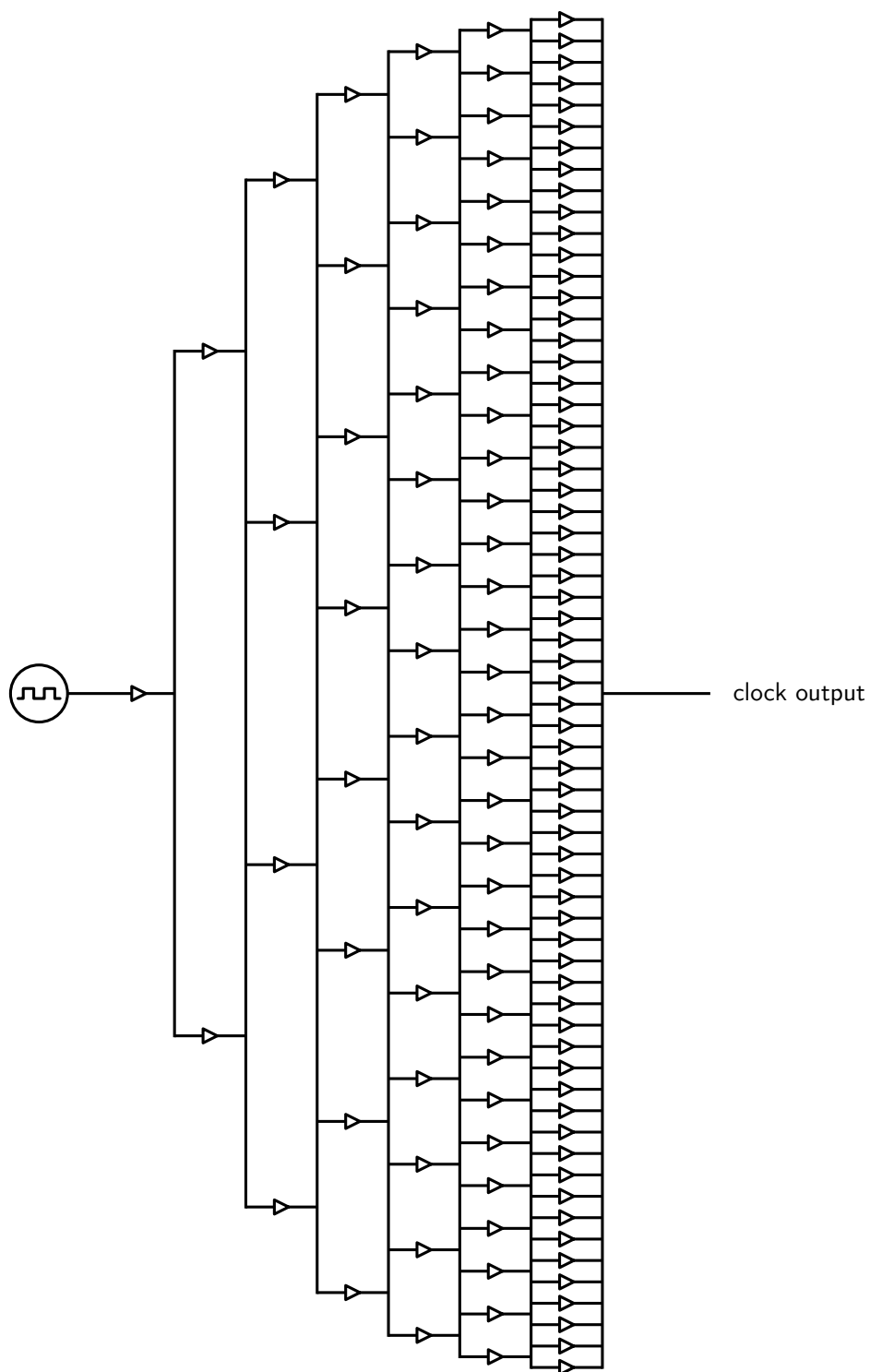
### 8.7.1 Clock driver

The clock driver was described in section 8.1 without a drawing. Figure 8.7 shows how the clock tree fans out. When placing components, I sought to have lengths nearly equal so that intersecting conductors would have the same phase. The VHF nature of the clock circuit and its UHF harmonics leave many questions unanswered as to whether (1) the circuit can coexist with the rest of the minicomputer, (2) whether clock skew across the system will be small enough, and (3) the level of effort that will be needed to comply with 47 CFR 15, the FCC's rules regarding unlicensed transmissions.

### 8.7.2 Click counter

There are four rising clock edges per CPU cycle. These rising edges have names click 0, click 1, click 2, and click 3. Two D flip-flops named clockphase0flop and clockphase1flop are wired as a two-bit counter that indicates which click applies to the *next* rising clock edge. (Because the clock edge must be acted on quickly, the counter must already indicate which click the rising edge is for.)

There is also some minor logic on the board that detects when a specific click, e.g. click 2, presently applies.



**Figure 8.7:** The clock driver uses eight 16-bit buffer ICs with stages wired in parallel.

### 8.7.3 Control decoder

#### Decoder RAM outputs

The control decoder can be viewed as a  $11 \times 72$  SRAM used in a read-only manner, although it is actually implemented as two tandem-connected  $17 \times 36$  SRAMs named D0 and D1. I opted early on to name the 72 outputs d100–d171, in order to minimize rework should three digits become necessary. These outputs are control signals that tell the minicomputer what to do and when to do it. Thus far, 40 of the 72 control signals have been allocated, with d140 through d171 reserved for future use.

The decoder’s control signals are listed in table 8.8. Their order is intentionally non-ideal from a documentation standpoint, because insertions and deletions would involve a lot of error-prone manual changes to the netlist.

The control signal list is not stable yet, so it is not extensively described here. There are a few general themes, however.

Because the  $\alpha$ ,  $\beta$ , and  $\gamma$  SRAMs never actuate on the same click, their function select bits from tables 7.11, 7.12, and 7.13 can use the same control signals d100–d105. For example, when the  $\beta$  ALU layer is active, the inactive  $\gamma$  layer RAMs will ignore  $\beta$ ’s function select bits. These nets have a fanout of 18, so the circuit includes some buffers to assure me that their control signals have enough drive.

The  $\theta$  and  $\zeta$  SRAMs never actuate on the same click, so their control signals d106–d110 are likewise shared. The fanout is only two, so no buffering is added.

Lockouts A, B, and G protect the nodes that are driven by the  $\alpha$ ,  $\beta$ , and  $\gamma$  RAMs respectively from overcurrent. For example, figure 8.6 indicates that the  $\alpha$  RAMs drive node 3, as do the page table, ff b, and ff m. From a system safety standpoint, it’s important that the control decoder not have *individual* control over these four current sources. Individual control could permit more than one current source—a logic 1 and a logic 0—per net, and perhaps lead to component damage. Instead, the control decoder outputs “advice” as to which of the four sources for

**Table 8.8:** Control decoder assigned bits.

bit name	description
d100	function select bit 0 for $\alpha$ , $\beta$ , $\gamma$
d101	function select bit 1 for $\alpha$ , $\beta$ , $\gamma$
d102	function select bit 2 for $\alpha$ , $\beta$ , $\gamma$
d103	function select bit 3 for $\alpha$ , $\beta$ , $\gamma$
d104	function select bit 4 for $\alpha$ , $\beta$ , $\gamma$
d105	function select bit 5 for $\alpha$ , $\beta$ , $\gamma$
d106	function select bit 0 for $\theta$ , $\zeta$
d107	function select bit 1 for $\theta$ , $\zeta$
d108	function select bit 2 for $\theta$ , $\zeta$
d109	function select bit 3 for $\theta$ , $\zeta$
d110	function select bit 4 for $\theta$ , $\zeta$
d111	lockout A advice
d112	lockout A other advice
d113	lockout B advice
d114	lockout B other advice
d115	lockout C advice
d116	lockout C other advice
d117	lockout G advice
d118	lockout G other advice
d119	lockout IP advice
d120	lockout IP other advice
d121	page table's $W\#$ (active low write input)
d122	branch if T(emporal) flag set
d123	branch if R(ange) flag set
d124	branch if N(egative) flag set
d125	branch if Z(ero) flag set
d126	branch if positive (neither N nor Z)
d127	invert T(emporal) or R(ange) branch condition
d128	invert ALU incoming T(emporal) flag
d129	permit ALU incoming T(emporal) flag
d130	clock flags into branch decision logic
d131	copy CPU instruction (write ff m)
d132	I/O output (write ff o)
d133	lockout Z advice
d134	lockout S advice
d135	lockout S other advice
d136	lockout E advice
d137	lockout E other advice
d138	preset call depth
d139	store N(egative) and Z(ero) flags

node 3 should turn on. Hardwired glue logic decodes this advice in such a manner that only one current source per net can be active at a time.

Because the lockout schemes are complex, numerous, tedious to write, and crucial to correct operating of the system, my implementation uses considerable automation alongside some manual recordkeeping to write lockouts into the netlists. This automation includes converting truth tables to sets of boolean expressions to match the truth tables, as well as selecting the fastest expression from each set based on datasheets for available components. These expressions are then used to automatically allocate and connect glue logic ICs to implement them correctly.

There are additional named lockouts in table 8.8 for the code RAM (C), instruction pointer (IP), call stack (S), call stack depth (E), and ALU  $\zeta$  RAM (Z). Some lockouts protect more than one node; for example, lockout A protects nodes 3, 6, and 9 from overcurrent. Specific details are in flux and out of scope for this document, which does not even identify nodes 6 and 9. There are also some trivially simple lockouts that will receive their advice from the firmware loader instead of the control decoder.

I have not numbered the advice bits within individual lockouts, so table 8.8 uses the current bit names from my notes, which are “advice” and “other advice.” They do not have place values, because their decoding identifies unnumbered physical sources of current.

After the function select and lockout advice bits, the remaining control decoder outputs support other ALU and CPU functions as marked in table 8.8.

## Decoder RAM inputs

The control decoder is a table containing 2048 rows that specify the control signals for 512 possible opcodes at each of the CPU cycle’s four clicks. This table is split horizontally between RAMs D0 and D1 to obtain 72 control signals from these 36-wide parts. Address pins 2–10 accept the opcode, and connect to the nine most significant

bits of the code RAM's output. Address pins 0 and 1, which identify which of the four clicks is current, are *grounded*.

As explained so far, these address connections have two problems. First, only click 0 is accessible, because the click select bits are soldered to ground. Second, the code RAM outputs the CPU instruction to node 1 in figure 8.6, but the instruction cannot remain at node 1 throughout all four click of an instruction. For example, during a WCM (write code memory) instruction, node 1 at some point must contain the instruction begin written instead of the instruction being executed. This means that RAMs D0 and D1 will not have useful input at their address pins while decoding all four clicks.

These two addressing problems are solved as follows. As table 8.5 shows, every click 2 is an instruction fetch, and every click 3 decodes the newly fetched instruction. As of any click 3, all 11 address bits to D0 and D1 are correct. Nine of the bits contain the newly-fetched opcode, and the two grounded bits indicate that control signals to be fetched for click 0. Many synchronous RAM models, including that used for D0 and D1, have a feature called *burst mode* that captures the address inputs and includes a wraparound counter for the two least significant bits of the address. The circuitry surrounding D0 and D1 instruct them to read the address to fetch from the address pins at click 3, and to increment that address internally for subsequent fetches at clicks 0, 1, and 2.

It is sheer coincidence that the CPU cycle of my architecture has four clicks, and that common SRAMs can count internally through four-word memory blocks. This coincidence spared some effort working around a data timing collision at node 1.

## 8.8 Simplicity and scale of the CPU

In fewer than fifty pages, this chapter describes a robust 36-bit CPU with only 220 soldered components. This CPU is one of many practical, affordable designs that

could emerge for solder-defined minicomputers. Although this CPU still requires an I/O subsystem, firmware loader, and preemptive multitasking before a physical machine would be worth using, I believe that its design evidences the merit of considering solder-defined approaches for controllers and computers for critical infrastructure.





## 9

# Forthcoming subsystems

The CPU infrastructure that chapter 8 describes needs only to be “finished” for it to run cleanly in simulation. More specifically, it is the control decoder firmware that is not finished. This firmware is generated by a C program that specifies the control signals for the four clicks of 15 families of opcodes. Five families appear to be written but may not be fully tested, and ten families remain to be written. Here is code that is tested and working for one such family:

```
case cy_alu:    // ALU operation
    m[0] = lo_a_A | lo_z_Zh | lo_g_LRh | lo_c_dest | p_no_write
           | lo_s_S;
    m[1] = lo_a_Ah | lo_b_B | lo_z_Zh | lo_g_LRh | lo_c_dest
           | lo_ip_next | p_no_write | lo_s_S;
    m[2] = lo_z_Z | lo_g_G | lo_c_C | p_no_write | lo_s_S;
    m[3] = lo_ip_jump | lo_z_Zh | lo_g_LR | lo_c_dest | save_inst
           | p_no_write | lo_s_S;
    break;
```

Put another way, the CPU as described in chapter 8 is about 100 lines of code from full and working simulation, although writing assembly language test cases and being ready to correct netlist errors are corequisites.

In contrast, table 8.1 lists three essential sections of the minicomputer for which no circuit designs exist. The base functionality from chapter 8 needs to be working in simulation before these further subsystems—preemptive multitasking, the firmware loader, and input and output—can be incorporated.

## 9.1 Preemptive multitasking

Informally, *preemptive multitasking* is a mechanism where an operating system can divide a CPU's attention between more than one program that is in memory and ready to run. The preemption process temporarily freezes whatever program is running after a short moment, and unfreezes the next program so it can run. My architecture refers to these programs as *users*, because they share the use of the CPU. The duration of a short moment is kept small, so that whatever programs are waiting to run take on the appearance and characteristics of running simultaneously.

All chapter 8 functionality needs to be in place before preemptive multitasking can be designed, because the preemption (stopping one program) and context switch (switching to another program) is particularly invasive. The entire CPU, comprising thousands of pins spread over 450 cm<sup>2</sup> of circuit board, needs to set aside what it was doing in a coordinated manner and pick up something else. This chore's intricacy is temporal as well as spacial. The CPU works on two instructions simultaneously as table 8.5 (p. 203) shows, and a context switch must atomically separate them. The many spacetime boundaries and topologies need to be completely specified before multitasking can manifest.

The existing design has an excellent start on preemptive multitasking, in that the CPU registers, return address stack, and page table entries are already segregated by user. This *user* is represented by an eight-bit number, so there can be up to 256 users. Changing the user number in a single eight-bit flip-flop is all that is needed for the CPU to use a different user's registers, stack, and page table.

I thought that a similar scheme would work for the instruction pointer and flags. There would be a RAM with all of the users' instruction pointers, and a RAM with all of the users' flags. In fact, with 5 currently defined flags and a 27-bit address space for code, one 36-wide RAM could hold both lists and have extra space. But the mechanism would be cumbersome and add a clock cycle to features that are already

crowded for time. The four clock cycles to execute an instruction would need to be five. Furthermore, there is not another place in the architecture where a RAM is read from or written to at the same memory location with every CPU cycle, let alone both read and write with every CPU cycle.

There are resources in figure 8.6 (p. 200) that can be used to save and restore the instruction pointer and flags between users. The context switches would not be as instantaneous on account of needing to do the memory transfers, but they can be fast enough that their duration does not matter. My dissertation research proposal anticipated this possibility where it specified that context switches that take as long as five CPU cycles would be acceptably fast.

Context switch interrupts can be raised such that the instruction pointer for the instruction to come back to is ready at node 0. Rather than add a RAM to store it, this address can be moved via `ff t` and `ff c` to the return address stack. This would reduce the stack allowance from 255 calls to 254, which is still plenty deep for an architecture that claims to not support recursion. The CPU flags can be stored alongside the return address in the same 36-bit word, because the code address space is only 27 bits.

One problem appears to exist, in that the identified path to save the instruction pointer adds one, leading to unpredictable one-instruction “holes” in executed code when context switches occur. The circuit has no accessible hardware that can reverse this addition, and 35 ICs would be needed to add a decrementer. But there is a perfect answer. Figure 3.3 (p. 46) shows the first eight bits of the incrementer circuit, and the top of this drawing is a net labeled “logic 1.” That net triggers the increment, so bringing a logic 0 from the control decoder is all that is needed to protect the resume address from modification.

The next puzzle to solve is grabbing control to do the context switch. My thought for a while was to use a 27-bit flip-flop with grounded inputs available to write node 0, thereby forcing a branch to some operating system code with special opcodes that

would do the necessary user, instruction pointer, and flag swaps. I now think that a different approach would use less hardware, while letting the operating system yield from whatever address it chooses, instead of having to resume at a hardwired address such as all zeros.

My suggestion is to branch the context via the control decoder instead of the code RAM. Thus far, RAMs D0 and D1 only use 11 of their 17 address bits, so their total content is just 4096 doublewords. By adding a 12th address bit to RAMs D0 and D1, a parallel copy of the firmware is easily switched to that can have the control signals needed to initiate the context switch. Also, because the parallel copy includes every click of every opcode, the firmware for context switching can be customized on a per-opcode basis if necessary. I anticipate that a 13th and possibly a 14th address bit would also be added so that the control decoder can issue a long enough sequence to finish the context switch.

There is an interesting metacontext in using the control decoder to switch context. First, there are no CPU instructions or opcodes that participate in the context switch. There would likely be instructions that identify the next user (privileged) and set a flag that the CPU should yield (nonprivileged), but the control decoder would make it happen. This would make CPU cycles the correct unit of duration for context switches rather than instructions, because instructions would not run during context switches. Another distinguishing characteristic is that the control sequence extends beyond the duration of one CPU cycle. According to my reasoning in section 7.1, this extended sequence is the first proposed use of *microcode* in my architecture.

Having established how multitasking may work, a mechanism for preemption is up next. This would be a binary counter that sets the yield flag every time a preset number of CPU instructions has passed. Some arbitration would be included to avoid conflicts with a software-initiated yield, in order that the intended user is switched to and no request is lost. The possible settings for the number of CPU cycles between context switches and the granularity of those settings can be decided when the timer

is designed. So can whether or not a mechanism for reclaiming unused time will be offered.

A minimal-hardware approach to the timer could use a 16-bit LFSR and switch context every 65 535 CPU cycles. This would impose very low overhead, possibly taking 5 CPU cycles to switch to the operating system, 7 cycles for the operating system to indicate if the time has come to switch users, and 5 cycles to return control to the correct user.<sup>1</sup> This would slow the minicomputer down by 1 out of 3 855 instructions, or about 260 PPM. Although this minimal-hardware scheme would only provide a fixed interrupt rate near 305 per second assuming the system runs at 20 MIPS, a counter within the operating system can achieve a lower effective rate, subject to the fixed granularity of the underlying interrupts.

## 9.2 Firmware loader

When power is applied to the minicomputer, the ALU's 20 RAMs are empty of knowledge and devoid of ability to compute logic or arithmetic. The two control unit RAMs are also empty, leaving at least 40 control signals on sabbatical leave. The code RAM contains no operating system and no code that can load one. The instruction pointer, in addition to being unknown, may be in any of four locations. And these are best-case predictions, because instead of zeros in the various RAMs, there may be garbage instead.

The firmware loader is the answer to these many problems. Its task is to move the binary tables that guide the minicomputer's operation data from nonvolatile storage into the 23 SRAMs where they are needed. Of these SRAMs, 22 will have their contents frozen by the firmware loader prior to setting the instruction pointer and starting the CPU.

Like preemptive multitasking, a complete CPU implementation through chap-

---

<sup>1</sup>I believe these counts to be conservative, and that the implementation would do better.

ter 8 is requisite before a firmware loader’s main design effort can begin. But the dependency is not for the same reasons. The firmware loader doesn’t have to interrupt complex processing, and its connections into the CPU are through designated “firmware load” flip-flops as in figure 8.6 (p. 200). Once past those flip-flops, the addresses and data from the loader need to find their way through the CPU’s many other flip-flops in carefully-orchestrated combinations to reach their destination RAMs.

For example, to load firmware into the ALU’s  $\alpha$  layer, the data at node 3 has to come in through node 2’s firmware load flip-flop, move through ff w to node 1, then through ff m to node 3, and stay there. Then the firmware load flip-flop is used a second time to set the address bits for the left operand on node 2. The right operand’s firmware load flip-flop has to set up node 5. The  $\alpha$  function select (operation) inputs have to be set via the a firmware loading flip-flop for the control decoder. The propagate and carry outputs from  $\alpha$  have to be written at the same time, and that node is supplied via another firmware load flip-flop that doesn’t appear in figure 8.6 (p. 200). Then the encoded range output for  $\alpha_5$  needs preset via another firmware load flip-flop that isn’t drawn. Only when all these preconditions are in place can the firmware loader send a write signal to the  $\alpha$  SRAMs. That signal will cause the first word to be written to the six  $\alpha$  RAMs. They will need another 262 143 words written into them. So the firmware loader’s connectivity to the CPU is straightforward enough, but a final and accurate netlist—all of chapter 8—is prerequisite to determine a sequence for the firmware loader to issue.

Because the firmware loader must load more than five million words from non-volatile storage to SRAM, one of its implementation challenges will be testing. My implementation takes one second to simulate 210 clock cycles (about 52 instructions) based on the chapter 8 netlist. If loading a word of firmware takes one clock cycle, a complete startup would take about 30 hours to simulate. But the nonvolatile storage is read no faster than one bit per clock cycle, and all the routing needed is in addition to that. A full startup would take months of CPU time to try once. The firmware

**Table 9.1:** Data exchange flip-flops from the firmware loader to the CPU.

name	width	description
ff boi	36	<b>b</b> eta <b>o</b> utput <b>i</b> nit
ff cdi	8	<b>c</b> arry <b>d</b> ecision and carry summary <b>i</b> nit
ff dci	72	<b>d</b> ecoder <b>i</b> nit
ff eri	5	<b>e</b> ncoded <b>r</b> ange <b>i</b> nit
ff fli	7	<b>f</b> lags ( $\zeta$ RAM) <b>i</b> nit
ff gfi	7	<b>g</b> amma's <b>f</b> lags <b>i</b> nit
ff loi	36	<b>l</b> eft <b>o</b> perand <b>i</b> nit
ff pci	12	<b>p</b> ropagate-carry <b>i</b> nit
ff roi	36	<b>r</b> ight <b>o</b> perand <b>i</b> nit

loader's design will need to allow simulations to parallelize dependably.

To reach the address and data pins on the SRAMs it initializes, the firmware loader needs to attach to 207 nets as shown in table 9.1. As section 8.6.2, ff boi will probably be removed, leaving 171 nets to be driven by 24 eight-bit flip-flops. This comes to 12 ICs, because each IC has two 8-bit flip-flops. Because the 24 flip-flops have separate clock and output enable lines, they can in principle be consolidated down to eight nets on the firmware loader side. Whether to reduce to 8, 36, or some other number of nets will depend on the flow approaches chosen.

In addition, the firmware loader will need to generate a number of control signals. At this time, these are known to include the **W#** (active-low write enable) pins for the  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\theta$ ,  $\zeta$ , D0, and D1 RAMs, as well as the **E1#** (active-low enable 1) pins for D0 and D1. I expect that more will be added.

### 9.2.1 Option 1: Purchased complex logic

The firmware loader represents somewhat of a chicken-and-egg problem, because its objective of moving data into a processor calls for a processor of some kind to do it. This device need only have a few GPIO (general-purpose input/output) pins for 2.5 V logic. Only outputs would be used.



Many premanufactured devices exist that would allow a commodity computer to load firmware into the CPU. A very lightweight approach would be a USB UART with GPIO pins, such as FTDI's FT232H series of ICs. This would add almost no hardware beyond what chapter 8 already includes. All further development challenges would jump through the USB cable to firmware loading software running on the external system. It's a great plan from a development and test standpoint, but my solder-defined minicomputer would not be freestanding and always need something external to help it power up. Perhaps as bad, the solder-defined machine's lifecycle would be capped to the longevity, code, and recordkeeping of the external computer that supplies the firmware.

An internal option would be to use one of many microcontrollers that offer GPIO, such as the RP2040 from Raspberry Pi Ltd. As of November 2022, this IC can be purchased for one dollar. This part has a few drawbacks. As purchased complex logic, it may in principle contain exploitable defects. But accessing the device to exploit these defects can be made difficult.

- Firmware loading happens via unidirectional flip-flops, so a running solder-defined system will not be able to alter anything concerning the RP2040.
- The firmware loader's circuitry beyond the RP2040 can include a flip-flop that locks out future changes, preventing the RP2040 from touching the solder-defined system once the CPU is running.
- Apart from its outgoing connections into the transfer flip-flops, the RP2040 would ordinarily be air-gapped, making a remote exploit rather improbable.
- The RP2040 does not include a clock with battery backup, making a calendar-triggered exploit less probable.

A potential concern with the RP2040 is that it may contain internal writable state that could enable some kind of usage-metered triggering of an exploit. The RP2040

uses an external serial memory that should power up in write-protected mode as a precaution.

Another concern about using a microcontroller like the RP2040 is that any firmware, operating system, and applications present on a system used to bootstrap a solder-defined computer should be open-source and audited. This would vastly expand the system's design and maintenance scope without a large enough return on that investment.

On the whole, a firmware loader based on a purchased microcontroller might not be a terrible decision. But I would prefer to use either of the solder-defined approaches in the following two sections. Either can be viable and may instill more confidence in the overall minicomputer.

### **9.2.2 Option 2: Hardwired logic after NOR flash**

A careful look should be taken at what data geometries and control signal sequences are likely to work well for firmware loading. It may be helpful to write a simplified, non-standard microcontroller with GPIO into the electrical simulation, and try some firmware loads to get a sense of how much assistive logic a firmware loader would need to implement.

Some models of serial memory ICs that contain NOR flash, such as the one I purchased (section 3.6.1), can be configured to read their contents out at completion of power-up without sending any command to do so. The device I bought can keep up with an 80 MHz clock, which happens to be my targeted clock speed for the minicomputer. Up to four bits per rising edge can be configured, so in theory the entire 128 Mibit flash can be read out in 0.84 seconds. It may be safer to divide the clock rate by two in case a substituted flash IC does not assure correct operation at 80 MHz, or someone tries to clock the CPU faster than 80 MHz. Extending the boot time another 0.84 seconds is unlikely to harm much.

The bits retrieved from the serial memory would be moved into shift registers

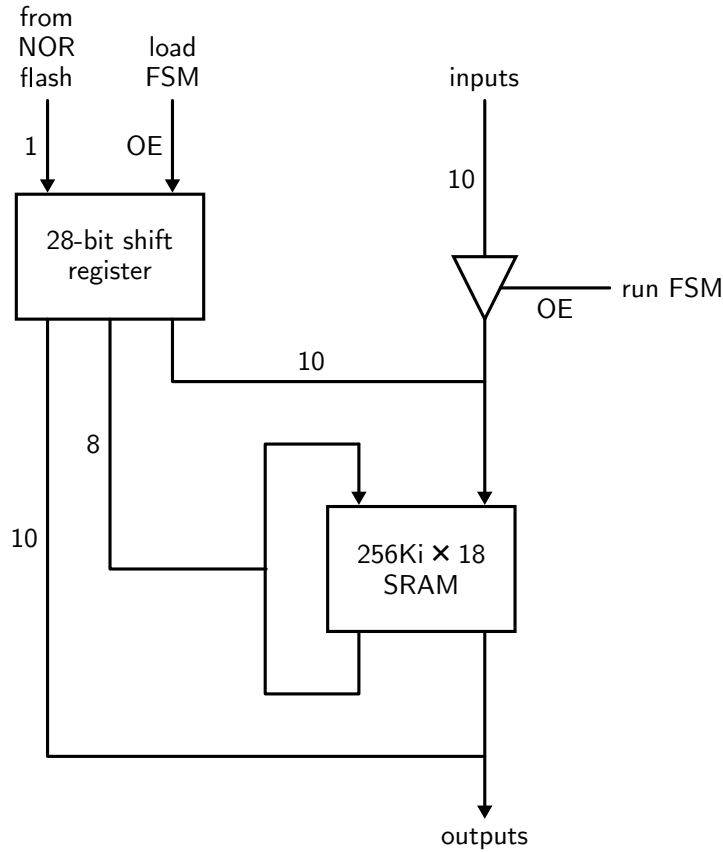
implemented with yet more dual eight-bit flip-flops. Some mechanism would decide when the registers are sufficiently loaded. At that point, the information in the shift registers is interpreted by hardwired logic as representing either a command that sequences some control signals, data to move to one of the table 9.1 flip-flops, an address to load into an address counter, or a count to move into a repetition counter.

Firmware loader counters need not increment in sequential order, so a linear feedback shift register (LFSR) is likely to be used instead. If a counter is for repetitions, its only requirement is that the counter signal after the required number of clock cycles has passed. A repetition counter would be useful for tasks such as interpreting the next 255 words from the serial memory as data to move instead of addresses or commands. An address counter may increment its low eight bits through 255 nonzero values so that address need not be reloaded. The all-zero case can't be incremented by an Galois LFSR, so it would use its own address load. This scheme amortizes to one address load per 128 words written to SRAM, so it does well.

A complementary or alternative means to increment address counters is to use the SRAMs' burst mode and ADV input pins. Burst mode may not turn out to need any fewer components than just tacking two more bits onto an address LFSR would need, so it may be better to leave the SRAMs' burst mode unused.

Although it would be bad for the serial memory to have any read errors that find their way into the CPU's implementation, it may be desirable to have some form of synchronization marking within the stream coming from the serial memory. Other than resynchronizing and continuing with the next valid address, I am uncertain what corrective action could be taken.

It may be helpful to compute a NOR flash checksum as firmware loads, although for 36-bit words this would require a lot of board space for XOR gates. An alternative would be to give the operating system read-only access to occasionally verify the serial memory via the I/O subsystem. Electrical assurances that the device stays read-only, possibly by never taking the device out of "autoboot" mode, would be important.



**Figure 9.1:** A single-RAM finite state machine with hardware to load its firmware. Freestanding numbers indicate wire multiplicities.

### 9.2.3 Option 3: Finite state machine after NOR flash

A single SRAM can be hacked into a small controller that implements a deterministic *finite state machine*, or *FSM*. Figure 9.1 shows an example with 10 input bits, 10 output bits, and 8 bits of internal state. Only the address (top) and data (bottom) pins of the RAM are drawn; additional control logic and wiring are needed.

The FSM of figure 9.1 obtains its firmware via a 28-bit shift register that surrounds it. Constructing the shift register uses a pair of dual 8-bit flip-flop ICs. The drawing looks problematic in that the eight fed-back nets appear to constrain the firmware being input to cases where eight of the output bits are identical to eight of the input bit. But there is no such constraint, because the SRAM write process

captures the address and data pins at different times. This allows the shift register to correctly align the bit stream for both the address and data pins.

Transfer of firmware from a serial NOR flash serial memory to the SRAM requires many shifts and writes. For simplicity understanding the drawing, the serial memory is configured to output one bit at a time.<sup>2</sup> The transfer process for each word needs the order of certain bits rearranged within the serial memory so that their clocked positions align with the diagram. The sequence to transfer each word is:

1. Clock  $A_0$ – $A_{17}$  from the NOR flash into the shift register.
2. Clock  $D_{10}$ – $D_{17}$  from the NOR flash into the shift register.
3. Clock  $D_0$ – $D_1$  from the NOR flash into the shift register.
4. Strobe the address for the write into the SRAM.
5. Clock  $D_2$ – $D_9$  from the NOR flash into the shift register.
6. Strobe the data for the write into the SRAM.

During the firmware transfer, “load FSM” is true and “run FSM” is false, surrounding the SRAM with the shift register outputs. When the transfer is complete, “load FSM” is false and “run FSM” is true, permitting the SRAM’s address lines to see the FSM’s most recent state (8 bits) and new inputs (10 bits) in order to look up its next state (8 bits) and next outputs (10 bits).

Control signals to load the FSM RAM (write enable, clocks, etc.) would be generated from a small amount of logic that can either be fully hardwired, or benefit from the serial memory’s presence to generate a sequence of control signals through a widened shift register and some glue logic.

The above process may be called *boot phase 1*, which loads only the single SRAM at the heart of the finite state machine. At the end of boot phase 1, the FSM is able

---

<sup>2</sup>The four-bit transfer mode would need a widened shift register and alignment bits added to each word to keep in sync with the 18-wide address and data nodes at the RAM.

to sequence far more complex logic. This logic will continue reading bits from the serial memory and transferring firmware to the 23 SRAMs in the CPU that need it. This process where the FSM loads the other 23 SRAMs is *boot phase 2*. As the final steps of phase 2, the FSM locks out all further firmware writes for the remaining time the power will be on. Then the FSM forces the CPU into a sane state via any control signals necessary, starts the CPU at location zero in the code memory, and terminates the FSM. The code memory will contain a small bootloader that was placed there by boot phase 2 and written in the minicomputer's instruction set as defined in chapters 7 and 8 and appendix B. Running this bootloader from the code memory is *boot phase 3*, which goes to the I/O subsystem to load an operating system from a storage or network device. The operating system's own initialization, run using code brought in via the I/O subsystem, is *boot phase 4*.

The firmware loader's finite state machine can be useful for implementing loops in the firmware transfer process. If enough internal states are available, the FSM table can implement loops directly. If there aren't enough internal states, the FSM's external wiring (input and output bits) can interface with a simple presettable counter such as an LFSR with some logic to detect when the sequence is finished.

The allocation of input, output, and internal state bits in figure 9.1 is for illustrative purposes and probably would not match the distributions of actual firmware loader designs.

#### **9.2.4 Option 4: Parallel NOR flash finite state machine**

Rather than use an SRAM IC as a finite state machine as in option 3, a parallel NOR flash IC might be used. The advantage would be the elimination of boot phase 1 and a possible saving in components. But there are many drawbacks:

- NOR flash has longer access time than SRAM, which would slow the boot process and probably makes the clock circuit more complex.

- The NOR flash for the FSM would require programming prior to soldering.
- Inspection of the NOR flash contents would be impractically difficult after assembly.
- Data retention in the soldered NOR flash is not forever, meaning that resoldering will be required every so many years to keep the computer working reliably.
- A NOR flash FSM violates the design goal of having all persistent state in one place.
- Parallel NOR flash ICs tend to be only eight bits wide, which may prove to be too limiting. The workaround would be to add *another* parallel NOR flash memory alongside it, as if two wrongs make a right.

## 9.3 Input and output

Attaching an I/O subsystem to a CPU is the necessary and sufficient requirement to build a computer. If I had to choose one peripheral that I couldn't live without on every machine, it would be the most accurate real-time clock I could afford. Unfortunately, factory lead time for Maxim's DS3231 can exceed 80 weeks as of October 2022, and a prominent distributor has about 120 000 on order.

As with multitasking and the firmware loader, the I/O subsystem can't be designed until chapter 8's preconditions are met. Here again, its dependence on the underlying CPU being ready is for a different reason. Unlike preemptive multitasking, the interconnect between the CPU and I/O is extremely simple—a few signals from the control decoder, and flip-flops from and back to the ALU. Unlike the firmware loader, the I/O subsystem has no particular dependence on the CPU design or its netlist. This time, the issue is that testing the I/O subsystem will require the ability to run programs of considerable complexity on the CPU.

Many I/O controller ICs have been introduced to the market, and it would be no challenge to connect one to the CPU via `ff o` and `ff i` (figure 8.6) and call the system finished. But I am seeking stronger security assurances for the I/O subsystem all the way from the CPU to the point where each peripheral attaches. Solder-defined controllers with open-source firmware and open-source device drivers can be feasible and affordable at transfer speeds of tens of millions of bits per second per wire.

My expectations for an I/O subsystem for the near term are as follows:

- Peripherals that adhere to the Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I<sup>2</sup>C) bus specifications will be supported. These will be the only data transfer mechanisms in the initial design.
- Both SPI and I<sup>2</sup>C support multiple peripherals on a single bus, sacrificing device isolation in favor of compactness, expandability, and flexibility. This offer will be declined. The I/O subsystem will prioritize security instead, so that no peripheral can detect another peripheral, eavesdrop on another peripheral, change another peripheral's data in flight, or block another peripheral's exchange of data.
- Each serial bus will attach to at most one peripheral.
- Each serial bus may have jumper pins or DIP switches to configure its connection point for I<sup>2</sup>C (open drain) or SPI (push-pull).
- Each serial bus need only be one bit wide per direction. Some peripherals such as serial memories can double or quadruple their transfer rates by using a minimal number of extra wires. I decline to make this an expectation for early implementations.
- Each serial bus will have exactly one interrupt request line. It will be maskable, active-low, and not shared with any other serial bus.



- Each serial bus will have exactly one outgoing device select line.
- A minicomputer will have at least eight serial buses. (Anyone who has set up a desktop computer with four USB connectors or a LAN with a four-port hub should understand why.)
- Serial bus transfers will be half duplex, in the sense that the buffer RAM will never simultaneously contain data to read and write from the bus.
- An I/O controller will be a simple bit-banging finite state machine that can only exchange data with one serial bus at a time.
- A minicomputer should (but may decline to) have two I/O controllers so that transfers between two peripherals can be expedited.
- Each serial bus should (but may decline to) multiplex to both I/O controllers, in order that the minicomputer can transfer between any two devices efficiently. Note that this configuration only enables one such transfer to happen at a time.
- Each serial bus will have and be confined by hardwired logic to its own dedicated memory for a transfer buffer. The confinement will not be escapable by malicious I/O controller firmware or malformed serial data.
- Each serial bus will have and be confined by hardwired logic to its own dedicated memory for its finite state machine firmware. The confinement will not be escapable by malicious I/O controller firmware or malformed serial data.
- The I/O subsystem and its serial peripherals will have no access to the CPU's registers, code memory, data memory, return address stack, or page table.
- Isolation of peripherals is intended to be strong but not absolute. The design will not consider side channel threats such as overvoltage, overcurrent, radiofrequency interference, compromising emanations, microphones, explosives,

quantum entanglement, postdoctoral scholars, or TARDISes.

I predict the I/O controller design will include these components and sections:

- A  $128\text{Ki} \times 36$  **buffer RAM** will temporarily store bits received from the serial bus, as well as temporarily store bits to be sent to the serial bus. For the purpose of this list, the 36-bit node that connects to the buffer RAM's data pins is termed the **parallel node**.
- An eight-to-one **multiplexer/demultiplexer** or *mux/demux*, likely implemented using tristate buffers, connects the I/O controller to one of eight serial buses on a one-at-a-time basis.
- A 36-bit **shift register** made from dual eight-bit flip-flops attaches to the parallel node. The shift register uses both parallel in serial out and serial in parallel out modes to exchange between the mux/demux and the buffer RAM.
- A 14-bit presettable **address counter** made from dual eight-bit flip-flops will drive the buffer RAM's address bits. The address counter will be a linear feedback shift register, so the all-zero address will not be available for buffer storage. It will not matter if the buffer contents are not in numeric order, because storing and loading will use the same address sequence. Only 14 of the buffer's 17 address bits engage in counting, because the other three bits identify which serial bus a block of memory belongs to.
- **Gateway logic** to connect the CPU's *ff o* and *ff i* to the parallel node without contention so the CPU can read and write the buffer RAM.
- A **finite state machine SRAM** that can be read, written, and have its internal state set by the CPU. Like the buffer RAM, the FSM RAM will be partitioned into eight blocks so that each serial bus is serviced by one block.

- A presetable **bit counter** LFSR to loop over 1 to 36 shift register positions.
- One or two presetable **general counter** LFSRs to support FSM firmware loops. The FSM may also use some of its internal state to unroll short loops.
- **Bus drivers** are firmware within the FSM that implement I<sup>2</sup>C and SPI data exchanges. These drivers can be custom-generated in the CPU for infrequently-changed parameters such as clock polarity, clock phase, and delay loops for slower peripherals. Frequently-changed parameters, such as the number of words to transfer, are also “hardcoded” at known offsets within a driver. The CPU will be able to update just the parameterized words of a driver prior to each I/O transfer.
- **Device drivers** are not part of the I/O controller. They are CPU code that synthesizes and interprets the data stream to and from a serial peripheral. Returning to the DS3231 as an example, a person writing a device driver would read the real-time clock’s datasheet. In contrast, a person writing its underlying bus driver would read the Inter-Integrated Circuit (I<sup>2</sup>C) specification.

It would have been very easy to provide some GPIO (general-purpose input/output) lines to the CPU without so much as a UART, attach I/O devices to those lines, and write some demonstration code to claim that the minicomputer “works.” But I have been around too long for this. I remember trying to write an assembly language program for the Motorola 6803 to communicate 300-baud serial data one bit at a time when I was about 15. The 6803 has a UART, but the computer it was in precluded its use. Someone bought me terminal software as a gift before I could finish my program, but I at least got a 6803 assembler written. I don’t want to go back to that time, wasting a perfectly good CPU on bit-at-a-time level changes.

As my budget grew, I owned machines with working 16550 UARTs. They had a 16-byte buffer, but my recollection is that I didn’t bother and wrote code that was

backwards-compatible with the 8250 and its single-byte buffer instead. It was already better than no UART at all. But I don't want to go back to that time either.

The I/O subsystem as this section envisions is not all that might be done, but it can transfer almost 72 Kibytes while the CPU does other things. The CPU will have to transfer all communication between the I/O subsystem's buffer and the CPU's data RAM, but the transfer can be 36 bits wide. The transfer may turn out to be faster than `memcpy`, which is typically thought of as fast code. The reason an I/O transfer can be faster than an ordinary memory copy is that with `memcpy`, two pointers must be incremented per word transferred. But when one side of the copy is the I/O subsystem, the address counter's LFSR does the I/O side increments, so the CPU is spared those add instructions.



# 10

## Fast parallel multipliers

This chapter is a nearly-verbatim copy of a preprint I wrote to close the gap, should it become necessary, between the ALU of chapter 5 and an ALU with a fast multiplier. Information is presented here about carry-skip adders and fast multipliers that considerably surpasses their descriptions in sections 4.4 and 4.9.

### Abstract

Reflow soldering can readily build robust CPUs solely from commodity static RAM (SRAM) and trivial glue logic, potentially avoiding trust and transparency concerns associated with the use of purchased microprocessors, ASICs, PLDs, and FPGAs. This chapter shows how hardware multipliers for such CPUs can be constructed out of separately-packaged SRAM ICs. This knowledge is new, because SRAM multipliers differ architecturally from multipliers built from basic gates. SRAM designs are aided by offset-binary representations of signed subproduct terms, by generalizing carry-skip addition to accept more than two addends, and by integrating carry-skip addition of subproducts simultaneously with carry-save addition. Arbitrarily large factors can be multiplied quickly using hierarchical carry-skip methods, although component counts will be correspondingly high. Tables of component and clock cycle counts for SRAM multipliers of common word sizes are presented. Open-source software is available to

generate and verify efficient multiplier designs.

## Keywords

ALU, backdoor, binary multiplication, carry-skip adder, computing with memory, CPU, hardware security, lookup table, LUT, maker movement, malware, manufacturing, minicomputer, multiplier, offset binary, open source hardware, privacy, ROM, semiconductor supply chain, solder-defined hardware, SRAM, surface-mount technology, trusted system.

## 10.1 Background

In 1977, researchers at University of Illinois, Urbana-Champaign built a 24-bit multiplier from 90 ICs on a  $381 \times 457$  mm ( $15 \times 18$  inch) circuit board [Stenzel77]. What's remarkable about this experiment is its heavy reliance on ROMs as logic elements, both for parallel computation of subproducts, and for adding the subproducts in parallel to obtain the final product. Building multipliers from ROM was not a trend of that time and did not catch on subsequently. But as a one-shot prototype, the idea was novel. 36 pieces of  $256 \times 8$  ROM multiplied four-bit subwords taken from both factors in parallel to obtain 36 eight-bit subproducts. These 288 bits were added in parallel using two IC types. 30 ICs were adders fashioned from  $1024 \times 4$  ROMs, each accepting five 2-bit addends with maximum sum 15. Another 12 ICs were commercial four-bit adders with carry input, with maximum sum 31. Two layers of these adders reduced the problem to a point where a carry-lookahead adder could produce the final product using 12 ICs. Total propagation delay was measured and reported to be 200 ns.

This chapter is an update, extension, and generalization of the earlier parallel multiplication work with ROMs. The scheme used 45 years ago needed too many ICs

containing too many transistors to be competitive then. Progress and demand soon brought stand-alone multiplier ICs, as well as integrated multipliers on microprocessor dies. By the end of the 1980s, whatever need that once existed to solder multipliers together out of smaller parts had passed. Nor was there ever an advantage to building multiplier ICs that used ROMs internally, because direct computation required far fewer transistors and computed products faster than table lookup. But recently, widespread fear concerning the VLSI supply chain has sparked my interest in having a fresh look at discrete-component multipliers.

An emerging concern for VLSI is that complex ICs may be subject to design defects or backdoors, and measures for inspection and audit of these chips are neither practical nor supported by manufacturers. One approach for providing a “supply chain firewall” may be to forgo use of complex and/or proprietary ICs, and instead to build CPUs and other complex logic from simple, generic parts. This vertical supply chain integration and simplification of individual components would not be practical for most applications, but for specific critical uses that need low to modest computing power, this strategy may have merit. My research of late has been designing an open-source minicomputer that contains no purchased complex logic (not even in silicon), can be assembled with maker-scale tools, and permits visible-scale inspection after assembly.

Surprisingly capable computers can be built without using microprocessors, FPGAs, PLDs, or other purchased complex logic, but there is one requirement that can’t be met without VLSI. That need is main memory. If a computer needs a million bits of primary storage, it’s either going to have over a million parts (tubes, relays, or whatever), or it’s going to have integrated circuit RAM. Consequently, a key assumption is that although computer designs might avoid using complex VLSI such as microprocessors and FPGAs, they can’t avoid using static RAM (SRAM).

SRAM may be inherently more resistant to exploitable defects than are microprocessors and other complex parts. For one thing, SRAM is generic, has multiple



sources, and is standardized by JEDEC. SRAM is simple and unlikely to contain the kinds of exploitable surprises commonly found in dynamic RAM or microprocessors. SRAM is volatile and forgets its contents when powered down. SRAM's address space is flat and consistent, ensuring that erasures and overwrites are dependable and irreversible. SRAM I/O timing is tight, increasing the difficulty of data tampering via on-die backdoors. SRAM dies employ regular cell patterns, making it harder for a manufacturer to conceal backdoors. SRAM ICs tend to use older process technologies such as 90, 65, or 45 nm [Private21], and therefore are easier to study than denser ICs. The low cost of small SRAMs also facilitates destructive inspection and testing of samples from purchased lots.

In this chapter, logic implemented by RAMs is always combinational, never sequential. More simply, RAMs are being used in lieu of ROMs. When power is applied, a loader moves firmware into the RAMs. Once the RAMs have their firmware, further writes to them are disabled, and only then does the operating system boot and run. The reason to choose RAM over ROM is speed: cheap SRAMs offer access times of 5.5 ns today, while contemporary ROMs come nowhere close. The disadvantage to using RAM is that many additional parts—flip-flops with three-state outputs—are required to move firmware into the RAMs.

If SRAM ICs are trustworthy for use storing data, they are probably even more trustworthy for implementing logic via lookup tables. SRAMs for storage hold large amounts of data at rest, but SRAMs used as logic see only tiny amounts of data at a time. Moreover, SRAM used as logic contains firmware only, and never records any of the data being processed.

One end product from my minicomputer work is a design for a robust 36-bit arithmetic logic unit (ALU) that uses just 20 synchronous SRAMs and trivial glue logic as components [Abel22b]. Its operations take three clock cycles to add, subtract, compare magnitude, compute minimum or maximum, shift or rotate by any number of bits, reverse the order of bits in a word, advance a linear feedback shift register,

hash a word for an associative array, or do a 36-bit cipher round. A CPU based on this ALU would need just four clock cycles to fetch instruction operands from registers, use the ALU, write the result to a register, and decode and fetch the next instruction. I confirmed this timing by circuit-level simulation of short programs. My clock goal for a working machine is 12.5 ns, which is 80 MHz or 20 MIPS.

In contrast to these fast operations, 36-bit multiplication to obtain a 72-bit product on my ALU takes 47 instructions. This isn't necessarily a problem, because the design is for general purpose use, not image processing or scientific computing. Much of computing doesn't need a lot of multiplication, and the multiplication that it does need is usually to compute offsets of array elements. This entails multiplying an array index by a small element size, a task my ALU already can do in three instructions if (a) the product will not exceed 36 bits, and (b) one factor is a constant from 0 through 63. The 36-bit ceiling is sufficient, because present SRAM ICs offer 24 address bits at most. So with arrays taken care of, faster multiplication hardware may not be needed.

Adding a fast multiplier to my minicomputer would add 49 RAMs to the design. The *latency*, or time between factor and product availability, would be five clock cycles. That's much faster than the status quo of 188 clock cycles (47 instructions), but the ALU would grow from 20 RAMs to 69 RAMs. Many applications cannot benefit from this added cost. Without this added hardware, a 20 MIPS implementation would already surpass 400,000 long multiplications per second, which is roughly comparable with early versions of Intel's i486.

This chapter is about the contrary case, where someone wants an SRAM CPU that can multiply faster. The approach is not unlike historical parallel multipliers, but SRAM's conciseness in expressing logic supports previously unreported optimizations.

Section 10.2 clarifies a brief list of symbols and terms.

Section 10.3 introduces parallel subproduct generation in SRAM multipliers, with attention given to factor partitioning, accommodation of optionally-signed factors,

and preparation of signed subfactors for summation by unsigned adders. For most multiplier configurations, signed factors do not add to latency or component count, and flexible signage does not add to latency.

Section 10.4 explains SRAM parallel summation principles, including arbitrary-geometry (carry-save) multi-operand addition, carry-skip multi-operand addition, and their simultaneous application to wide summands. Search techniques are suggested to minimize the latency and component count of generated circuits. Multi-layer carry-skip addition, which has potential use for word sizes larger than 64 bits, is discussed. Data and trends are presented for various SRAM sizes and multiplier configurations, with an observation that the best SRAM size to use for most multipliers is the smallest size available.

Section 10.5 restates this work’s main contributions.

The explanations and data in this chapter are supported by an open-source tool [Abel22a] for producing and verifying netlists and firmware for SRAM multipliers. RAM sizes from 2–63 address lines and 2–63 data lines are supported. The left and right factors can range from one bit to thousands and need not match. Input signages need not match and can be unsigned, signed, or flexible-signage. There also are options for solution search strategy, optimizations, batch processing, multithreading, and validation by simulation. Written in Rust 2018, the software has no external dependencies.

This chapter mentions  $256\text{Ki} \times 18$  SRAMs in several explanations, because this is the smallest widely available size today for synchronous parts. Its corresponding size for asynchronous parts is  $64\text{Ki} \times 16$ .

## 10.2 Notation and definitions

$a \cdot b, a * b, (a)(b), ab$ . Scalar multiplication of  $a$  by  $b$ .

$a \times b$ . Rectangular dimensions of  $a$  by  $b$ .

*Cycle.* Interval between consecutive rising clock edges. For contemporary synchronous parts, consider 12.5 ns.

*Either signage.* Signage specified at time of use.

*Fixed signage.* Ability of a factor input to accommodate only unsigned or only signed words.

*Flexible signage.* Ability to compute with either signage.

*Latency.* Number of cycles a multiplier takes to convert factors into their product. Equal to number of SRAM layers.

*Parallel multiplier.* Hardware that computes products by splitting individual multiplication problems into small pieces that execute simultaneously. This older term does not suggest the presence of more than one multiplier in a system [Dadda65].

*RAM.* Implied to be SRAM in this chapter.

*Sign bit.* Most significant bit of a signed (sub)word. All sign bits have negative place value.

*Signage.* Designation of a word as signed, unsigned, or *either*. Not to be confused with *sign bit*.

*$M \times N$  SRAM.* SRAM with  $\log_2 M$  address lines and  $N$  data lines. Its capacity is  $MN$  bits.  $M$  is always a power of two. Suffixes  $Ki$  and  $Mi$  denote multiples of  $2^{10}$  and  $2^{20}$ .

*SRAM.* Memory IC used as a substitute for ROM for its high speed and point-of-use programmability.

*Subfactor.* Portion of a factor in a parallel multiplier.

*Subproduct.* Product of (ordinarily two) subfactors.

## 10.3 Generation of partial products

In a parallel SRAM multiplier, the multiplication process is split into time-consecutive sets of tasks, where all tasks within a set execute simultaneously. Each set is imple-

mented as a layer of SRAMs that do not depend on each other's result, so that all SRAMs within the layer can actuate simultaneously. Subproducts are generated in parallel by the first layer of SRAMs, then summation of the generated subproducts is done using as many subsequent layers as are needed to tally the eventual product.

For evaluating designs, the latency of the multiplier is the number of SRAM layers the computation passes through. If the SRAMs are synchronous, the time required to multiply is the clock period multiplied by the number of layers. No control signals are needed within the multiplier, other than a clock signal that is common to all of the SRAMs.

### 10.3.1 Unsigned partial products

The simplest parallel  $M \times N$ -bit multiplier uses the method of grade school long multiplication, except in base 2. Assuming  $M \leq N$ , subproduct generation will use  $MN$  AND gates, and will need to be followed by some method of adding  $M$   $N$ -bit numbers. The array of bits to sum form a parallelogram. We see this shape in the three addends when multiplying  $101_2$  by  $11010_2$ , where  $M = 3$  and  $N = 5$ :

$$\begin{array}{r}
 \phantom{00000}11010 \\
 * \phantom{00000}101 \\
 \hline
 \phantom{00000}11010 \\
 \phantom{00000}00000 \\
 + \phantom{00000}11010 \\
 \hline
 10000010
 \end{array}$$

The preceding example may be considered to have 15 one-bit subproducts, because two-input AND gates are used as the basis operation for subproduct generation. But gates with more inputs can be used, provided that they produce the desired subproducts. When ROM ICs first appeared, [Johnson72] illustrated an 8-bit multiplier that partitioned its factors into four-bit subwords and computed their subproducts using four  $256 \times 8$  ROMs. The bit positions looked like:

	AAAABBBB	
*	CCCCDDDD	
	EEEEEEEE	(DDDD * BBBB)
	FFFFFFF	(DDDD * AAAA)
	GGGGGGGG	(CCCC * BBBB)
+ HHHHHHHH		(CCCC * AAAA)
	(16-bit product)	

The benefit of using eight-input ROMs over two-input ANDs is, only 28 sub-product bits remain to be added instead of 255. It's easy to generalize this scheme for today's RAMs and arbitrary factor lengths  $M$  and  $N$ . These lengths will have to be partitioned, with the hard constraint that every pair of subfactors needs to fit within the input bits of one RAM. If a  $32 \times 32$ -bit multiplier is built using  $256\text{Ki} \times 18$  ICs, the simplest partitioning uses 8-bit subwords, and two inputs will go unused on each chip. It looks like:

AAAAAAA	BBBBBBB	CCCCCCC	DDDDDDD
*	EEEEEEE	FFFFFFF	GGGGGGG
	HHHHHHH		

Although multiplier designs should seek as much computation as practical from each RAM, it's inevitable that some inputs will go unused as seen here. There are 16 subproducts of 16 bits each, so the total number of subproduct bits will be 256. Here is another partitioning:

AAAAAAAAA	BBBBBBBBB	CCCCCCCCC	
*	DDDDDD	EEEEEE	FFFFFFF
		GGGGGGG	HHHHHHH

The subword lengths are 10, 11, 11 and 6, 6, 6, 7, 7 for these factors, so RAMs for their pairs will use 16, 17, or 18 input bits. There turn out to be 256 subproduct bits, as with the previous partitioning, so the effort to sum subproducts is comparable between these examples. On the other hand, the firmware increases from 1 048 576 words of 16 bits to 2 293 760 words of 16, 17, and 18 bits, and that's a small negative for this partitioning. But I would still choose this partitioning with three and five

subfactors instead of four and four, because only 15 RAMs need to be bought, soldered, and powered instead of 16. Thus, the main heuristic for generating subproduct circuitry is, employ as few RAMs to produce as few subproduct bits as possible.

The second factor has other partitionings, such as 7, 7, 6, 6, 6 instead of 6, 6, 6, 7, 7. But it may help to use the larger subword size for the least significant bits, because (for this example) one more bit of the product will be known immediately on subproduct generation. There will be one less place value that carries need to propagate through later.

Parallel subproduct generation can be extended to build multipliers that accept more than two factors simultaneously, although the number of SRAMs needed will grow proportionally to the product of the input lengths. For example, a 3-input  $N$ -bit adder would need  $O(N^3)$  parts. Nevertheless, [Kobayashi81] illustrates a 3-input, 4-bit multiplier that uses eleven  $256 \times 8$  ROMs.

### 10.3.2 Signed partial products

Two's complement multiplication [Booth51, Wallace64, Baugh73, Hatamian86] is comparatively easy with SRAM, because much complexity gets abstracted away in the firmware. In the unsigned byte  $abcdefgh$ , where letters indicate bits,  $a$  has place value  $2^7$ , and the others have place values  $2^6$ – $2^0$ . If instead the byte is signed, the only change is that  $a$  has place value  $-2^7$ . The distributive law allows us to bring the negative place value along with any bits attached to it. For example,

$$abcdefgh \cdot ijkl = 16 abcd \cdot ijkl + efgh \cdot ijkl$$

whether  $a$ 's place value is positive or negative. This means we can freely break the multiplication into a sum of partial products, even though the place value of some bits may be negative. The subproduct RAMs would still have to “know” the signage of their inputs for their results to be correct. For example,  $1000_2 \cdot 0111_2$  will be either  $00111000_2$  or  $11001000_2$  depending on whether the factors are unsigned or signed.

If all we do is that much—keep track of the signage of all subfactors and subproducts—then signed addition can directly combine the subproducts into the final product. This works, but borrows now need to propagate from right to left in addition to carries. The strain on SRAM capacity is immense, because each borrow or carry in several-operand addition uses four or more bits, and every input bit added to a RAM requires its capacity to double. Also, it would be convenient to not have to reason about signed arithmetic for more of the circuit than necessary.

The method of [Baugh73], later improved by [Hatamian86], uses bit complements and subproduct rearrangement in such a manner that the unsigned sum of all subproducts will equal the correct signed product when truncated to the correct width. These intrinsic operations are perfect when elementary gates are used as logic elements, but a method for SRAM multipliers can reflect SRAM's greater capabilities. In the case of SRAM, all that is necessary is to exaggerate a little.

Here is a method for signed multiplication. When subfactors  $x$  and  $y$  are given to a RAM, instead of having the RAM look up the product  $xy$ , have it look up

$$f(x, y) = xy - \min_{x, y}(xy)$$

instead. This exaggerates  $xy$  by just enough that it can't be negative. It doesn't matter if neither, either, or both subfactors are signed. The result will honor

$$0 \leq f(x, y) \leq \max_{x, y}(xy) - \min_{x, y}(xy)$$

and will never be negative. The extrema here are merely constants derived from the width and signage of  $x$  and  $y$ .

As an example, consider an  $8 \times 8$  signed multiplier using four  $256 \times 8$  RAMs. The eight-bit factors partition into four-bit pairs  $a : b$  and  $c : d$ , where  $-8 \leq a \leq 7$ ,



$0 \leq b \leq 15$ ,  $-8 \leq c \leq 7$ , and  $0 \leq d \leq 15$ . The product desired is

$$\begin{aligned}(a : b)(c : d) &= (16a + b)(16c + d) \\ &= 256ac + 16ad + 16bc + bd\end{aligned}$$

where scaling by 256 and 16 are just left shifts of 8 and 4 bits. Four RAMs are needed to compute

$$\begin{aligned}f_0(a, c) &= ac + 56 \\ f_1(a, d) &= ad + 120 \\ f_2(b, c) &= bc + 120 \\ f_3(b, d) &= bd\end{aligned}$$

because the minimum possible products of  $ac$ ,  $ad$ ,  $bc$ , and  $bd$  are  $-56$ ,  $-120$ ,  $-120$ , and  $0$  respectively. But the column-appropriate sum from these RAMs overstates the product  $(a : b)(c : d)$ , because they compute

$$\begin{aligned}&256(ac + 56) + 16(ad + 120) + 16(bc + 120) + bd \\ &= 256ac + 16ad + 16bc + bd + 18\,176 \\ &= (a : b)(c : d) + 18\,176\end{aligned}$$

instead. We can correct this by exaggerating by yet another  $47\,360$ , because  $18\,176 + 47\,360 = 2^{16}$ , and the subproduct summation process will only keep the rightmost 16 bits.

Thus the subproduct RAMs introduce a constant bias of  $+18\,176$  to the multiplier, but an additional bias of  $+47\,360 = 1011\,1001\,0000\,0000_2$  would correct it. This correction can be applied anywhere within the multiplier circuit, as long as it does

get added in. Noting that  $47\,360 = 256 \cdot 185$ , we can replace

$$f_0(a, c) = ac + 56 + 185 = ac + 241,$$

and suddenly the entire scheme is perfect. Although RAM  $f_0$  will routinely overflow past eight bits, its excess is beyond the 16-bit width of the product. To summarize this example, an eight-bit signed multiplication problem is distributed to four RAMs to determine subproducts, and even though only one RAM's ( $f_3$ 's) subproduct is individually correct, the column-appropriate unsigned sum of the four RAM outputs will give the correct signed product every time.

### 10.3.3 Mixed-signage multipliers

It's common for CPUs to offer two multiplication instructions per supported word size—one for unsigned factors, and another for signed factors. A single multiplier can provide both services by having the control unit specify signage of its factors at the time of use. Signage only needs consideration for multipliers that produce doubleword results, because the less significant word does not change. CPUs should also have instructions for mixed-signage multiplication. The cost to accommodate mixed signage is negligible, but many CPUs overlook it.

Introducing flexible signage to traditional multipliers (no SRAM) can be done by sign extending by one bit. For example, a 36-bit either-signage multiplier can be built as a 37-bit signed multiplier. If the factor bits are  $L_0 \dots L_{35}$  and  $R_0 \dots R_{35}$ , then  $L_{36} = L_{35} \wedge C_L$  and  $R_{36} = R_{35} \wedge C_R$ , where control signals  $C_L$  and  $C_R$  are true when their respective factors are signed and false otherwise. Components serving only the two most significant bits of the product can be omitted, because the intended product is 72 bits, not 74.

The sign extension scheme for traditional multipliers can be used with SRAM multipliers, but two AND gates would have to be inserted in the datapath, perhaps

**Table 10.1:** Number of  $256\text{Ki} \times 18$  SRAMs needed for various multipliers.

signage	bits per factor						
	31	32	33	34	35	36	37
unsigned $\times$ unsigned	40	41	43	39	41	41	50
signed $\times$ signed	40	41	43	40	41	41	50
either $\times$ either	39	40	44	41	41	49	50

The cost to specify signage at the time of multiplication is akin to adding an input bit. The seeming improvement from 33 to 34 bits comes with a latency increase from four to five clock cycles.

increasing the clock period for the whole CPU. This is unnecessary, however, because signage can be supplied directly with the most significant factors. Continuing the 36-bit example, RAMs can accept  $C_L$  with  $L_{35}$ , and  $C_R$  with  $R_{35}$ , instead of computing intermediate values  $L_{36}$  and  $R_{36}$  via AND gates. The firmware tables are written as if  $L_{36}$  and  $R_{36}$  were available.

The one limitation of directly supplying signage control signals to subproduct RAMs instead of sign-extending first is, the signage must always accompany the most significant bit that it modifies. This increases the lower bound on permissible RAM size: fixed-signage multipliers can be built from  $4 \times 2$  SRAMs, but flexible-signage multipliers require at least  $8 \times 2$  SRAMs were one control signal appears, and at least  $16 \times 2$  SRAMs where both control signals appear. This constraint is moot for typical multipliers, because typical RAMs are much larger.

The cost to extend an input from always unsigned or always signed to support either is about the same as sizing the input one bit wider. This trend is somewhat visible in table 10.1, although the data are not smooth. In most cases, the number of parts needed to build a flexible-signage multiplier is only slightly greater than for its fixed-signage version, and the latency usually does not increase.

## 10.4 Partial product summation

A parallel multiplier can compute all subproducts for a multiplication during its first clock cycle by using a separate SRAM for each subproduct. The remainder of the circuit adds these subproducts in their correct place values to obtain the final product. The subproducts can be selectively exaggerated such that all are unsigned, yet their truncated sum yields the correct unsigned or signed final product. Thus the summation portion of the circuit only needs to use unsigned arithmetic. If neither factor is signed, the product is unsigned; otherwise, the product is signed.

Just like digits in grade-school long multiplication, the bits comprising partial products appear at specific place values where they are to be added. For example, a factor for a 24-bit unsigned multiplier can be partitioned into two 12-bit subfactors, and another factor into four 6-bit subfactors. Pairs taken from the two sets will each have 18 bits, so eight  $256\text{Ki} \times 18$  RAMs can look up their subproducts, which align in these 48 columns for summation:



*Dot diagrams* like the above were popularized by [Dadda65]. For clarity of derivation, the subproducts from the two 12-bit subfactors are shown in different colors. Dot diagrams are often rearranged and compressed to improve legibility of the remaining addends, and often look like a triangle:



Throughout this chapter, the least significant bit is drawn rightmost, and carries propagate from right to left. The leftmost bit in this example has place value  $2^{47}$ .

### 10.4.1 Carry-save addition

A dynasty of papers has been written about how to add subproducts. [Booth51] found a creative way to reduce the number of addends by half. Full adders were used in parallel by [Wallace64], who called the layers *pseudoadders*, because carries within a layer were passed directly to the next layer in their original place values instead of propagating them out first. The term *carry-save* refers to this deferral of carry propagation to a layer time. A more generalized term for full adder appeared in [Dadda65], which called it a  $(3, 2)$  counter, because “counting” three same-weight input bits yields two different-weight output bits.

The summation process was to keep reducing the number of input bits until at most two bits remain in each column; that is, two addends are left. At that point, a carry-propagating adder such as carry-lookahead or carry-skip would be used to produce the final product.

When full adders were used for the carry-save steps, each step would reduce the number of bits to sum by no more than one-third. A race ensued to find methods to increase the reduction per step in hope of reducing total summation delay. Some papers looked at soldered circuits, some at ASICs, others FPGAs. A few used ROMs as logic elements, some built from small circuits or basic gates, and some began with transistors. Permitted geometries changed along the way. For instance, [Verma07] considered  $(7, 3)$  counters, which required that all input bits for any counter come from the same column. But [Mora05, Mora08] used wider circuits that add the same number of bits from a few contiguous columns. This restriction was dropped in [Afshar08], which wrote of *generalized parallel counters* that permit differing numbers of bits per column.<sup>1</sup> A brilliant observation at IBM [Weinberger81] disavowed any need to count subproduct bits at all, because the real goal is to reduce their number without changing the total, not keep track of column-wise counts. This introduced an

---

<sup>1</sup>These circuits may be too generalized to call *counters* instead of *adders*. I have been using the phrase *arbitrary-geometry adder*.

efficient family of on-die logic elements called *compressors*, which have an explanation in [Oklobdzija96]. As for ROMs, [Stenzel77] soldered 66 together to build a multiplier, and [Paul09] used on-die sparse ROMs to build an ASIC multiplier.

All this literature has little bearing on SRAM carry-save addition, because an SRAM implements all functions within its capacity with equal efficiency. It's just a lookup table. All that is wanted is to take as many input bits as possible in order from least significant to most significant, and output as few bits as possible—which turn out to be the place-value-weighted sum of the input bits.

Figure 10.1 shows how subproducts from the 24-bit multiplier example would be summed. Although the caption says the factors are unsigned, the circuit would be identical for signed factors, albeit the firmware would differ. The coincidence does not apply for all word sizes, but happens by chance for 24 bits  $\times$  24 bits. Four layers of parallel SRAM follow the subproduct generation layer. This circuit, like all SRAM multipliers in this chapter, offers one multiplication per clock cycle, but there will be a certain number of cycles—a *latency*, which is equal to the count of RAM layers—before the product appears for a given pair of factors. So this figure shows *four* layers, but the total multiplier has a latency of *five* cycles when the subproduct generation layer is included.

The first SRAM layer in figure 10.1 is at the top, with up to five input bits per column. The horizontal column indicates place value, growing right-to-left from  $2^0$  to  $2^{47}$ . Each symbol within a column denotes an output bit from the prior layer, which is subproduct lookup in this case. Hexadecimal digits indicate which of this layer's eight RAMs each bit leads to.

The six least significant bits are alone in their columns, and therefore are their own sum. They are drawn as dashes to indicate unchanged passage to the next layer. They are called *final*, because they need no further computation. Another dash appears for a singleton bit at place value  $2^{28}$ , which did not fit in RAM 5. This bit would be its own column sum in RAM 6, so including it would waste half of that

```

888888887777776666655544443333222211111111-----
88-7777776666655554443333222211111111
77766665555444333322221
77766665555444433322221
6-5554444333

BBBBBBBBBBBBBAAAAAAAAA-9999999999-----
BB  B-A AAAAAAA -99 99 999

-----CCCCCCCCCCCCC-----
CC          CC

DDDDDDDDDDDD-----
D

```

**Figure 10.1:** Subproduct summation steps for a 24-bit unsigned multiplier using carry-save adders. The product is 48 bits. SRAMs are  $256\text{Ki} \times 18$ . Input bits to each of the 13 RAMs appear as hexadecimal digits, with each RAM accepting up to 18 bits. Dashes show bits that are deferred or already final. Color indicates **arbitrary-geometry addition**.

RAM's cells for no computation benefit. A dash appears at place value  $2^{39}$  for the same reason: its bit is too late for RAM 7 but too early for RAM 8. Depending on architecture decisions, bits marked with dashes will likely need D flip-flops to maintain pipeline consistency.

RAMs 1–7 absorb 18 subproduct bits each, leaving 10 bits for RAM 8. Each RAM's output bits are contiguous, one per column, and reflect the maximum possible sum of its addends. For example, RAM 5 has maximum sum  $1000010_2$  and therefore outputs seven bits across place values  $2^{25}$ – $2^{31}$ . The totality of these output bits defines the shape of the next layer, which is reduced further by RAMs 9–B. RAM C is alone in its layer, because the bits to its left and right already indicate their own sum. RAM D in the final layer finishes the subproduct addition and therefore the multiplication.

This all-carry-save method of figure 10.1 is of limited value, because the number of layers will grow approximately linearly with the word size. It's easy to see that ripple carry through RAMs 1, 9, C, and D is what limits the speed of this circuit, and

the problem would get worse as factors get longer. Multiplying 24-bit words already takes five cycles, and 64-bit words would take ten. The next section will show a method for 24-bit multiplication in four cycles, and 64-bit multiplication in five.

An unshown optimization exists for figure 10.1. This drawing shows 13 RAMs, but my tool [Abel22a] for synthesizing multipliers only uses 12, because RAMs **8** and **B** can be combined. RAM **B** only indicates 16 input bits, of which 8 come from RAM **8**, which has 10 input bits and sends its output to RAM **B** exclusively. This permits elimination of RAM **8** in favor of 10 D flip-flops, whereupon RAM **B** is fed with the  $16 - 8 + 10 = 18$  inputs that need computed. Firmware for RAM **B** is a functional composition of what the two firmwares would have done, sparing one RAM from the design. The complete multiplier then uses 20 RAMs: 8 for subproduct generation and 12 for carry-save addition.

### 10.4.2 Carry-skip addition

The terms *carry-save* and *carry-skip* are easily confused because of their shared initials and the coincidence that *save* and *skip* both support meanings *bail out* and *let alone* [Ward96]. The distinction is, a *carry-save* adder can “obviate the necessity of” carry propagation, but a *carry-skip* adder’s action is “passing over an interval from one thing to another” [Webster13]. Carry-skip’s passing over of components and component delay can appreciably shorten carry propagation paths and the latency of addition [Lehman61].

Carry-skip adders can be described using a mixed radix numeral system. In the ensuing discussion, all numbers are non-negative integers. The equations below apply to any number base and hold true for decimal calculators, etc., but the examples presented are in binary.

If some radix  $R$  has  $k$  ordered factors, these factors can represent  $k$  simultaneously-operating stages of a carry-skip adder modulo some base  $R$ .



$$R = \prod_{i=0}^{k-1} r_i \text{ where } \forall i, r_i > 1$$

This causes any summand  $X$  modulo  $R$  to have a unique representation using  $k$  terms in mixed radix:

$$\begin{aligned} X &:= x_{k-1} : \dots : x_1 : x_0 \\ &= x_0 + \sum_{i=1}^{k-1} x_i \prod_{j=1}^i r_{j-1} \text{ where } \forall i, 0 \leq x_i < r_i. \end{aligned}$$

Here is an example with  $k = 3$ , corresponding to a carry-skip adder that is implemented using three parallel stages. A summand  $X$  might be represented as  $01 : 11010 : 101$  in base 2, where colons indicate  $X$ 's partitioning according to the mixed radices that are factors of  $R$ . In particular:

$$R = 2^{10} \quad X = 01 \, 11010 \, 101_2$$

$$r_0 = 2^3 \quad x_0 = 101_2$$

$$r_1 = 2^5 \quad x_1 = 11010_2$$

$$r_2 = 2^2 \quad x_2 = 01_2$$

When adding a set of numbers  $S := \{S^0, S^1, \dots, S^{|S|-1}\}$  modulo  $R$ , the computation can stay in the mixed radix. If  $T := t_{k-1} : \dots : t_1 : t_0 = \sum S$ , then  $\forall i, 0 \leq i < k$ ,

$$t_i = (\Sigma_i + a_i) \bmod r_i, \quad \text{where} \quad (10.1)$$

$$\Sigma_i = \sum_{m=0}^{|S|-1} S_i^m, \quad (10.2)$$

$$a_0 = 0,$$

$$a_{i+1} = \lfloor (\Sigma_i + a_i) \div r_i \rfloor. \quad (10.3)$$

Here,  $\Sigma_i$  is the *tentative sum* before carry for term  $i$ , and  $a_i$  is its “actual,” recursive incoming carry. An example showing how sums and carries are determined from

arbitrary-chosen addend bits with  $|S| = 4$  and the same 10-bit mixed radix as before is:

10	11110	010	addend
00	01101	110	addend
11	10000	001	addend
01	10101	110	addend
10	01		carry
00	10001	111	sum

A machine can add quickly modulo  $R$  by computing the  $k$  terms of (10.1) in parallel. This requires parallel computation of (10.2), which uses one SRAM per term after the problem has been made small enough by carry-save adders. The difficulty is parallel computation of (10.3), which causes every  $t_i$  to recursively consume every  $\Sigma_i$  up to it as input, rapidly exhausting the few input bits an SRAM can accept.

There is a better way. Consider

$$a_i = c_i + c'_i,$$

where actual carry  $a_i$  is the sum of the “direct” carry  $c_i$  from the immediately preceding term only, and an occasional recursive carry  $c'_i$  from any earlier terms. Then rewrite:

$$t_i = (\Sigma_i + c_i + c'_i) \bmod r_i, \tag{10.4}$$

$$c_0 = 0,$$

$$c_{i+1} = \lfloor \Sigma_i \div r_i \rfloor, \tag{10.5}$$

$$c'_0 = 0,$$

$$c'_{i+1} = \left\lfloor \frac{((\Sigma_i + c_i) \bmod r_i) + c'_i}{r_i} \right\rfloor. \tag{10.6}$$

The idea of (10.6) is that  $c'_{i+1}$  is zero unless there is a recursive carry. For example when adding 1 to 999 in decimal,  $c_1 = 1$  and  $c'_1 = 0$  due to the direct carry into the tens place, but  $c_2 = 0$  and  $c'_2 = 1$  due to the recursive carry into the hundreds place.

The equation is such that  $c'_{i+1}$  steps from 0 to 1 if  $c'_i$  becomes large enough.

The work is simplified if  $c'_{i+1} \leq 1$  can be guaranteed. This will be recursively true if  $\forall i, c_i \leq r_i$ , which can be simply phrased as “don’t add too many numbers at a time.” In a binary adder, the constraint means that the number of carry bits into any term cannot exceed that term’s width. This spares the division in (10.6):

$$c'_{i+1} = \begin{cases} 1 & \text{when } c_i + c'_i \geq r_i - (\Sigma_i \bmod r_i), \\ 0 & \text{otherwise.} \end{cases} \quad (10.7)$$

The largest that  $c_i + c'_i$  can be for a given  $i$  is called the *maximum effective carry value* (MECV<sub>*i*</sub>) contributing to  $t_i$ . These constant MECVs are easily computed by maxing out all of an adder’s inputs. Here are the MECV determinations for the ten-bit mixed radix example with  $|S| = 4$ :

11	11111	111	addend
11	11111	111	addend
11	11111	111	addend
11	11111	111	addend
11	11		MECV
<hr/>			
11	11111	100	sum

A *propagate encoding*, written as  $p_i$  and often simply called *propagate*, is a range-bounded encoding of  $p'_i$ , a tentative sum’s closeness to wraparound modulo  $r_i$ .

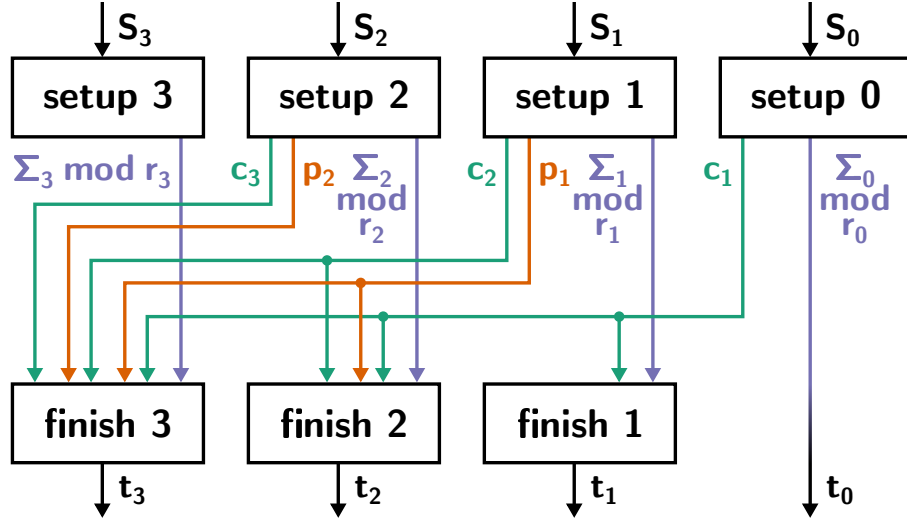
$$p'_i = r_i - (\Sigma_i \bmod r_i) \quad (10.8)$$

$$p_i = \begin{cases} p'_i & \text{when } p'_i \leq \text{MECV}_i, \\ 0 & \text{otherwise.} \end{cases} \quad (10.9)$$

Now (10.7) can be rewritten again.

$$c'_{i+1} = \begin{cases} 1 & \text{when } p_i \neq 0 \text{ and } c_i + c'_i \geq p_i, \\ 0 & \text{otherwise.} \end{cases} \quad (10.10)$$

A *two-layer carry-skip adder* adds quickly modulo  $R$  with a latency of two cycles.



**Figure 10.2:** Two-layer carry-skip adder, shown computing four terms in parallel. There is no presumption as to number of summands or radix, except that each carry  $c_i$  may not spill wider than its nearest destination term. The radix typically will vary between terms, as Figs. 10.3 and 10.4 show.

The first layer is called *carry-skip setup* or just *setup*, and uses just one RAM per term to find  $\Sigma_i \bmod r_i$ ,  $c_i$ ,  $p_i$ , and internal value  $p'_i$  from (10.2), (10.5), (10.9), and (10.8) respectively. The second layer is called *carry-skip finish* or just *finish*, and uses one RAM per term to find total  $t_i$  and internal value  $c'_i$  from (10.4) and (10.10).

Figure 10.2 draws a two-layer carry-skip adder having four terms. The drawing, like the preceding discussion, is radix-agnostic and does not presume that any logic elements or representations are binary. No presumption exists that the terms have the same radix, and there is no presumption as to the number of addends in  $S$ . The only practical constraint is that the problem must be small enough that each box in figure 10.2 can be implemented by one RAM. If not, the number of addends can be reduced by carry-save addition, and/or the problem can be truncated to a smaller radix  $R$ .



**Figure 10.3:** Subproduct summation for a 24-bit unsigned multiplier using carry-save alongside carry-skip adders. SRAMs are  $256\text{Ki} \times 18$ . Colors denote arbitrary-geometry addition, finishing arbitrary-geometry addition, carry-skip setup, finishing carry-skip setup, and carry-skip finish.

### 10.4.3 Fast subproduct totals

Figure 10.3 shows the 24-bit unsigned multiplier from figure 10.1, except that carry-skip summation is now used alongside carry-save, reducing latency from five cycles to four. One RAM has been added for 21 in total: 8 RAMs generate subproducts for 13 RAMs to sum. Figure 10.3 appears to show 15 summation RAMs, but underused RAMs **D** and **F** are consolidated into one package by later optimization, as are RAMs **9** and **B**.

Figure 10.3 and similar figure 10.4 retain green to indicate bits summed by arbitrary-geometry (carry-save) addition. Inputs to carry-skip setup RAMs are red. Bits destined for RAMs that do carry-skip finish are underlined. There are three combinations of these. Carry-skip finish without other processing is blue with underline. Finishing arbitrary-geometry addition, green with underline, indicates a carry-save adder that needs to include in its sum direct and recursive carries from the previous layer. Finishing carry-skip setup, red with underline, marks the horizontal and vertical intersection term of two carry-skip adders in consecutive layers. Using figure 10.2 as a model, “finishing setup” superposes in a single RAM the “finish 3” RAM (which is running low on input bits) of one adder with the “setup 0” RAM

(with more available) of a subsequent adder.

For legibility, Figs. 10.3 and 10.4 show all output bits from carry-save RAMs, all  $\Sigma_i \bmod r_i$  from setup RAMs, and all  $t_i$  from finish RAMs, but not any propagate or carry bits from setup RAMs. Not only would these be tricky to interpret, but copies of  $p_i$  and  $c_{i+1}$  are consumed by more than one RAM. Nonetheless, it's possible with some practice to surmise where these appear. In figure 10.3, setup **1** sends a carry but not a propagate to **9** and **A**. Setup **2** sends two carry bits to **9** and **A**, plus a propagate to cover **1**'s MECV of 1. Setup **3** sends three carry bits to **A**, plus two propagate bits to cover **2**'s MECV of 3. **9** serves double-duty finishing the terms from **2** and **3**, because **9** happens to be large enough.

The second summation layer finishes one carry-skip adder with **9** and **A**, and simultaneously sets up another with **A**, **B**, and **C**. **A** sends a carry bit but no propagate to **E** and **F**. **B** sends a propagate to **E** and **F**, but doesn't carry, because it only has one summand bit. Here is an opportunity for further optimization, first because **B** is an identity function with equal sum and propagate, and second because its same-valued bit occupies two inputs of **E**, with one read as a sum and another read as a propagate. The tool I wrote does not do that specific optimization. **C** sends a carry to **F**, along with a propagate to cover **B**'s MECV of 1.

The essential heuristic for efficient summation in medium-sized<sup>2</sup> multipliers is, favor aggressive use of carry-skip addition. By taking as many columns as possible for each carry-skip setup and not backtracking to evaluate other possibilities, consistently efficient multipliers are produced. But not optimally efficient, because carry-skip adders have “combinatorial hiccups” that I suspect are due to SRAM granularity.

My multiplier synthesis tool [Abel22a] supports a few methods for generating candidate multipliers, from which a “lowest cost” circuit can be chosen. These strategies look at just one RAM at a time, just one layer at a time, or the whole multiplier at once, and backtrack accordingly. The relative cost of two designs is the number of

---

<sup>2</sup>All carry-save summation is sometimes more efficient for narrow factors, and multilayer carry-skip summation for wide factors.

layers, with ties broken by number of SRAMs, with those ties broken by total firmware size in bits. The 24-bit unsigned multiplier of figure 10.3 is the winning design among 133 candidates, sparing 12 288 firmware bits compared to the second-place circuit, which doesn't happen to have the anomalous one-input-bit **B** RAM.

The 64-bit, either-signage multiplier in figure 10.4 was chosen from 350 651 designs. Its seven-term carry-skip step contributes largely to its ability to multiply in six cycles, instead of ten cycles if using only carry-save adders. The first carry-skip stage has only one setup RAM, so two carry bits but no propagate bits are generated. Conceptually this seems identical to arbitrary-geometry addition, but the implementation outcome differs. Greedy arbitrary-geometry addition would have taken 18 input bits and led to a multiplier with 152 RAMs, not 151 as in the figure. Carry-skip setup had to stop early at 17 bits, because term boundaries must be located at place value boundaries. In this instance, a locally worse strategy happens to produce a globally cheaper multiplier. Unfortunately, an exhaustive search of all opportunities to strategically stop some number of bits short when filling RAMs has dreadful time complexity.

Table 10.2 shows the effect of SRAM size and allowable latency on multiplier width. The table cells are factor widths, in bits, that my tool is able to design a multiplier for within the allotted number of layers, given a few assumptions:

- The multiplier takes two factors of the same width.
- Factors are either-signage; that is, factors may be specified as any combination of signed and unsigned at the time they are presented.
- Signage control does not count toward factor width.
- Only the methods covered so far are used, namely arbitrary-geometry and two-layer carry-skip addition.

The maximums in table 10.2 would in most cases be one greater for fixed-signage





**Table 10.2:** Maximum bits per factor for two-input, either-signage, readily-designable multipliers for various SRAM sizes and latencies. Only two-layer carry-skip and arbitrary-geometry adders are used; cascaded carry-skip schemes are considered in table 10.4. Widths in **bold color** use a greedy algorithm to reduce design time at the expense of factor width.

SRAM size	latency in clock cycles									
$2^n \times n$	1	2	3	4	5	6	7	8	9	10
4	1	1	2	3	4	5	6	8	9	11
5	1	2	3	5	5	8	10	12	14	17
6	2	2	3	5	8	11	14	20	23	28
7	2	2	4	6	10	14	19	26	33	38
8	3	3	4	8	13	19	26	35	43	45
9	3	3	6	10	15	23	34	43	43	46
10	4	4	7	14	20	29	44	58	53	65
11	4	4	7	14	24	35	51	56	64	72
12	5	5	9	17	29	43	63	62	78	103
13	5	5	10	17	34	50	75	72	84	110
14	6	6	10	20	39	62	88	84	127	126
15	6	6	12	23	44	68	88	100	140	150
16	7	7	13	26	49	78	80	105	146	153
17	7	7	15	27	55	88	102	110	136	159
18	8	8	15	33	63	99	84	131	145	200
19	8	8	16	34	69	109	114	171	191	220
20	9	9	19	39	77	90	104	212	221	246
21	9	9	19	41	86	105	143	198	222	308
22	10	10	21	47	92	135	192	209	252	300
23	10	10	21	49	103	164	197	249	296	390
24	11	11	22	54	111	175	185	276	360	375
25	11	11	24	54	120	192	249	301	351	384
26	12	12	25	64	130	198	230	301	347	401
27	12	12	26	64	140	185	247	344	448	539
28	13	13	31	72	153	243	294	371	416	505
29	13	13	32	74	163	201	257	304	395	460
30	14	14	32	78	149	284	303	443	488	587
31	14	14	33	83	184	267	336	369	479	627
32	15	15	37	90	183	307	343	505	530	695
33	15	15	38	95	209	308	362	461	562	661
34	16	16	38	101	205	374	378	564	623	649
35	16	16	41	105	200	338	409	578	585	776
36	17	17	41	111	232	406	608	623	766	867

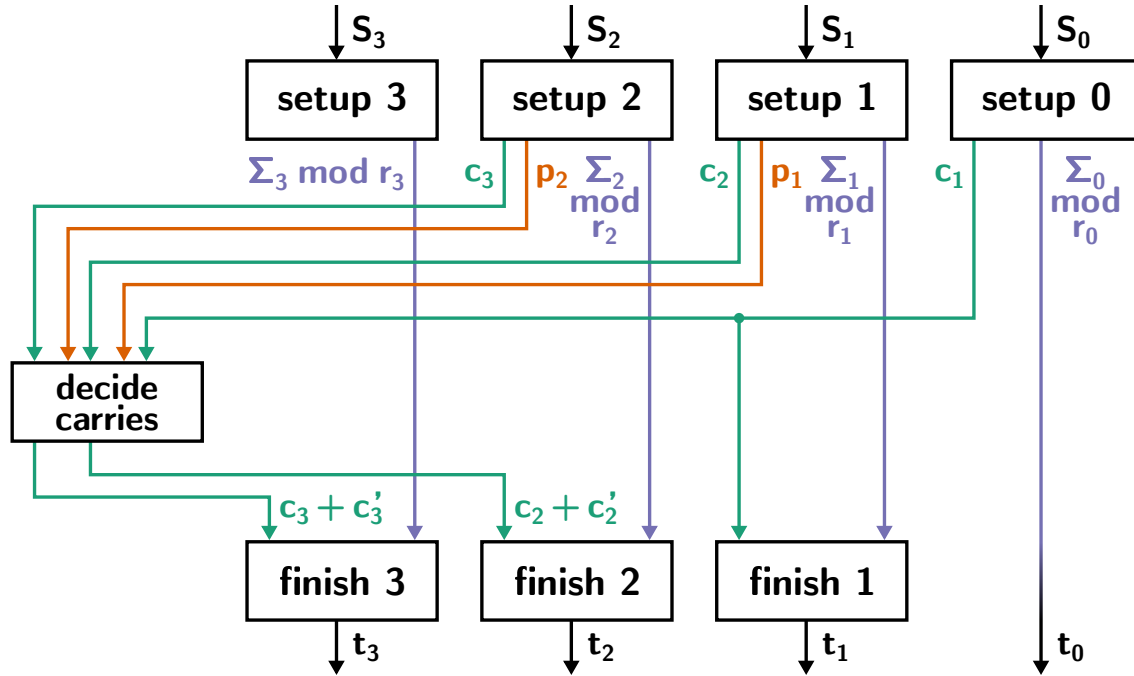
multipliers. For example,  $256\text{Ki} \times 18$  RAMs can build a 64-bit unsigned  $\times$  64-bit signed multiplier in five layers instead of six.

Although present commercial synchronous SRAMs have between  $2^{18}$  and  $2^{24}$  words of either 18 or 36 bits, the rows of table 10.2 show a wider range to show trends. One trend is that doubling SRAM size has only a small effect on the number of bits that can be multiplied for a fixed latency. Consequently, it usually takes a large increase in RAM size to build the same multiplier in one fewer layer.

Table cells with medium black print indicate known maximum widths for their latency, so if a cell says 24, it means 25 was also attempted but did not succeed. Forty thread hours on a circa 2012 computer were used to compute each row of black print. These were full backtracking runs that considered the entire multiplier cost. This left 128 empty cells that ran out of CPU time. These appear in **bold color**, and were completed by greedily adding one RAM at a time to the circuit without backtracking. There is a discontinuity between the two regions of the table, with the colored portion showing reduced factor widths possible because of the reduced design effort. This duality of visualization loses some information: for example,  $30 \times 30$  SRAMs reached a 170-bit multiplier with 5-cycle latency before the 40-thread-hour limit. This exceeds the table's figure of **149** bits, but 170 bits is not thought to be the maximum reachable for the method indicated, and therefore is not reported in this table of maximums.

#### 10.4.4 Multilayer carry-skip addition

For typical CPU word sizes, the previous section concluded this chapter. In seven layers, 557 commodity  $256\text{Ki} \times 18$  RAMs, can multiply  $128 \times 128$  bits with either signage. But what if someone asks about a fast multiplier for  $8192 \times 8192$  bits? The method thus far breaks down. Even doubling the latency to 14 layers would only reach  $300 \times 300$  bits, because finish RAM inputs in two-layer carry-skip adders clog up with propagate and carry data, limiting how wide each layer can stretch. Yet it



**Figure 10.5:** Three-layer carry-skip adder. The dedicated carry decision RAM frees finish RAM inputs to support wider summands, but adds latency. Expansion to four or more layers would invoke a tree of carry decision RAMs, and accommodate very wide summands.

is possible to multiply 8 192 bits in 14 layers by using a more potent carry scheme.

Figure 10.5 shows a *three-layer carry-skip adder*, which starts with figure 10.2's two-layer adder, but separates carry propagation into its own layer instead of propagating carries and adjusting tentative sums within the same RAMs. Although this decluttering adds a cycle of latency, input bits are freed in the leftmost finish RAMs, allowing wider terms in the adder, and the carry decision RAM has all of its input bits dedicated to carry and propagate information, allowing more terms. Three-layer adders are especially useful within the non-multiplying portion of ALUs, because an SRAM logarithmic shifter and a lot more can be superposed over the second and third layer in the manner of chapter 5, resulting in a robust set of ALU operations. But for multipliers built from  $256\text{Ki} \times 18$  SRAMs, the latency added by the new middle layer will not be recovered by the longer carry propagation chains it facilitates.

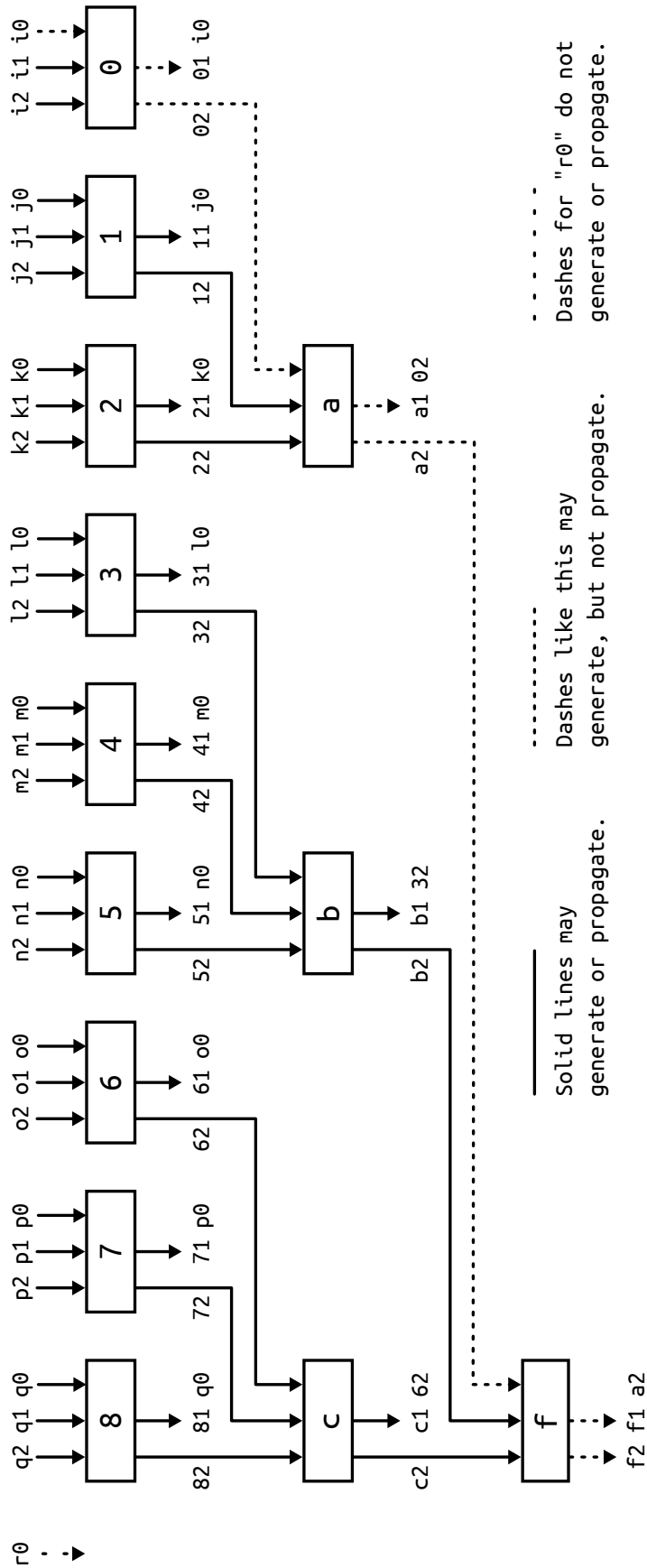
**Table 10.3:** .

Computation of the 27 carry decisions from figure 10.6’s hierarchical carry-skip adder example. **Bold color** indicates which finish RAM each carry decision is for. Up to three partial carry decisions from figure 10.6, shown here from most to least significant, are combined for each decision.

<b>i1</b>	i0	<b>ℓ1</b>	ℓ0	a2	<b>o1</b>	o0	f1
<b>i2</b>	01	<b>ℓ2</b>	31	a2	<b>o2</b>	61	f1
<b>j0</b>	02	<b>m0</b>	32	a2	<b>p0</b>	62	f1
<b>j1</b>	j0	<b>m1</b>	m0	32	<b>p1</b>	p0	62 f1
<b>j2</b>	11	<b>m2</b>	41	32	<b>p2</b>	71	62 f1
<b>k0</b>	a1	<b>n0</b>	b1	a2	<b>q0</b>	c1	f1
<b>k1</b>	k0	<b>n1</b>	n0	b1	<b>q1</b>	q0	c1 f1
<b>k2</b>	21	<b>n2</b>	51	b1	<b>q2</b>	81	c1 f1
<b>ℓ0</b>	a2	<b>o0</b>	f1		<b>r0</b>	f2	

What would help is allowing the carry decision RAM of figure 10.5 to have as many inputs as needed, rather than stopping around 18 bits. The approach is isomorphic with canonical parallel methods of computing exclusive prefix sums, where scalar addition has been replaced by carry aggregation. Leaving that metaphor aside, the intuition is that a long line of setup RAMs can be partitioned into small segments, with each segment having its own RAM to process carries. These will then aggregate into progressively smaller layers of carry processing. This structure is shown in figure 10.6, where 28 terms of carry-skip setup, labeled i0 through r0, are combined in a hierarchical scheme that soon computes the correct carry decision for each finish RAM. This is the foundation for a *multi-layer carry-skip adder*.

Section 10.4.2 generalized carry-skip addition for any number of summands, introducing the useful possibility that carries and propagates not be limited to one bit each. Multilayer carry-skip adders are enough complex and robust, that it helps in these to re-impose the conventional expectation that each term uses at most one bit each for propagate and carry. In particular, for any term  $i$ , exactly one of



**Figure 10.6:** Hierarchical scheme for a five-layer, 28-term carry-skip adder. The 28 nodes across the top line are propagate and carry information from 28 setup RAMs. Carry decisions for the 27 finish RAMs combine up to three nodes each from this tree. Table 10.3 shows which nodes contribute to each carry decision. The two-character node labels have no purpose other than to identify the nodes unambiguously.

condition	notation	description
$c_{i+1} = 0$ and $p_i = 0$	-	no carry
$c_{i+1} = 1$ and $p_i = 0$	G	generate
$c_{i+1} = 0$ and $p_i = 1$	P	propagate

applies. Limiting each place value to a maximum of two summand bits is simple and sufficient (but not necessary) to ensure one of these three conditions always holds. When this is the case, the setup RAMs will output some sequence of carry and propagate information, such as for 28 terms:

$$\text{P-PGP---PGGP--P-PPGPP-GPPPGG}$$

These terms are written from most significant (left) to least (right). *Carry aggregation* can be viewed as a function, notated solely by adjacency without an operator symbol, that is associative but not commutative. The above could be rewritten as

$$\text{P } ((-PG)(P--)(-PG)) ((GP-)(-P-)(PPG)) ((PP-)(GPP)(PGG))$$

using the associative property. This can be reduced by the *carry aggregation function* using its ternary truth table:

$t_{i+1}$	$t_i$	$t_{i+1}$	$t_i$	$t_{i+1}$	$t_i$	$t_{i+1}$	$t_i$	$t_{i+1}$	$t_i$	$t_{i+1}$	$t_i$
-	-	-	-	G	-	G	-	P	-	-	-
-	G	-	-	G	G	G	-	P	G	G	-
-	P	-	-	G	P	G	-	P	P	P	-

The 28-term example then simplifies to:

$$\begin{aligned} & \text{P } ((-)(-)(-)) ((G)(-)(G)) ((-)(G)(G)) \\ &= \text{P } (-)(G)(-) \\ &= - \end{aligned}$$

**Table 10.4:** Characteristics of hierarchical carry-skip unsigned multipliers employing  $256\text{Ki} \times 18$  SRAMS.

bits per factor	number of layers	number of SRAMs	firmware size (bits)
32	6	40	86 839 360
64	6	143	287 047 690
128	7	526	1 415 782 400
256	8	2 042	5 378 101 864
512	9	7 899	21 684 594 176
1 024	10	31 295	85 062 327 270
2 048	12	124 653	338 455 229 862
4 096	13	497 306	1 347 697 573 632
8 192	14	1 981 799	5 404 324 177 388

Figure 10.6 depicts a 28-term SRAM carry aggregation tree with a fanout of three, paralleling with the above example. A typical fanout would be nine, because a  $256\text{Ki} \times 18$  RAM can process nine carry/propagate pairs, but three is easier to draw and read. The nodes of figure 10.6 indicate partial carry decisions for the figure's subtrees. Here, *partial* means an answer may not yet be  $-$  or  $G$ , but may be  $P$ , meaning undecided. The eventual carry decision for each term will be a composition of the carry information for all less significant terms, like an exclusive prefix sum, as found by further merging one to three subtrees. Table 10.3 shows which trees to combine for each finish RAM's carry decision, which will always be either  $-$  or  $G$ .

My tool [Abel22a] can synthesize multipliers with multi-layer carry-skip adders, and is publicly available for any purpose. Table 10.4 shows the latency and cost to build such unsigned multipliers for power-of-two widths. Here again, the devices in the table can all do one multiplication per clock cycle, but the output will lag the input by the indicated number of layers. In real circuits, the large fanout from subfactor bits to many RAMs will either need amplification, thereby adding some delay, or a reduction in clock speed.

## 10.5 Implication and contribution

By using large lookup tables that are compatible in size with presently available SRAM ICs, long-established methods for binary integer multiplication can be augmented in a manner that permits easy construction of multipliers using maker-scale tools. Specifically, hardware multipliers can be included in solder-defined minicomputers comparable to the architecture of this dissertation in size, cost, power consumption, and speed without resorting to any purchased complex logic such as microprocessors, ASICs, PLDs, or FPGAs. Although these multipliers will be orders of magnitude larger, costlier, slower, and more cumbersome (because the firmware lives in volatile memory) than present on-die multipliers, a potential workaround for limited use now exists in the event dependence on the supply chain for microprocessors, ASICs, PLDs, and FPGAs needs to lessen on account of trust and reliability.

This work is the first to present a radix-agnostic treatment of carry-skip addition, extend carry-skip addition to consider multiple addends and multiple carry bits, propose an offset binary technique to represent subproducts, or offer a design methodology for adapting parallel multiplication of any geometry to lookup tables for any plausible RAM size. The open-source tool [Abel22a] is the first implementation of these methods in code. Tables 10.1, 10.2, and 10.4 offer the first hard figures published with respect to speed, number of SRAMs, and firmware size of SRAM multipliers.





# 11

## Minicomputer implementation

The central deliverable I wrote into my topic proposal was a fully-assembled minicomputer that could run programs. The end of November 2022 marks 18 months past the date I projected to receive an assembled board, and as yet not even its *design* is finished. I don't count this as a loss, and I am glad that I challenged myself with an ambitious project. It's possible that I accomplished more than I would have with fewer expectations.

This chapter is about what I have done to bring solder-defined minicomputers from being a suggestion to being deployed around the world. Already, a 36-bit solder-defined CPU has been designed and validated in simulation that almost fully demonstrates the “chapter 8 minicomputer.” Today's implementation is about 20 lines<sup>1</sup> of firmware code from implementing all the operations in table 11.1, and about 100 lines of firmware code from implementing everything through chapter 8.

The implementation is exclusively in simulation at this time, and is completely built from plain ASCII text files in various languages: C, Python 3, a small macro language for specifying netlists, a language for specifying component locations and rotations, and the architecture's assembly language. There are zero external dependencies beyond the libraries that come by default with C and Python.

At 19 402 lines as of November 2022, the implementation is small relative to what

---

<sup>1</sup>The 20 lines are to define instruction pointer control signals for `CALL` and `RETURN`.

**Table 11.1:** Number of instructions required for some common tasks. All instructions take four clock cycles.

number of instructions	arithmetic	logic	complex
0	accumulate range-exceeded flag		
1	add	bitwise Boolean (16 operations)	load or store data RAM
	subtract	shift by a preset # of positions	call, return, (un)conditional jump
	compare	rotate by a preset # of positions	36-bit linear feedback shift register
	minimum	reverse order of bits in word	accumulate hash function on 36 bits
	maximum	parity	36-bit cipher round function
2	absolute value (up to 31 bits)	shift or rotate bit through T flag	incr. or decrement mirrored word
		shift by a variable # of positions	36-bit pseudorandom number
	absolute value (full 36 bits)	rotate by a variable # of positions	popcount (Hamming weight)
3		leading or trailing bit manipulation	
	unsigned mult. by constant 0–63		
4	unsigned mult. by variable 0–63	count leading or trailing 0s or 1s	possibly any 36-bit permutation
5			any 36-bit permutation
47	36-bit unsigned mult. (72-bit result)		

**Table 11.2:** Tally of the minicomputer firmware size, measured in rows.

type	number of RAMs	number of operations	input bits per operation	number of rows
ALU $\alpha$	6	29	12	712 704
ALU $\beta$	6	15	12	386 640
ALU $\gamma$	6	20	12	491 520
ALU $\theta$	1	12	13	98 304
ALU $\zeta$	1	16	10	16 384
decoder	2	163	2	1 304
total	22			1 688 856

it includes. This count, which includes comments and blank lines, includes everything needed to go from a fresh host image with C and Python 3 installed, to running electrical simulations of a chapter 8 minicomputer running assembler programs.

Although this chapter describes the implementation, specifics as to how the source tree is structured and what files are named are sufficiently likely to change as to not belong here. Appendix D contains a guide of what can be found where.

## 11.1 Firmware implementation

Table 11.2 lists the firmware-containing RAMs, the number of operations each implements, and how many input bits each operation accepts. These figures lead to a total number of rows, or SRAM addresses where a value must exist, for each case. Almost the entire firmware of the minicomputer is for the arithmetic logic unit. The remaining firmware specifies control signals for the four clock cycles of 163 opcodes.

The contents of the  $\alpha_i$  RAMs are interchangeable for some operations but not all, because tribble position is a parameter for certain operations such as  $\alpha.e35$  (p. 144). Tribble position also causes differences among the the  $\beta_i$  and  $\gamma_i$  RAMs.

The motivation for Table 11.2 is to establish two needs. Although the firmware is simply a collection of lookup tables, the total number of table entries is 1 688 856, so

manually writing the firmware is not practical. Second, the size and importance of the ALU firmware in particular make regression testing a technical and moral imperative.

The principal point to understand about the firmware is that it's algorithmically generated, so it exists in three forms today:

1. *Firmware source code* is a program that generates lookup tables that control the ALU and control decoder.
2. *Executable code* is produced by a C compiler from the firmware source code.
3. *Firmware* is produced by running the executable code.

A *firmware image* residing in nonvolatile storage will be a fourth form in the future. This image will contain both the firmware and additional code that the firmware loader needs to load the firmware into the ALU and control decoder. Section 9.2 describes this process.

A small virtual machine has been written that can execute all ALU opcodes, all conditional and unconditional JUMPs, CALL, RETURN, HALT, NOP, and the load immediate opcodes IMB, IMH, IMN, and IMP. This virtual machine is accessible via a tool `vm` that can

- test single ALU instructions from the command line,
- load and run assembler programs that use the supported opcodes,
- run regression tests of most ALU opcodes,
- measure characteristics of the MIX and XIM opcodes, and
- generate pseudorandom byte streams for Dieharder [Brown20] testing.

Listing 11.1 shows `vm`'s help text for its command line options. Listing 11.2 shows a manual test of the LSL (logical shift left) instruction using `vm`.

The regression tests included with `vm` verify the numeric and CPU flag output of 110 ALU opcodes. Correct operation of overrange checking is a central component of

```
$ vm -h
Usage: vm [OPTION]... [EXPRESSION]
```

If EXPRESSION is present, like "34 a.sss -12345", calculate it.

Other notable uses and options:

```
-a FILE    run assembler on FILE and execute
-c         clear screen at startup
-d         show more diagnostics
-f         flip left and right operands to EXPRESSION
-h         display this help and exit
-r         run regression tests
-s OPT     call experimental stats module with option OPT
-w         swizzle right arg of EXPRESSION across tribbles
-N         set N(egate) flag to evaluate EXPRESSION
-R         set R(ange) flag to evaluate EXPRESSION
-T         set T(emporal) flag to evaluate EXPRESSION
-Z         set Z(ero) flag to evaluate EXPRESSION
```

**Listing 11.1:** Command line options for vm tool.

```
$ vm 000077007777 lsl 050505050505
      L 000000 000000 111111 000000 111111 111111      t -> t
      R 000101 000101 000101 000101 000101 000101
= alpha 000000 000000 111111 000000 111111 111111      p 001011
L  beta 010110 010110 010110 010110 010110 001011      c 110100
l  gamma 000000 011111 100000 011111 111111 100000      d 000000 r
      flags i n r t z
unsigned 528613344
signed 528613344
```

**Listing 11.2:** Manual ALU test of a five-bit logical shift left. The bits to be shifted appear at input L. Input R has the number of positions to shift available at all subwords. The shifted word appears at output  $\gamma$ .

these tests. Assembler programs for short ( $36 \times 6$ -bit) and long ( $36 \times 36$ -bit) multiplication, R(ange) flag save and restore, the two-instruction macro for absolute value, and the two-instruction macro for popcount (Hamming weight) are also tested.<sup>2</sup>

Operands for regression testing are derived from `/dev/urandom`. Different types of intentionally skewed distributions are used for a portion of the trials in order to improve test coverage for some of the opcodes. For example, additive opcodes receive extra trials with operands close to 0 and  $-(2^{35})$ . Short multiplication and **NUDGE** receive extra trials with the leading 1 bit's position uniformly distributed.

Wrapping addition and subtraction are missing from regression testing. They were added later and somewhat in a hurry, and are simpler than their range-checked counterparts. They should be included. **TXOR** (transposing XOR) is likewise missing from testing and should be added.

Simple unary instructions are also missing from regression testing. I did not view these as a priority, because they operate on a tribble-wise basis in only one ALU layer per instruction. This makes independent tests more difficult to write. Where possible, the ALU regression tests are independent of the firmware implementation. Because the  $\alpha$ ,  $\beta$ , and  $\gamma$  SRAMs only accept six-bit slices of their left and right operands, their desired outputs are computed in C using 36-bit (in reality 64-bit) arithmetic and compared on a whole-ALU basis.<sup>3</sup> Because of the missing regression tests for simple unary, I have manually checked the simple unary functions carefully and quarantined them in a change-restricted source file.

Stacked unary instructions are missing from regression testing. Adding this testing is perhaps the highest-priority ALU firmware work that remains. These tests depend on first defining and stabilizing implementations for the stacked unary macros in table 7.18. One reason I have neglected this testing is that the outcome of these

---

<sup>2</sup>The test output for long multiplication indicates that the R(ange) flag is set. This multiplication can never overflow, and should therefore never set this flag, so attention is needed here. The 72-bit products are computed correctly.

<sup>3</sup>Because **MIX** and **XIM** necessarily do subword mixing, they can't be tested by independent 36-bit code. Instead, they are just checked to ensure they are each other's inverse.

tests will affect the firmware only and not the netlist or circuit board.

## 11.2 Assembler

As assembler with the features needed to test the firmware and circuits has been written in C. More capabilities will be added as they become necessary.

### 11.2.1 What the assembler includes

Presently, the assembler supports all ALU opcodes, all conditional and unconditional JUMPs, CALL, RETURN, HALT, and NOP. The load immediate opcodes IMB, IMH, IMN, and IMP are available using a pseudo-instruction IMM that selects the opcode that can represent the immediate value given. These instructions only have room for 18-bit constants, so arbitrary 36-bit immediate values do not fit. Here is how to load an 36-bit immediate value, and what the assembler will eventually do behind the scenes:

```
unsigned fact53 temp

; imm fact53 53316291173 ; will not assemble

imm fact53 53316157440 ; left half of word
imm temp 133733 ; right half of word
fact53 = fact53 or temp ; intended result
```

Appendix A introduces basic features of the assembly language that will not be restated in this chapter. The appendix is from a manual-in-progress for an assembler-in-progress, so a few of its provisions are yet to be implemented. The principal differences between what the implementation does as of November 2022 and what Appendix A specifies are these:

- Decimal numerals work. Octal numerals follow C's easy-to-implement but otherwise dangerous tradition of a leading zero, so 077 means 63. No other radices



work yet, nor do the backtick radix notations. Once the backtick radices are working, leading zeros will be made safe, at which time 077 will mean 77.

- Underscore group separators work for numerals, so you can write 123456 as 12\_3456\_. But underscore may not be used as a prefix, so you can't write 789 as \_789 until this is corrected.
- The assembler permanently allocates registers for all numeric constants it finds, so the code `a = a + 1` will create a read-only register named "1" in order that the statement can increment as intended. The virtual machine `vm` presets these registers prior to simulating any code, and everything works. But a real minicomputer doesn't initialize registers by extrasensory perception, and there is no executable file format defined that would let a program loader initialize registers either. Therefore, the electrical simulation `ns` will run the code just fine—except the register named "1" will not be initialized, causing the increment to have undefined results. The present workaround is to define and initialize registers for all immediates in the source code like this:

```
unsigned one
imm one 1
; ...
a = a + one
```

Listing 11.3 shows diagnostic output from `vm` when the Fibonacci number program (listing 8.1, p. 176) is run. The CPU has no I/O, so the program has none either, but the assembled code, symbol table, and a list of changed register values at the time the program terminates can be seen. The virtual machine zeros all registers when a program is started, so the changed registers are the ones that have nonzero final contents. The symbol table could bear a few explanations:

- The right column is the identifier in the source code. This is the symbol table's key, and you'll see the lines are sorted on this right column. Although it is more

```

$ vm -a fibclean.a

--- code ---
  nop 0 0 0
  nop 0 0 0
  nop 0 0 0
  imp 0 53 6
  j 0 0 6
  halt 0 0 0
  imp 0 1 2
cmp.uu 6 2 0
  j.le 0 0 20
  imp 0 1 4
  imp 0 0 5
  imp 0 1 1
a.uuu 5 1 3
  j.t 0 0 22
a.uuu 2 4 2
a.uuu 0 1 5
a.uuu 0 3 1
cmp.uu 6 2 0
  j.ne 0 0 12
  j 0 0 5
a.uuu 0 6 1
  j 0 0 5
  imb 511 511 1
  j 0 0 5
  halt 0 0 0

--- symbols ---
  0 u. 0
  1 u. answer
  5 l. back
  6 l. fib
  2 u. i
 12 l. loop
  3 u. next
  4 u. one
  5 u. prev
 20 l. small
 22 l. toobig
  6 u. x

--- running ---

--- changed registers ---
53316291173 answer
          53 i
53316291173 next
          1 one
32951280099 prev
          53 x

```

**Listing 11.3:** Virtual machine output running the Fibonacci program of listing 8.1 (p. 176). The right column is a continuation of the left column.

traditional to put the key on the left, identifiers have no length limit, so the list will display more cleanly with identifiers at the end of the lines.

- The middle column indicates whether the identifier refers to a register number (not register contents) or a label. Unsigned registers display as `u`, and signed registers if present would display `s`. Labels display `l` and are destinations of `JUMP` and `CALL` instructions. They resolve to an address in the code memory.
- The left column is the register number or code memory address that the identifier resolves to. Right now there is only one namespace for identifiers, so a label and register may not have the same name.
- Register numbers are assigned in the lexical order of their names, not their order of appearance in a program. Labels are assigned in their order of appearance, which for listing 8.1 happens to match their lexical order.
- There is a strange register named `0`, although the immediate value zero does not appear in the source code. The architecture does not have any assignment opcodes, so lines such as `prev = answer` are implemented by adding zero with the `a.uuu` (add unsigned to unsigned with unsigned result) opcode, which requires the assembler to supply a zero immediate value.<sup>4</sup>

### 11.2.2 Future assembler features

As more of the architecture becomes available, the assembler will need more capabilities in order to test or use the expanded implementation. Here are a few things that would be added:

- Labels will have their own namespace. They will also appear separately in

---

<sup>4</sup>This immediate value is not initialized in the electrical simulation `ns`, so the program's appearance of correctness in netlist simulations is accidental. Extrapolating on this, the `NR` (not right) instruction of table 7.3 (p. 107) is probably broken in the electrical simulation due to its all-ones left operand that the assembler does not provide initialization code for.

diagnostic output, rather than hide alongside other symbols.

- The memory instructions LD, ST0, RCM1, RCM2, WCM, RDM, WDM, RPT, and WPT will be supported.
- The remaining non-ALU instructions, such as CALI (call and initialize) and those that chapter 9's subsystems will bring, will be supported.
- Radices for numeric constants will be available per section A.3.
- There will be spill mechanisms for programs that use more than 512 registers.
- The assembler will compute permutation operands per section A.8.
- Programs will be permitted to contain multiple modules.
- Decisions about scopes and namespaces will be made.
- Program loader conventions and an executable file format that is not overly complex will be decided on.
- Methods for defining blocks of memory will be established.
- The language will afford practical handling of records with named fields.
- The assembler macros in tables 7.18 (p. 163) and 7.19 (p. 168) will be implemented using a non-hardcoded (not C code) technique that permits ordinary assembler programs to define their own macros that have similar capabilities.
- The virtual machine will be extended to simulate non-ALU instructions. The reason this is needed is that the electrical simulation `ns` runs slowly, and cross-development of nontrivial programs will need fast testing.

## 11.3 Netlist definition and processing

### 11.3.1 Off-the-shelf electronic design automation software

Because the minicomputer’s hardware and firmware is to be open source, all intellectual property necessary to realize the minicomputer at any stage, such as software for laying out the circuit board, must also be available to anyone without restriction. In 2020, I looked at then-available electronic design software in hopes of finding something that could capture defining information about the circuit and lay out circuit boards. I don’t regret excluding commercial and closed-source products from consideration, but something I should have done is approach electrical engineering faculty at Wright State and ask if they could suggest any software packages. Instead, I searched on my own for alternatives.

For reasons forgotten, I settled on KiCad [KiCad22] and started learning to use version 5.0.2. Also for reasons forgotten, I was not optimistic about any of KiCad’s peers at that time. KiCad has helped me very little thus far, due to two limitations.

The first is that a minicomputer’s netlist is topologically complex. KiCad comes with a schematic editor, but schematics are planar drawings. Labels are supported for places where drawing a wire would be too cumbersome, but I learned quickly that my minicomputer schematic would be nearly *all* labels. This challenged my hope that a graphical editor could be a practical design tool. The problem can be seen in figures 11.1 and 11.2, which are early efforts to draw the ALU’s  $\alpha$ ,  $\beta$ , and  $\gamma$  RAMs. The drawings show 44-pin  $64\text{Ki} \times 16$  asynchronous SRAMs, which I had in mind at the time. Two drawings exist in order to show different pins on the 18 SRAMs.

Having that much of a schematic with 18 ICs, 36 bypass capacitors, and 864 pins to solder, I decided I was ready to learn a little of KiCad’s circuit board layout tool. When the tool starts for the first time, the components appear as a big cluster on a board, and my task was to drag them to where they go. After that, I would start

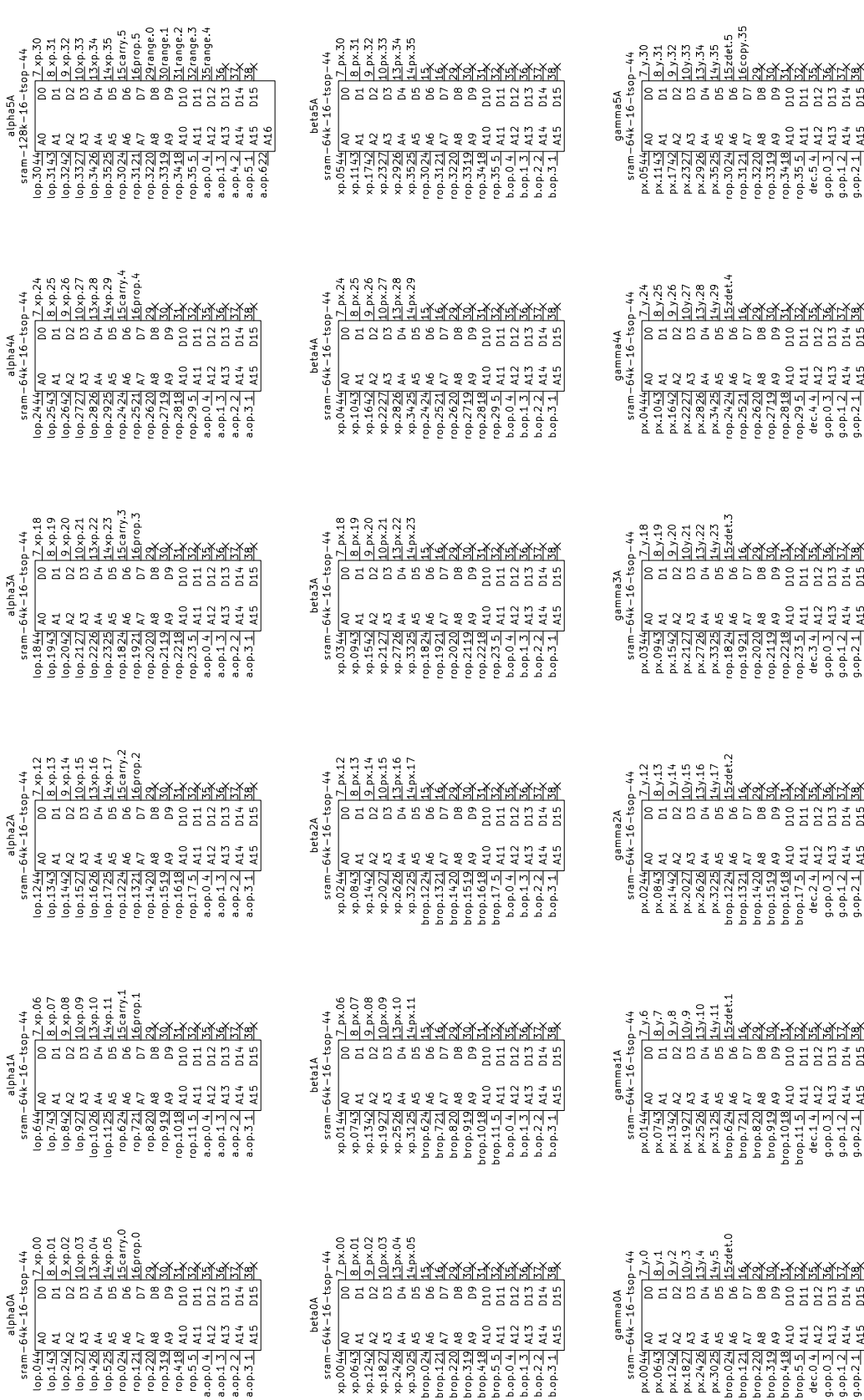
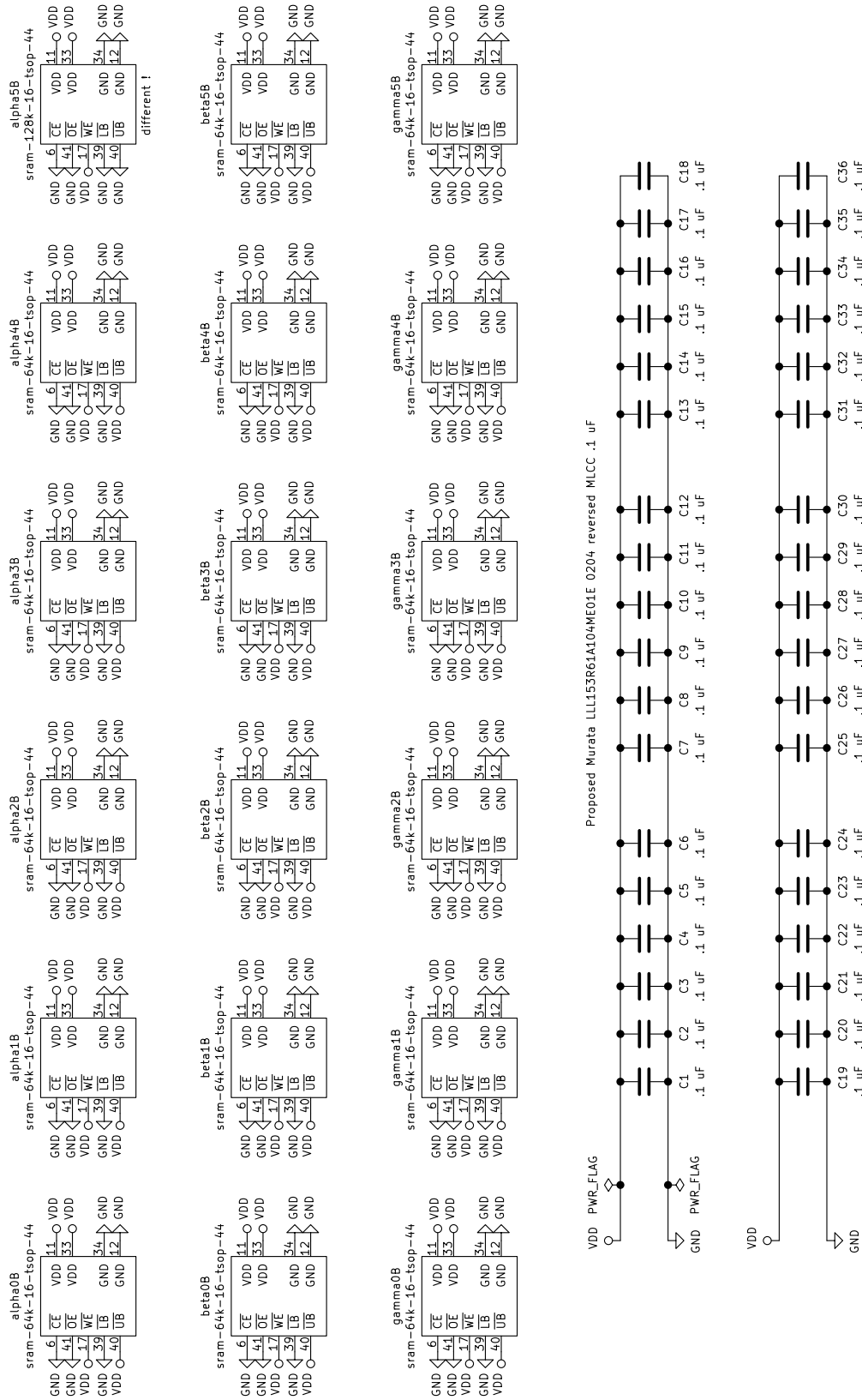


Figure 11.1: Partial KiCad drawing of the ALU datapath. This was as awkward to draw as it is useless for understanding.



**Figure 11.2:** Partial KiCad drawing of the ALU power connections. A mistake here could hide for a long time.

routing *tracks*, which are unetched copper runs between pins. But KiCad proved catastrophically inept at even the simpler task. The issue was performance: the circuit was too large, and the CPU was too busy, for any plausible editing using a mouse.

So not only was KiCad awkward to use in every way in defining the circuit (not to mention being able to read it meaningfully), but KiCad also proved useless converting the circuit into even a component placement, let alone milling instructions for a board. There was another problem. There were only 864 pins in what I was editing, and KiCad would fare at least polynomially slower as the design expanded. Chapter 8's minicomputer already has 6 678 pins and still lacks bypass capacitors and more.

The whole idea of trying to do this graphically continued to seem wrong to me. KiCad wasn't to blame on that score. I would stubbornly go on to draw my own schematics in an SVG editor. These number 24 pages and have been of some value, although they now require extensive edits. Figures 3.2 and 3.3 (pp. 45 and 46) show two of these free-form schematics. But the SVG drawings don't capture any netlist information for simulation testing or circuit board automation. I would have to come up with a scheme that better fit the architecture's needs.

### **11.3.2 A typewritten netlist**

A non-graphical means for entering a machine-readable netlist was necessary, and it would have to be easy for me to use and require minimal learning by future maintainers. I had carte blanche in its syntax and semantics, so I considered what an easy implementation might be. Here is what I came up with.

#### **A tiny token-processing language**

Because several of the minicomputer's nodes have as many as 36 nets, it would be helpful to have a method for describing sequences of things. My approach was mo-



tivated from the brace expansion offered by the Bash command shell, which behaves like this:

```
$ echo {top,bottom}-{left,right}
top-left top-right bottom-left bottom-right

$ echo {A,B,C}{5,17,41}
A5 A17 A41 B5 B17 B41 C5 C17 C41
```

So here is what I came up with. An input file specifies netlists via a series of whitespace-delimited tokens. No tokens can contain whitespace. Everything in the file is case-sensitive.

A macro system provides substring replacement within tokens. Macros can have, but are not required to have, multiple replacements, causing tokens to multiply in the Cartesian sense. All macros are expanded from left to right.

Macro names begin and end with angle brackets; e.g., `<rock>`. Like tokens, macro names can't contain whitespace or internal angle brackets. There are four kinds of macros:

## KIND 1: USER SPECIFIED

If the netlist preamble contains the definition:

```
.macro <rock> roy harold scherer
```

Then these two lines are equivalent:

```
.named <rock>'s_part
.named roy's_part harold's_part scherer's_part
```

So now there are *three* parts. Yes, it's okay to use ' in names, because the macro system imposes very few rules about non-whitespace symbols. This first of the four kinds of macros is the only kind that uses a `.macro` directive in the netlist preamble to define it.

## KIND 2: RANGES OF NUMBERS

Simple up and down integer counting is offered. Zero-padding to the width of the lesser bound is supported. So these two lines are equivalent:

```
.pins = set<12-08>
.pins = set12 set11 set10 set09 set08
```

## KIND 3: REPETITION

To connect the five `set` pins to a single net that is named `.vdd`:

```
.wire set<12-08> = <5*>.vdd
```

Note the `.wire` directive works pairwise across `=`, so the above is equivalent to five individual `.wire` directives that make one connection each.

## KIND 4: SERIES OF NAMES

Use commas in macros to expand a set. Like this:

```
.wire <mon,tue,wed,thu,fri>.y = <tue,wed,thu,fri,mon>.a
```

This contrived example suggests a shift register that spans the weekdays. Five connections are created, starting with `mon.y` to `tue.a`.

The processing to implement the four kinds of macros is extremely simple. When a netlist is read, it is parsed to find occurrences of macro kinds 2, 3, and 4 before other processing is done. These macros are converted into and stored as multiple instances of kind 1, where the commas, dashes, and asterisks lose their special meaning and become part of the literal macro names. This transforms the macro substitution task into a trivial find and replace, leaving only Cartesian product expansion of token lists to be done after that.

The entire macro definition and expansion process is implemented in 104 lines of Python. It truly is a tiny language, but it pays big dividends. As an example, the

six RAMs for the ALU's  $\beta$  layer have a total of 600 pins, but their wiring takes only 48 lines to specify all of them! The  $\beta$  layer could have been a little tedious to key in, because of its incoming and outgoing transpositions per figures 5.1 and 5.3 (pp. 75 and 77). But the macro language needs only four lines to do *both* transpositions:

```
.macro <a-trans> <0-5>.a0 <0-5>.a1 <0-5>.a2 <0-5>.a3 <0-5>.a4 <0-5>.a5
.macro <d-trans> <0-5>.d0 <0-5>.d1 <0-5>.d2 <0-5>.d3 <0-5>.d4 <0-5>.d5
.wire beta<a-trans> = B.l<0-35>
.wire beta<d-trans> = B.y<0-35>
```

Without macros, 72 connections would have needed manual entry. Here are the first seven, which may be helpful understanding how the macros unroll:

```
.wire beta0.a0 = B.l0      ; (lowercase ell, not the digit one)
.wire beta1.a0 = B.l1
.wire beta2.a0 = B.l2
.wire beta3.a0 = B.l3
.wire beta4.a0 = B.l4
.wire beta5.a0 = B.l5
.wire beta0.a1 = B.l6
```

The transposition macros can't be abbreviated further, because the left-to-right expansion order would nullify the transposition. This would not work:

```
.macro <a-trans> <0-5>.a<0-5>
.macro <d-trans> <0-5>.d<0-5>
.wire beta<a-trans> = B.l<0-35>
.wire beta<d-trans> = B.y<0-35>
```

because we would get an untransposed connection sequence starting with:

```
.wire beta0.a0 = B.l0      ; (lowercase ell, not the digit one)
.wire beta0.a1 = B.l1
.wire beta0.a2 = B.l2
.wire beta0.a3 = B.l3
.wire beta0.a4 = B.l4
.wire beta0.a5 = B.l5
.wire beta1.a0 = B.l6
```

## Netlist file structure

Lines within the file are just lines, and there is no line-splicing mechanism. Contiguous whitespace within a line acts like a single space, separating runs of non-whitespace characters into tokens. Leading and trailing whitespace is ignored, as are blank lines. A semicolon introduces a comment to the end of the line it appears on.

Multi-line comments are available via `.off` and `.on` directives that suspend and resume parsing. They need not be balanced in any fashion. Here is an example of their use:

```
.off
How about those Buckeyes!
I mean THE OHIO STATE UNIVERSITY BUCKEYES.
Not some other, wannabe buckeyes.

Below is a circuit inspired by the Buckeyes.
.on
```

Another kind of comment is available to get the operator's attention when a netlist is processed. An `.attn` directive prints text to the screen next to the line number the directive is on. This makes the run "look unclean," because netlist processing is usually silent. Here is an example:

```
.attn Be sure to reconnect the branch decision pin!
```

A netlist file has a first, second, and third section called *phases* where different types of directives are allowed. There is no phase demarcation in the file itself, but a phase 1 directive may not appear after the first phase 2 directive, nor may a phase 1 directive appear after a phase 3 directive.

The only phase 1 directive is `.macro`, which has already been described and defines kind 1 macros. This instinctively makes sense: a macro must be defined prior to use.

Phase 2 directives define the types of components that go in the netlist and creates specific instances of them. For example, phase 2 is where the netlist explains

what a pull resistor is, and further explains how many there are and gives them all names. This too makes sense: components must exist prior to use, and their attributes must be known before they can be created with these attributes.

Phase 3 directives connect existing components together, most often with *wires* implemented as copper tracks on a circuit board, but sometimes with glue logic in addition to wires.

## Defining and using component types

The file needs adequate descriptions of each kind of component used. Each of these descriptions begins with the `.kind` directive, and a short block of parameters. Thus a new component may look like this:

```
.kind dual-nand
.footprint Package_S0:VSSOP-8_2.4x2.1mm_P0.5mm
.lib 74xGxx
.part 74AUC2G00
.value 74AUC2G00
.order Texas Instruments SN74AUC2G00DCUR
.mm 2.1 3.2
.npins 8
```

Some of the parameters are used only by KiCad. Although I rejected KiCad as offering utility for drawing schematics, producing netlists, or placing components, I have yet to evaluate KiCad for routing tracks. I am not optimistic, but the file format includes information that can be exported to KiCad.

Here is a list of directives that apply to a component as a whole, as opposed to applying individually to a pin:

- `.kind`      This name determines the behavior that the simulator will confer on this kind of component. Pick a name that you like. (This name will need to be present in the simulator's source code.)
- `.npins`     This is the number of pins on the component. The information is used when checking for human mistakes in the file, and for determining if a

component is “real.” It’s sometimes helpful to define non-physical, “conceptual” components or circuit sections to simplify and document wiring. Omitting `.npins` makes a component conceptual. Conceptual components do not undergo rule checking and are not exported anywhere.

- `.order` This is the part number that will be given in the generated bill of materials. This should be as specific as possible, with packaging specified and such, so the generated bill of materials can be used to create purchase orders. This directive permits multiple-token values (embedded spaces are allowed).
- `.lib` This information is only exported to KiCad.
- `.part` This information is only exported to KiCad.
- `.value` This information is only exported to KiCad. This directive permits multiple-token values (embedded spaces are allowed).
- `.footprint` This information is only exported to KiCad.
- `.price` A best-estimate cost to purchase one component of this kind, given as a floating-point number. The generated bill of materials will total these.
- `.mm` Maximum bounding box of this component, in mm, including pins but not the soldering footprint. Specify width and then height. This information is used for estimating circuit board size.
- `.rect` Linear pin geometry. Not suitable for ball grid array parts. Seven space-separated numbers must appear on this line in this order:
  - `dw` separation between the north-south rows of pins
  - `dh` separation between the east-west rows of pins
  - `pitch` pitch between centers of adjacent pins
  - `west` number of pins on left side

**south**    number of pins on “bottom” side (often 0)

**east**     number of pins on right side

**north**    number of pins on “top” side (often 0)

All linear measurements millimeters between nominal pin centers. If south and north are both 0, **dh** is irrelevant since there are no pins to separate. Pin 1 is the northmost on the west side. From there, pins are consecutively numbered going counterclockwise. Information from **.rect** is used for board layout, estimating track lengths and capacitances based on component placement. If you have an odd part with a triangle geometry or something, just fabricate something here that’s good enough for simulation. A part’s location affects timing more than its pin positions.

**.noload**    Instructs the simulator not to model propagation delay associated with capacitive loads of this component.

**.load**      Describes capacitances. Four parameters must appear in this order:

**in\_pF**            capacitance of each input pin

**io\_pF**            capacitance of each I/O pin

**delay\_pF**        test capacitance for datasheet’s propagation delay

**ns\_pF\_slope**    added nanoseconds per picofarad over **delay\_pF**

**.pins**        This directive indicates the function for each physical pin. Its name is easily confused with **.npins**, which indicates the number of pins. Here is a one-line example of **.pins**:

```
.pins <1-8> = 1a 1b 2y gnd 2a 2b 1y vdd
```

**.pins** can be used as many times as needed until all pins have been specified. It is not required that everything go on one line. If a **.pins** line only defines one pin, it is still **.pins**, not **.pin**.

To the left of the = sign are the pin numbers, 8 in all. The ordering and number convention are yours to control. For ball grid array (BGA) packages, these numbers are usually alphanumeric instead of numeric.

For conceptual devices (not real devices), there are no physical pins, so the pin numbers are omitted. But the = sign stays.

To the right of the = sign are pin names corresponding pairwise to the pin numbers on the left. There aren't many naming rules, but spaces and semicolons do not work in pin names.

The above were all whole-component directives. There are also directives that apply specifically to individual pins and groups of pins. First, the usage of each pin must be indicated. Continuing the example for **dual-nand**:

```
.power vdd gnd
.in      1a 1b 2a 2b
.out     1y 2y

.oute    ; used for output pins that have output enable
.bidir   ; used for bidirectional pins
.nc       ; used for pins to be left disconnected
.pull    ; used to mark the weak side of pull resistors
.pflag    ; used to indicate availability of power
```

The last five directives, **.oute** through **.pflag**, don't identify any pins, because they don't apply to **dual-nand** devices. It is more legible to omit these in component descriptions, but leaving them in without pins has the same effect on the netlist.

**.power** marks pins that require power. It is different from **.pflag**, which indicates that power is available. These are for rule checking to hunt for human mistakes: every pin that requires power must be in a net that has power available.

**.in** and **.out** mark pins used as inputs and outputs.

At this point, all directives that describe components have been briefly explained. There is one more directive in phase 2, and it is used to instantiate and name com-



ponents. Here is an example of its use, where 50 `dual-nand` packages (total of 100 NAND gates) are created and given names:

```
.named nand<01-50>
```

Note that the names like `nand01` have no bearing on what the gates *are* or *do*. NAND gates can as easily have names such as `xor01` or `alabama`. It is the `.kind` directive that tells the simulator what the device does.

### Connecting pins together

Phase 3 of a netlist file wires components together. A `.wire` directive works pairwise across an `=` sign. For instance, to connect `nand01.1y` to `nand33.1a` and `nand02.1y` to `nand33.1b`:

```
.wire nand01.1y nand02.1y = nand33.1a nand33.1b
```

Macros can get a lot done without much typing. Let's power those 100 NAND gates from the previous section:

```
.wire nand<01-50>.<vdd,gnd> = <50*>circuit.<vdd,gnd>
```

When wiring circuits, the exact pin of the exact component is specified, always joined with a period without internal whitespace. Component names may also contain periods, but the rightmost period within a given token always sets off name of a pin as provided via the `.pins` directive for that component's kind.

### A special provision for clock driver wiring

A `.short` directive is used to connect buffers in parallel for clock distribution. These parallel connections increase available drive and (kind of) maintain a single concept of the clock waveform and try to minimize skew. The KiCad operator will have to work some magic getting these pins connected with equal-length tracks. Here's the syntax (note there is no `=` sign):

```
.short 1866 source.out<0-3> dest.in<0-7>
```

It's always **.short**, then an integer delay in picoseconds, and then some number of input and output pins. Order of pins does not matter, because they all get connected together.

On the KiCad side, **.short** connects a single track to all of the pins; that is, the multiple outputs drive some large fan-out of inputs. The circuit board designer will have to combine the output lines to a common point via equal-length paths, and then distribute to each input by equal-length paths, such that the total track delay from any output to any input is (in this example) 1866 ps.

The simulation software is not intended to handle such fan-ins and fan-outs, and even if it could, it knows nothing of waveforms, rise and fall times, signal distortion, or noise. Instead, what the simulator receives from this script is that each destination is driven by just one of the sources with the specified path delay (1866 ps in this example), with the sources chosen round-robin to keep the fan-out within the simulator's configured **fan\_out\_max**.

For design rule checking, only one output pin is checked in any **.shorted** net. This prevents generation of “out with out” error messages.

## Glue logic synthesis

A **.glue** directive permits the user to write a logic function in reverse Polish notation (RPN), with the line being interpreted as if the user had typed in the requisite **.named** and **.wire** directives to implement the gate instances and wiring. This can save a lot of human labor and errors. But there are some limitations:

- The only gates are **nand**, **nor**, **not**, **and**, **or**, **notnot**, and **xor**. Like directives, these are lowercase names.
- The SN74AUC2Gxx chips are used, where xx = 00, 02, 04, 08, 32, 34, and 86 for the above. Note that all of these chips are dual-gate packages, and the reason I

use these is that they are faster than their single-gate counterparts, and easier to solder by hand than their leadless quad-gate counterparts.

- These devices must be declared in Phase 2 and have a `.kind` known to this script, which is `dual-nand`, `dual-nor`, .... (They must not have a `.named` directive, because `.glue` instantiates and names its gates automatically.)
- `not` and `notnot` have one input; the others have two.
- I didn't want to reserve the word `buf` to mean buffer, so it's called `notnot`.

Usage looks like this (and operations must be lowercase):

```
.glue out.0 = in.0 not
.glue out.1 = in.0 in.1 xor
.glue out.2 = in.0 in.1 and in.2 xor
.glue out.3 = in.0 in.1 and in.2 and in.3 xor
```

This example has 3 xor gates, 2 and gates, and an inverter. The `.glue` directive factors out redundant calculations, so the `and` computed for `out.2` simply fans over as an input for `out.3` without waste.

Sometimes a high-current load needs to be driven, and a low-delay method for this is to use two copies of the last gate in parallel. This is signified by prepending `2*` to the boolean operation. For instance:

```
.glue parity.y = p.a p.b xor p.c 2*xor
```

Doubling always is done in a single physical package. So the `p.a p.b xor` will not be on the same chip as the final, high-current `xor`. There is no support for tripling, quadrupling, etc.

A `.cohort` directive prevents `.glue` from using two halves of a dual IC for distant parts of a circuit board. To share the same chip, two gates must be glued within the same cohort. There are two forms of the directive:

```
.cohort                ; from here forward is a unique cohort
.cohort mypart         ; from here forward is a cohort named mypart
```

The first form is simple: the following `.glue` forms a cohort where its gates can share one chip. Any unused halves are abandoned when the next cohort is entered. The second form lets us return to a cohort later in a file and use up any unused halves of gates from that cohort. The reason we offer the second form is that the logical structure of your netlist may not always correspond with the order that glue logic is specified in.

The `.cohort` directive only affects the operation of the `.glue` directive. `.cohort`'s use is not mandatory, although very advisable in large circuits. `.cohort` has the added benefit of including the cohort name as part of the generated part's name, which assists in part identification during the manual task of specifying component positions.

A nontrivial example of `.glue` alongside `.cohort` and a conceptual component (one that doesn't have any ICs, but exists to help organize the netlist) is the instruction pointer implementation shown in listing 11.4. The first page of its 3½-page free-form schematic is in figure 3.3.

As a human factors experiment to assess how straightforward `.glue` is to use, I started a timer before translating the schematic into what turned out to be 33 `.glue` directives, typing them in as I progressed. I thought all was going well, but the result was not correct, and I had to start again from the beginning. That result did work and has not been changed in more than a year since. The total time it took me to manually convert the schematic into typing, including the false start, was 49 minutes and 2 seconds to specify connections for all 278 pins and 35 ICs. This glue logic synthesis capability is easy and fast to use, and tends to preclude human mistakes.

## **Diagnostic features**

There is a `.draw` directive for diagnostic use. `.draw` is followed by a list of pins. For each pin given, an SVG file named after the pin is generated with a drawing of the net the pin is in.

```

; -----
; IP incremter conceptual component and glue logic
; -----
.kind ip_incremter
.pins = in<0-26> out<0-26> stage<0-5>
.named ipi

.cohort incremter
.glue ipi.out0 = ipi.in0 not
.glue ipi.out1 = ipi.in0 ipi.in1 xor
.glue ipi.out2 = ipi.in0 ipi.in1 and ipi.in2 xor
.glue ipi.out3 = ipi.in0 ipi.in1 and ipi.in2 and ipi.in3 xor
.glue ipi.stage0 = ipi.in0 ipi.in1 and ipi.in2 ipi.in3 and and

.glue ipi.out4 = ipi.stage0 ipi.in4 xor
.glue ipi.out5 = ipi.stage0 ipi.in4 and ipi.in5 xor
.glue ipi.out6 = ipi.stage0 ipi.in4 and ipi.in5 and ipi.in6 xor
.glue ipi.out7 = ipi.stage0 ipi.in4 and ipi.in5 and ipi.in6 and ipi.in7 xor
.glue ipi.stage1 = ipi.in4 ipi.in5 and ipi.in6 ipi.in7 and and ipi.stage0 and

.glue ipi.out8 = ipi.stage1 ipi.in8 xor
.glue ipi.out9 = ipi.stage1 ipi.in8 and ipi.in9 xor
.glue ipi.out10 = ipi.stage1 ipi.in8 and ipi.in9 and ipi.in10 xor
.glue ipi.out11 = ipi.stage1 ipi.in8 and ipi.in9 and ipi.in10 and ipi.in11 xor
.glue ipi.stage2 = ipi.in8 ipi.in9 and ipi.in10 ipi.in11 and and ipi.stage1 and

.glue ipi.out12 = ipi.stage2 ipi.in12 xor
.glue ipi.out13 = ipi.stage2 ipi.in12 and ipi.in13 xor
.glue ipi.out14 = ipi.stage2 ipi.in12 and ipi.in13 and ipi.in14 xor
.glue ipi.out15 = ipi.stage2 ipi.in12 and ipi.in13 and ipi.in14 and ipi.in15 xor
.glue ipi.stage3 = ipi.in12 ipi.in13 and ipi.in14 ipi.in15 and and ipi.stage2 and

.glue ipi.out16 = ipi.stage3 ipi.in16 xor
.glue ipi.out17 = ipi.stage3 ipi.in16 and ipi.in17 xor
.glue ipi.out18 = ipi.stage3 ipi.in16 and ipi.in17 and ipi.in18 xor
.glue ipi.out19 = ipi.stage3 ipi.in16 and ipi.in17 and ipi.in18 and ipi.in19 xor
.glue ipi.stage4 = ipi.in16 ipi.in17 and ipi.in18 ipi.in19 and and ipi.stage3 and

.glue ipi.out20 = ipi.stage4 ipi.in20 xor
.glue ipi.out21 = ipi.stage4 ipi.in20 and ipi.in21 xor
.glue ipi.out22 = ipi.stage4 ipi.in20 and ipi.in21 and ipi.in22 xor
.glue ipi.out23 = ipi.stage4 ipi.in20 and ipi.in21 and ipi.in22 and ipi.in23 xor
.glue ipi.stage5 = ipi.in20 ipi.in21 and ipi.in22 ipi.in23 and and ipi.stage4 and

.glue ipi.out24 = ipi.stage5 ipi.in24 xor
.glue ipi.out25 = ipi.stage5 ipi.in24 and ipi.in25 xor
.glue ipi.out26 = ipi.stage5 ipi.in24 and ipi.in25 and ipi.in26 xor

```

**Listing 11.4:** Instruction pointer incremter (figure 3.3) implemented.

A design rule checker is included that is characteristic of some schematic editors and VLSI design software. This is worth the 50 lines of Python it took to implement, because the timing-oriented electrical simulation doesn't test a number of these cases. The design rule checker verifies that all nets are free of these mistakes:

- power pin by itself
- power pin with output pin
- power pin with tristate pin
- power pin with bidirectional pin
- power pin without power available marker
- more than one power available marker
- power pin with pull resistor load side
- more than one pull resistor
- input pin by itself
- input pin without a pull resistor or output/tristate/bidirectional/power pin
- more than one output pin
- output pin with pull resistor
- output pin with tristate pin
- output pin with bidirectional pin
- bidirectional pin by itself

## **Terseness of netlist specification**

The file described in this section (11.3.2) is 1 941 lines as of November 2022. The first 890 lines specify component type details, how many of each there are, and what their names are. There are also some preliminary comments, and ten macros are defined. The connections are made in the remaining 1 051 lines. This is a very convenient and efficient mechanism for specifying and maintaining a minicomputer netlist.

### **11.3.3 Component placement**

A non-graphical means of choosing and describing component positions and their orientations was needed. To do this optimally is probably an NP-complete problem, so two heuristics were used. The first heuristic was, I made an “educated guess” as to where everything would go. This is to say, software was not used to directly position components. But with 220 components, I would need another heuristic to simplify my choices.

As section 8.1 mentions, the components fall into three size categories that I named *small*, *medium*, and *large*. I partitioned the circuit board into a  $10 \times 8$  grid where the large ICs could be placed. These are the SRAMs, and they have 100 pins. The grid is not isotropic, but has an aspect ratio to approximate the shape of the SRAM ICs. This grid’s uniform offsets for SRAMs are easily visible in figure 8.1 (pp. 180–181). I looked at the remaining components and determined that two medium components—the ones with 48 pins—can fit in one grid space, or nine small components, or one medium with five small components. All I needed to do was manually assign related components onto *tiles* that fit in one grid space each, and then arrange the tiles as best I could on the grid.

Listing 11.5 is a sample of a tiny language I concocted to specify what components go on which tiles. The tiles have names like spreadsheet cells, with range **a1** through **j8**. I summarized what each tile did on slips of rectangular paper, arranged them on

```

.....      ....      ....
374.37      ....      ....      ....
.....      nor      and      nand
.....      ....      ....      : :
.....      and      xor      or
374.38      ....      : :      ....
.....      ....      : :      ....
nand      nand      and

.....      ....
374.40      flop
.....      ....
.....      flop
.....      ....
or      xor      xor

.....
D0

.tile h2
374.37
374.38

.tile i2
glue-lockouts-nor-1
glue-lockouts-and-5
glue-lockouts-nand-2
east: glue-lockouts-and-6
glue-lockouts-xor-0
north: glue-lockouts-or-1
glue-lockouts-nand-3
south: glue-lockouts-nand-4
glue-lockouts-and-7

.tile h3
374.40
ffeflop
ffsflop
glue-call-depth-ctr-or-0
glue-call-depth-ctr-xor-0
glue-call-depth-ctr-xor-1

.tile i3
D0

```

**Listing 11.5:** Component placement syntax showing four circuit board tiles.

a table, and typed up the eventual placements. Listing 11.5 shows the specification for four of figure 8.1’s tiles. This happens to be the only  $2 \times 2$  space in the grid having all four tile population schemes, having a maximum of one, two, six, or nine components depending on their sizes. Within a tile, component names appear from left to right, repeating from top to bottom.

There are three subtleties to mention about listing 11.5. First, the effect of my grouping by component function is visible. Tile `i2` has nine ICs that have `lockouts` as part of their name. I did not name these components, because they were automatically generated by `.glue` directives. But I did have a `.cohort lockouts` prior to making those connections, which is where the `lockouts` part of their name came



from. Likewise for tile `h3`, there is a `.cohort call-depth-counter` in the netlist.

Another subtlety is component orientation. In packages with two rows of pins, pin 1 is usually shown on datasheets at the top left, and the highest-numbered pin at the top right. For both, the top is involved, and so I call this orientation *north*. For *east* components, the lowest and highest-numbered pins are on the right side, and so on. Where orientation is not specified, the small ICs all face west, and the large ones face north. The medium ICs face west if there are two on the tile, otherwise they face north. It turns out that tiles `h2`, `i2`, `h3`, and `i3` have all of their components at their default orientation in figure 8.1, so for listing 11.5, I rotated three components on tile `i2` to illustrate how their orientation is overridden.

The final subtlety is off-topic, except it’s so glaring in the listing. ICs 374.37 and 374.38 have terrible names. The 374 makes it obvious they are—according to the manufacturer—16-bit flip-flops, but what do they do, and why did I not name them for it? The reason is that these are dual 8-bit flip-flops, in the sense that each IC has two clock and two output enable pins. Having more than one section, these ICs can have more than one purpose, and this creates a naming problem. In the netlist, conceptual components are used to represent the eight-bit halves, and these are given names based on what they actually do. These halves eventually get “snapped into” physical ICs using `.wire` directives near the end of the file. But a drawback of the process is that the conceptual components and their names are stripped out of further processing, and this makes setting up and interpreting electrical simulations more tedious. Useful names should persist through the whole toolchain.

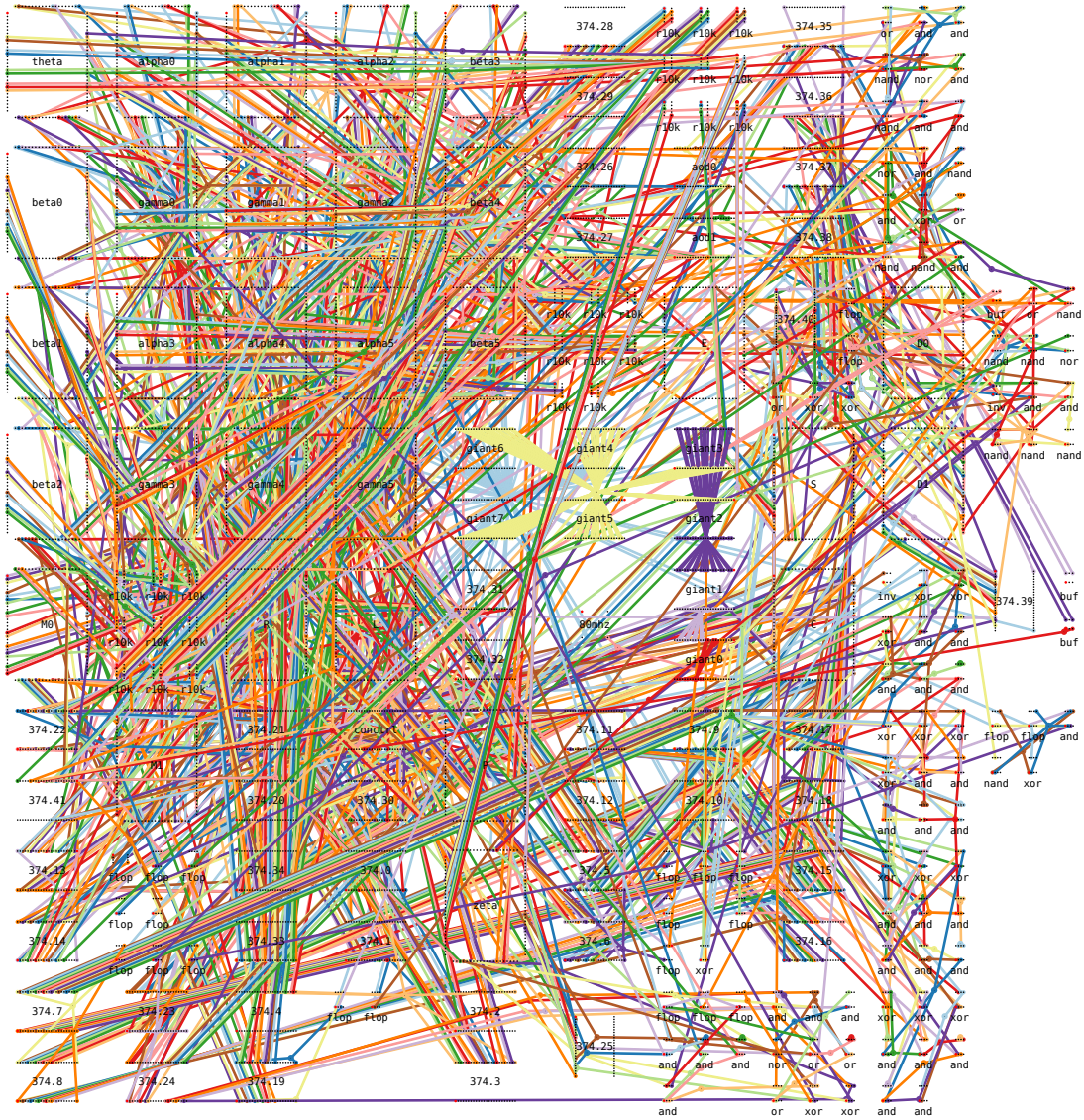
### 11.3.4 Netlist summary information output

The typewritten netlist (section 11.3.2) and component placement files (section 11.3.3) are processed through a Python 3 script. As of November 2022, this script is 2618 lines long and perhaps should be split into easier-to-digest sections. It produces five kinds of output.

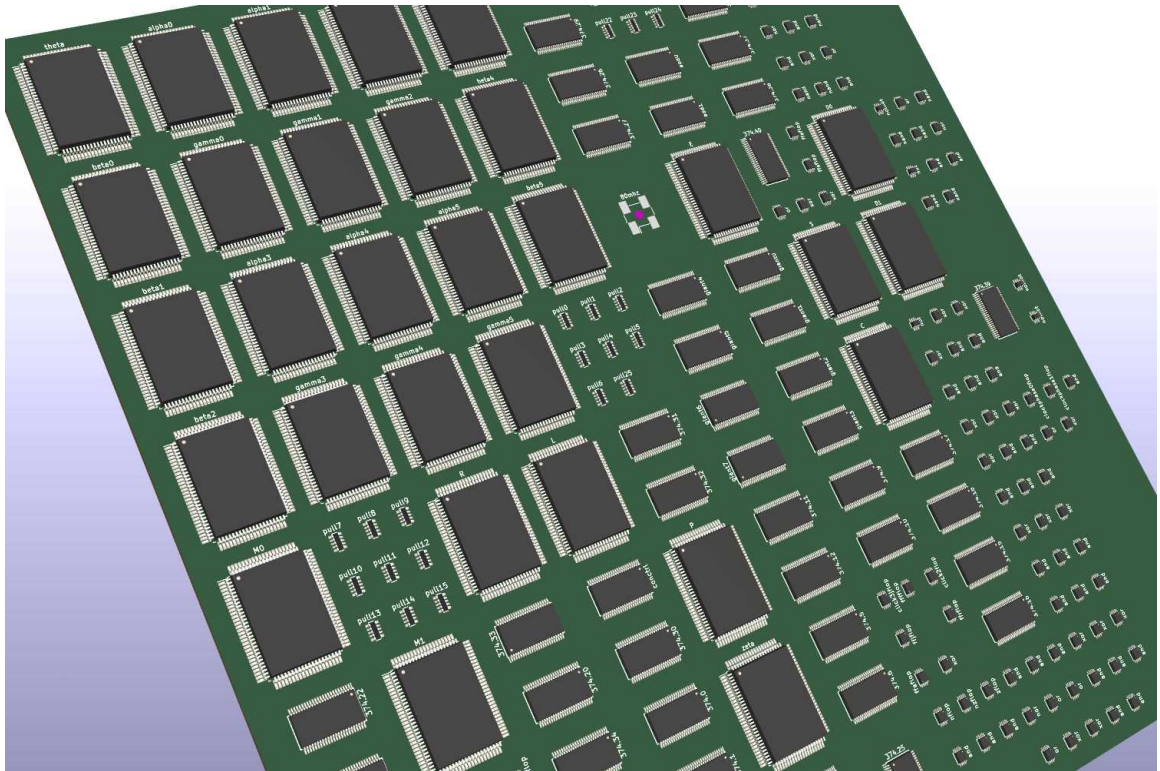
- A bill of materials file is written, showing how many of each `.kind` of IC, their total `.price`, and the part number to `.order`. A total count of parts, pins, and nets is given. Circuit board dimensions are estimated as a function of tile separation. A list of dual-gate glue logic ICs that have an unused gate is given. A list of the names of all components is also included.
- A floorplan is drawn that looks like figures 8.1 and 8.2 (pp. 180–181 and 182).
- Pin-to-pin connectivity between pins can be drawn underneath a floorplan. This can be sampled to include only a percentage of nets, or specified using `.draw` to draw only an identified net. Figure 11.3 shows a connectivity drawing for the whole chapter 8 minicomputer. Connections to  $V_{DD}$  and ground are not drawn to improve legibility.

For each net drawn, a point is found on the board that locally minimizes the sum of distances from that point to each pin in the net. This point is that net's *hub*, and it may not be globally minimal. The hub is drawn as a small dot, and straight lines of the same color are drawn from the hub to each pin. A limited palette of colors is used, so one color will typically denote many nets. But where colors are different, they are guaranteed to be electrically separated. Lines do not indicate any proposed length or routing of a track, but only show relationships among nets and pins.

- A netlist is written in KiCad's format for possible use to route tracks. KiCad is able to draw pictures of circuit boards using imported netlists, although its output resolution is limited by the size of the display used while drawing. Figure 11.4 shows KiCad's drawing of a chapter 8 minicomputer.
- A component list with propagation delay estimates for all connections is written for use by the electrical simulation. The next section explains the assumptions that undergird these estimates.



**Figure 11.3:** A drawing of the chapter 8 minicomputer's non-power connections.



**Figure 11.4:** A drawing by KiCad of the chapter 8 minicomputer.

```

kind flop                ; create one component of kind 'flop'
name ffiflop

kind 16374                ; create three components of kind '16374'
name 374.0
name 374.1
name 374.2

from ffiflop              ; five signal paths
to 374.0
signal q 1oe# 750          ; from ffiflop.q to 374.0.1oe# takes 750 ps
signal q 2oe# 737          ; from ffiflop.q to 374.0.2oe# takes 737 ps
to 374.1
signal q 1oe# 700          ; from ffiflop.q to 374.1.1oe# takes 700 ps
signal q 2oe# 669          ; from ffiflop.q to 374.1.2oe# takes 669 ps
to 374.2
signal q 1oe# 764          ; from ffiflop.q to 374.2.1oe# takes 764 ps

```

**Listing 11.6:** Connectivity syntax example with five signal paths. Typical use is without comments or blank lines, and abbreviates the keywords to **k**, **n**, **f**, **t**, and **s**.

### 11.3.5 Estimating propagation delay between pins

The hardest task for the previous section's Python script is to prepare a list of all signal paths on the circuit board, including a plausible propagation delay estimate for each path. This is written to a file in the format of listing 11.6. For the chapter 8 minicomputer implementation as of November 2022, the file comes to 6 754 lines, which is close in order of magnitude to the 6 678 total component pins. The time granularity for simulation is 1 ps, so propagation delays are given in that unit.

Propagation delay is estimated on the basis of total net capacitance and the track length from the source pin to destination pin. Because the simulation is only intended to provide a conservative estimate as to attainable CPU speed, track layouts are guessed so as to be plausible, but in many cases not close to optimal. The hub and spoke model of section 11.3.4 attempts, to the nearest whole millimeter on the x- and y-axes, to place a net's central connection point (hub) such that the combined

distances from the hub to each pin of the net is minimized. The script is content with a local minimum, and I believe there are situations where a more optimal hub placement may be found.

One defect with the hub and spoke model is that long redundant runs occur in places. Consider a hypothetical net with four collinear pins, all having x-coordinate 0, and having y-coordinates 0, 5, 95, and 100 mm. The hub will be at (0, 50). An ideal track with no obstructions would be a 100 mm line segment from one end to the other, and the track's capacitance will be based on 100 mm of copper. But this is not the model implemented. The hub indeed is at (0, 50), but the track calculation naively uses four line segments with lengths 50, 45, 45, and 50, which contribute 190 mm of copper to the capacitance.

Although shortest routes are drawn in figure 11.3 to show connectivity, shortest routes between pins are not used to estimate track lengths. It is assumed rather that layout rules constrain tracks to orientations that are whole multiples of 45 degrees. Due east-west, north-south, northeast-southwest, and northwest-southeast are the only directions tracks may run. Figure 11.5 shows an approximately 10% sampling of the non-power nets with these track assumptions. As with figure 11.3, this figure depicts a timing and capacitance estimate and not a track layout. This would be blatantly evident if figure 11.5 were to try drawing all of the nets, because north-south and east-west runs would overlap rows of pins on a grand and physically impermissible (not to mention illegible) scale. Many diagonal runs would also overlap.

For track length computations, every pin is presumed to have an extra 5 mm of track that connects it for the purpose of evading obstacles. This may not be enough allowance for estimation purposes, but it can be revisited easily following some experience routing finalized, manufacturable tracks on the board.

The capacitance of a net is estimated as the sum of capacitances for all of its pins. The capacitance of any presently-driven output pin is excluded, because that component's datasheet already accounted for it in propagation delay estimates. Added to





this capacitance is a track estimate of 0.08 pF per mm from all spokes, including the allowance for dodging obstacles.

The pin-to-pin propagation delay is computed from the component's datasheet, with a straight-line approximation added for the net's capacitance in excess of the datasheet's test condition. The amount of time to add per unit of capacitance comes from either a second load condition on the datasheet, when available, or estimates taken from comparable datasheets. Glue logic that are in tandem pairs for higher drives is modeled by halving the load capacitance for timing estimates. In addition to the datasheet's propagation delay and excess capacitance propagation delay, additional delay for track length is assessed at 6 ps/mm. The applicable length is directly along the two spokes through the hub, again including the length allowance for getting around obstacles.

## **11.4 Electrical and timing simulation**

Demonstrating that my architecture works is not the purpose of simulation, because only construction can do that. The purpose of simulation is to search quickly for ways that the netlist may be broken or inadvertently incomplete. Knowing in advance that a circuit will fail allows the step of building, testing, and troubleshooting a broken physical model to be skipped. On the other hand, a netlist with a clean simulation result instills confidence for taking on more risk at a time when circuits need to be physically built and validated. An effective simulation tool can immensely speed the development process. Previous simulations that I wrote for other projects have shortened hours to moments, and shortened months to hours.

### **11.4.1 Off-the-shelf simulation software**

Several circuit simulation tools are already published, and in theory at least, one or more of these could be applied to test the minicomputer netlist. I decided to write my



own simulator notwithstanding the time it would take to write or the risk of errors and oversights on my part. Here are the reasons I did this.

- Existing tools are sufficiently capable as to have a large learning curve.
- I believed I could write a simple tool in about as much time as I could learn an existing one.
- I would have full knowledge of the capability and use of what I write myself.
- I could decide all features and parameters for software that I write myself.
- Writing my own simulator eliminates all risk of selecting an undesirable existing one.
- Existing tools would require further inputs, particularly component simulation models. I would have to write any models I could not obtain from a manufacturer.
- At least one manufacturer in my bill of materials requires a nondisclosure agreement as a precondition of obtaining models. My implementation would not be allowed to distribute such models, rendering my simulation useless to (and possibly untrusted by) others.
- I would have no control over the speed of existing simulation software. For a circuit with thousands of pins and a need to monitor timing closely, this is a huge concern. An analog electrical simulation would be prohibitively slow, but a non-electrical logical simulation would not detect timing problems.
- I would have no control over the size, quality, documentation, suitability, or ease of use of an existing tool.
- An existing tool would introduce external dependencies, possibly many.

- An existing tool would create sensitivity to version mismatches.
- I happen to enjoy writing code and do it fairly well.

A case that sounds convincing could be made that with an open hardware specification, existing, well-validated tools that are already familiar in the field should be used. That case didn't win this round, but here is one more reason my architecture is open: anyone is welcome to take on that mantle.

### 11.4.2 Electrical simulator description

I wrote a simulator that can load the netlist, initialize the 22 firmware-containing RAMs with their firmware, assemble programs into the code RAM, force the circuit into a consistent state, set the instruction pointer to address zero, and start the CPU as the netlist describes it.

Because there are many components, trace output can be turned on and off on a per-component basis. A few pseudo-component types are defined for diagnostic purposes, and serve functions such as DIP switches and LEDs. But unlike their real counterparts, the switches can be changed and the LEDs can be checked with picosecond granularity. These pseudo-components have been useful for small regression test scripts for the real components, but they haven't found much use in the minicomputer simulation. The crazy state the netlist powers up in generates many pages of error messages, so they can be suppressed until a fixed point in time that the machine is known to have stabilized. Component parameters can be changed on the fly, such as changing the oscillator frequency, clocking known state into 8-bit flip-flops, write-protecting firmware RAMs until the system has stabilized, and forcing a RAM to do a one-time read cycle to break an infinite loop between the control decoder and code RAMs.

In addition to component trace output, the simulator has a regression test capability with scriptable tests of output pin states. This is a very limited feature,

because only one pin can be sampled per line of testing instructions. Evaluation of simulator output will need better automation for regression testing. Most simulator runs are evaluated manually as of November 2022. This evaluation usually comes to three questions. First, does the program terminate as anticipated? Many problems would cause the simulation to run indefinitely. Second, does a run's output have any worrisome colors? Output is color-coded, and errors are immediately visible. Third, was the correct end-of-program result clocked into the appropriate register? When all three conditions are met, it's time to move on.

The simulator has an indirect output that can be obtained by inference: the maximum oscillator speed the netlist functions at. This speed is determined by manually changing the duration of the oscillator's half-period in the script and re-running the simulation. Repeating this a few times in a manually-imposed binary search can find the half-period at which the simulation succeeds with correct output, but adding another picosecond causes failures.

In addition to enabling manual identification of incorrect execution of assembler programs, the simulator detects the following invalid electrical and timing situations:

- A clock input pin is not low for long enough.
- A clock input pin is not high for long enough.
- A clock input pin is driven by a high-impedance output, a pin with unknown state, or two conflicting pins.
- A flip-flop sees a setup or hold time violation.
- A flip-flop that supports preset or clear encounters a removal or recovery time violation.
- A flip-flop's preset or clear input is driven by a high-impedance output, a pin with unknown state, or two conflicting pins.

- A flip-flop is clocked during a preset or clear.
- An SRAM sees a setup or hold violation on a clocked input.
- An SRAM input is clocked while driven by a high-impedance output, a pin with unknown state, or two conflicting pins.

The above error conditions are dynamic; that is, they are time-dependent problems that may occur during simulation. These represent approximately an eighth of the problems the simulator can report. The other error messages signal uninteresting, static problems such as misspelling a component name.

The electrical simulation is written in C, a language I have been comfortable with for a third of a century. Good alternatives to C could have been Fortran, which can sometimes optimize more aggressively than C to reach higher speeds, and Rust, which offers certain reliability guarantees. Of these three, C is most mainstream at present and may be less of a challenge for many maintainers. The program has no external dependencies, and at 3 719 lines (including comments and blank lines) is very compact. Half of this code is for the simulator proper, and the other half simulates specific component types.

Understanding the many pins, capabilities, and alternatives offered by synchronous SRAM ICs took required a lot of time to understand the datasheets. But when I wrote the device simulation code based on my understanding of these ICs, the task was strikingly simpler and faster than I anticipated. The SRAM datasheets were so thorough and unambiguous describing their state transitions, that a table-driven finite state machine for them worked from almost the first test.

The simulator is a discrete event simulation, where a priority queue delivers information from pins where outputs are produced to pins where inputs are consumed after the computed propagation delay for the path. The pin relationship is many-to-many in that one of several output pins can drive a net at one time, and a net may drive many input pins. For each input pin, the simulator maintains the state

of each output pin that drives it, which may be low, high, or uncertain, as well as the impedance of the the output pin, which may be low or high. An output pin's state and impedance are always transmitted separately via the priority queue, so that input pins receive these transitions independently at correct times.

When an output transition for state or impedance reaches the head of the priority queue, the affected input pin and that pin's component are placed in singly-linked "dirty" lists for recalculation. At the end of each picosecond that exists in the queue, all sources for each dirty pin are tabulated to determine that input pin's new state, which will be one of three possibilities:

**low**            The input is considered a logic zero.

**high**           The input is considered a logic one.

**uncertain**    The input is either (a) in a transitioning state that has not settled, (b) dependent on an upstream uncertain state, or (c) being driven high and low simultaneously by competing outputs.

Cause (a) for an uncertain input pin is a pending state transition. This is to accommodate the double-transition of outputs after SRAM reads: when an SRAM is read, the data at its output pins will become invalid—the word from the previous read will no longer be available—before the pins show the correct output for the current read. This relates to strongly to CPU integrity and needs to be modeled correctly. Section 3.4 explains why this behavior matters.

Cause (b) for an uncertain input pin happens when a gate cannot determine its output because one or more of its inputs is uncertain. In real life, the gate will choose a high or low output, possibly after a short metastable period. But rather than try to model metastability or forcing a misleading decision, the simulation will place the non-computable output in the uncertain state. This is a good thing for making simulation results interpretable, but it also causes "uncertainty loops" that hang the

simulated CPU on power-up. The simulation script must forcibly break these loops before anything can get done.

Cause (c) for an uncertain input pin is when outputs are fighting over whether a net should be low or high. This is normal for very brief periods at hundreds of input pins, but it is a high-current situation that could damage components if the matter is not quickly settled.

Because of the simulation's hundreds of components and thousands of pins, along with the necessity to simulate potentially long assembler programs at the pin level, speed of simulation is very important. This is why the C language, a heap-based priority queue of closures (callback functions with their arguments), dirty lists, and other fast approaches are important. The simulator would be difficult to parallelize other than by breaking an assembler program into independent smaller programs, which often would be impractical.

Careful attention is given to memory handling. At the end of any simulation, any memory that was allocated is freed and counted, and a brightly-colored message is written if the counts do not match. A simulator run with the Fibonacci program of listing 8.1 (p. 176) does 6 439 `malloc` calls and a matching number of `free` calls. Of these, 5 952 calls are to label every pin with its name, which only happens once.

### **11.4.3 Simulator test script semantics and example**

The simulation process follows a human-written script that has a few responsibilities. One is that component, connectivity, and timing information extracted from the netlist needs to be loaded. This is the hardware side of what will be simulated. Because this hardware information is specified in the same language as the human-written test script, it can be either a file loaded by or pasted directly into the test script.

The three-page listing 11.7 shows a working test script. The language is so domain-specific and ad-hoc as to be very quirky, and there are points in the simulator

```

cls                                ; clear display
note fibclean.ns                  ; print "fibclean.ns" at top of run

; Suppress error messages until 135 ns, because there will be 1599
; "invalid clocked value" warnings due to still-uncertain inputs.

unstable 135000                    ; suppress error messages up to 167 ns

load frozen.ns                    ; load converted netlist and signal timings

trace C                           ; trace the code RAM
trace L                           ; trace the left register file
trace R                           ; trace right register file
trace 80mhz                        ; trace the oscillator (its name is 80mhz)

; Artificially write-protect the code RAM and all of the firmware RAMs
; to prevent spurious writes before the CPU stabilizes.

tweak C ro
tweak D0 ro
tweak D1 ro
tweak alpha0 ro
tweak alpha1 ro
tweak alpha2 ro
tweak alpha3 ro
tweak alpha4 ro
tweak alpha5 ro
tweak beta0 ro
tweak beta1 ro
tweak beta2 ro
tweak beta3 ro
tweak beta4 ro
tweak beta5 ro
tweak gamma0 ro
tweak gamma1 ro
tweak gamma2 ro
tweak gamma3 ro
tweak gamma4 ro
tweak gamma5 ro
tweak theta ro
tweak zeta ro

```

**Listing 11.7:** Simulator script to start CPU and run Fibonacci program. (1 of 3)

```

; An oscillator half-period of 7757 ps is 64.46 MHz or 16.11 MIPS.
; This is the fastest speed that this netlist can run this program.

tweak 80mhz 7757          ; oscillator half-period is 7757 ps

; The two-bit counter for click 0 ... click 3 is starts up in an
; infinite loop of uncertain outputs, neither low nor high.
; Temporarily connect the these flip-flops' CLR# pins to ground to
; force their outputs low. Then run the simulation through 10 ns.

from rail
to clockphase0flop
signal lo clr# 100
to clockphase1flop
signal lo clr# 100
quit 10000
settle

; Switch the CLR# pins from ground to Vdd to allow the click counter
; to run normally. Then run the simulation through 20 ns.

to clockphase0flop
signal hi clr# 100
to clockphase1flop
signal hi clr# 100
quit 20000
settle

asm fibclean.a          ; assemble the Fibonacci program into code RAM

; At 47.5 ns, the two-bit click counter will be at click 2. The next
; rising clock will read the code RAM. Run simulation to that time.

quit 47500
settle

```

**Listing 11.7:** Simulator script to start CPU and run Fibonacci program. (2 of 3)



```

; The instruction pointer bounces around four 27-bit flip-flops, each
; constructed from two dual 8-bit flip-flops. To start reading code at
; location 0, we clock zeros into all of these flip-flops. To avoid
; conflicts, ff a, ff j, and ff r outputs are disabled, and ff f
; outputs are enabled.

```

```

tweak 374.5 a-=0          ; ff a (Address for code read/write)
tweak 374.5 b-=0
tweak 374.6 a-=0
tweak 374.6 b-=0
tweak 374.9 a+=0          ; ff f (From inst. ptr. incrementer)
tweak 374.9 b+=0
tweak 374.10 a+=0
tweak 374.10 b+=0
tweak 374.11 a-=0         ; ff j (Jump/call destination)
tweak 374.11 b-=0
tweak 374.12 a-=0
tweak 374.12 b-=0
tweak 374.15 a-=0         ; ff r (Return address)
tweak 374.15 b-=0
tweak 374.16 a-=0
tweak 374.16 b-=0

```

```

; The code and control decoder RAMs are at an impasse with "uncertain"
; outputs to each other. Tell the code RAM it's okay to read the first
; opcode for the control decoder. From that point on, the control
; decoder will be able to signal the code RAM to read. The "forceread"
; tweak does not persist, but only affects the next read.

```

```

tweak C forceread

```

```

; The simulator interprets the end of the script here as permission to
; run indefinitely now. The run will end when the HALT instruction is
; executed in the Fibonacci program.

```

**Listing 11.7:** Simulator script to start CPU and run Fibonacci program. (3 of 3)

source code that add strangeness to the script semantics. The listing comments explain much of what is happening. Here are some places where these comments may be expanded on.

The file `frozen.ns` is where the hardware information is taken from. Listing 11.6 annotates a few lines from this file. The file imported by `load` is not restricted to connectivity and timing information, but may contain any commands that are known to the simulator.

Although `80mhz` looks like a quantity, it is actually the name of an already-purchased crystal oscillator. Thus far, simulations indicate that some changes to the circuit would be necessary to use this part as the oscillator. The simulated top safe speed is 64.46 MHz as of November 2022.

A careful reading of where the two-bit counter is cleared to logic zero has an apparent contradiction. The `CLR#` pins on the flip-flops are connected to ground, and then *also* connected to  $V_{DD}$ . At face value, this would be electrically disastrous. Behind the scenes in the source code, the pseudo-component named `rail` has a magical property that no other component has. When `rail` is connected to a pin that has existing connections, a warning is issued and the existing connections are dropped. This is was done as a scripting convenience for regression testing. No pin is ever connected to ground and  $V_{DD}$  at the same time. The 100 ps propagation time specified from `rail` to the flip-flops is an arbitrary small positive interval.

The ill-named `quit` command specifies a time to pause the simulation and continue the test script on the next line. Somewhat inconveniently, `quit` requires an absolute time expressed in total picoseconds, so if more tests are cut and pasted in front of a series of `quits`, the script maintainer would have to update all of the times.

The `settle` command is also ill-named, because it causes the simulation to start (or continue). The simulation will run until either the current `quit` time has been reached or the signal queue is empty. The latter will not occur in a minicomputer, because the clock oscillator will keep placing transitions in the queue.

The kludgiest way for the simulation to stop running is for a **HALT** instruction to be executed. Behind the scenes, the simulator and netlist don't know what any instruction means, but the source code for simulating SRAMs monitors code RAM fetches. When a **HALT** instruction is read from the code RAM, a **quit** is entered for 25 ns later, at which time the **HALT** is presumably being executed by the CPU.

The **tweak** command sends one token (a string of characters without spaces) to the C code that is simulating a component. The string's format is chosen to make the C code that interprets it very simple. Sometimes that goal is at odds with the **tweak** command's legibility. In the 16 lines on the last page where the instruction pointer is being zeroed, the eight-bit halves of the flip-flops are written as **a** and **b**, the **-** sign means turn the outputs off, and **+** means turn the outputs on. The **=0** means to clock a decimal 0 into that half's eight bits. A presumptive reading of **a-=0** to mean "subtract zero from **a**" is not the right interpretation.

The **forceread** tweak on the code RAM overrides the simulator's pin semantics that unknown outputs are left unknown instead of being allowed to collapse onto low or high. When the opcode bits read from the code RAM are unknown, those bits go to the control decoder RAMs to determine the correct control signals for that opcode. All those control signals therefore become unknown. One of those control signals is the enable input **E1#** for the code RAM. The code RAM will enable if **E1#** is low, but not if **E1#** is unknown, so the next attempted code fetch will send another unknown opcode to the control decoder. It's an infinite loop. The **forceread** tweak tells the code RAM to pretend, for just the next clock cycle, that **E1#** is low and a read can proceed. This breaks the final stalemate that was keeping the CPU from running.

The end of the test script is peculiar, because there is an implicit **settle** in how it is interpreted. So when there appears to be nothing left to do, the simulation resumes. To stop at the end of the script, **quit 1** can be written, and if at least one picosecond was simulated, the test will terminate.

#### 11.4.4 Simulator output example

A run of listing 11.7's test script produces 4944 lines of output. Double-page listing 11.8 shows the final lines of this run. Some output that is useful for certain runs, but not this one, was removed to improved legibility. This included octal and hexadecimal translations of register values, and oscillator trace output for clicks 1, 2, and 3. The brown click 0 trace output occurs at high-to-low oscillator transitions.

Several features of the CPU's operation can be seen in listing 11.8. Out of 30 SRAMs, only the code RAM and two register file copies are traced in this listing, and none of the flip-flops are traced. The oscillator can be seen running in brown, code RAM reads show the fetched opcodes and operands as red decimal numbers, and tandem stores to the left and right register files appear in blue. The direction transitions of all three RAMs' data lines is visible, with the outputs coming on (low impedance) and turning off (high impedance).

The last line of output, in green, directs the operator to a file named **fire.report**. This report is a tabulation of overcurrent occurrences during the simulation. Many nets are driven by different output pins at different times, and temporal tolerances are tight. One output may be enabled picoseconds before its neighbor is disabled. Or an output may transition to high impedance before another turns on, but the output shutting down may be so far away that at the input pin's location, its vestigial voltage is still driving the net briefly when a conflicting signal arrives. A few picoseconds of such overcurrent, if they do not occur too frequently, will not damage any components.

The simulator keeps some track of these overcurrent conditions as they occur. For each input pin, the duration of the longest conflict—being driven simultaneously high and low—is stored. In addition, the duration of the *shortest* non-overcurrent period at each pin is stored. No further details are saved. It is not known how many overcurrent instances there are, when they occur, or anything. But the stored

```

fetched C[12]    a.uuu 3 5 1                                22,891,592 ps
C outputs going lo-Z
***** click 0 at next rise *****                        22,898,665 ps
reading 32951280099 from R[1]                                22,907,106 ps
reading 20365011074 from L[5]
C outputs going hi-Z
fetched C[13]    j.t 0 0 22                                22,953,648 ps
C outputs going lo-Z
L outputs going hi-Z
R outputs going hi-Z
***** click 0 at next rise *****                        22,960,721 ps
STORING 53316291173 to R[3]                                  22,969,162 ps
reading 0 from R[0]
R outputs going lo-Z
C outputs going hi-Z
STORING 53316291173 to L[3]
reading 0 from L[0]
L outputs going lo-Z
fetched C[14]    a.uuu 2 2 4                                23,015,704 ps
C outputs going lo-Z
***** click 0 at next rise *****                        23,022,777 ps
reading 1 from R[4]                                          23,031,218 ps
C outputs going hi-Z
reading 52 from L[2]
fetched C[15]    a.uuu 5 0 1                                23,077,760 ps
C outputs going lo-Z
L outputs going hi-Z
R outputs going hi-Z
***** click 0 at next rise *****                        23,084,833 ps
C outputs going hi-Z                                        23,093,274 ps
STORING 53 to R[2]
reading 32951280099 from R[1]
R outputs going lo-Z
STORING 53 to L[2]
reading 0 from L[0]
L outputs going lo-Z
fetched C[16]    a.uuu 1 0 3                                23,139,816 ps
C outputs going lo-Z
L outputs going hi-Z
R outputs going hi-Z
***** click 0 at next rise *****                        23,146,889 ps
STORING 32951280099 to R[5]                                  23,155,330 ps

```

**Listing 11.8:** Final portion of simulator output with Fibonacci program. (1 of 2)

```

reading 53316291173 from R[3]
R outputs going lo-Z
C outputs going hi-Z
STORING 32951280099 to L[5]
reading 0 from L[0]
L outputs going lo-Z
fetched C[17]    cmp.uu 0 6 2                23,201,872 ps
C outputs going lo-Z
R outputs going hi-Z
L outputs going hi-Z
***** click 0 at next rise *****        23,208,945 ps
STORING 53316291173 to L[1]                  23,217,386 ps
reading 53 from L[6]
L outputs going lo-Z
STORING 53316291173 to R[1]
reading 53 from R[2]
R outputs going lo-Z
C outputs going hi-Z
L outputs going hi-Z                23,263,928 ps
R outputs going hi-Z
fetched C[18]    j.ne 0 0 12
C outputs going lo-Z
***** click 0 at next rise *****        23,271,001 ps
C outputs going hi-Z                    23,279,442 ps
reading 0 from R[0]
R outputs going lo-Z
reading 0 from L[0]
L outputs going lo-Z
fetched C[19]    j 0 0 5                    23,325,984 ps
C outputs going lo-Z
***** click 0 at next rise *****        23,333,057 ps
C outputs going hi-Z                    23,341,498 ps
reading 0 from L[0]
reading 0 from R[0]
fetched C[5]     halt 0 0 0                23,388,040 ps
C outputs going lo-Z
***** click 0 at next rise *****        23,395,113 ps
C outputs going hi-Z                    23,403,554 ps
reading 0 from R[0]
reading 0 from L[0]
quit at 23413040 ps                      23,413,040 ps
see fire.report for fire information

```

**Listing 11.8:** Final portion of simulator output with Fibonacci program. (2 of 2)

information does contain how many picoseconds the longest short circuit was present, an important number to be aware of in order to avoid component damage.

Dividing the longest overcurrent interval by the shortest non-overcurrent interval establishes an upper bound on what the duty cycle of an overcurrent net may have been during a simulation. The long overcurrent may not be close in time to the short non-overcurrent, so the duty cycle may not have been close to the quotient—but it is guaranteed to not have been worse.

Table 11.3 shows saved overcurrent information from `fire.report` for the  $\alpha_4$  RAM, which of all components had the most worrisome figures. I would guess that upon further investigation, the 5.8 ns overcurrent at the d4 pin would be demonstrated to be a harmless one-time startup glitch. Until the minicomputer is ready for physical construction, it would be premature to be concerned about nanosecond-scale overcurrents that show up during simulation. But when the time comes, at least one tool for identifying harmful overcurrent in the design and suppressing it will be available.

### 11.4.5 Hazards, limitations, and next steps of simulation

Revisiting a caution from section 11.4.2, simulations do well at uncovering things that go wrong when they are run. But they are not so good at identifying what *could* go wrong, but did not. If a simulation is so fortunate as to have enough CPU resources, the simulation’s main lapses will come from the system model and the underlying assumptions of that model.

Table 11.3 shows part of a system-wide survey of overcurrent risks, with the affected nets identified by input pins they contain. This is one example of a problematic simulation assumption. *Not all nets contain input pins.* Therefore, there are nets that were not evaluated for overcurrent. An example is the output pins of `ff o`, which move words from the ALU to the I/O subsystem. The I/O side of `ff o` hasn’t been designed yet, so that node has no input pins. Although finishing the minicom-

input pin	longest conflict (ps)	shortest non-conflict (ps)	worst-case conflict fraction
a0	571	15 014	0.038
a1	630	15 014	0.042
a2	648	15 014	0.043
a3	670	15 014	0.045
a4	687	15 014	0.046
a5	703	15 014	0.047
a6	636	15 014	0.042
a7	654	15 014	0.044
a8	723	15 014	0.048
a9	674	15 014	0.045
a10	548	15 014	0.036
a11	560	15 014	0.037
d0	3 500	58 556	0.060
d1	3 425	58 631	0.058
d2	3 400	58 656	0.058
d3	3 500	58 556	0.060
d4	5 796	56 260	0.103
d5	2 513	59 543	0.042
d6	3 500	58 556	0.060
d7	4 063	57 993	0.070
cke#	6	2 555	0.002
e1#	6	46 536	0.000

**Table 11.3:** Overcurrent conflicts at the nets of  $\alpha_4$ 's input pins.

puter will resolve this particular example, similar cases could exist, such as an output that only attaches to a connector. So only considering overcurrent at input pins is a simulation risk at this time.

Another risk is that substituted components will rarely have the same specifications as the originals. The most foreseeable fault would be upgrading a code or data RAM from  $1\text{Mi} \times 36$  to  $2\text{Mi} \times 36$ , which my netlist is designed to accommodate. The  $1\text{Mi}$  RAM has a  $t_{KQ}$  (clock to output valid time) of 5.5 ns, but the  $2\text{Mi}$  RAM's  $t_{KQ}$  is 6.5 ns. How will this be controlled? Somewhat strangely, a further upgrade to



4Mi  $\times$  36 returns to the better  $t_{KQ}$  of 5.5 ns.<sup>5</sup>

Another limitation is that the simulation accepts a manufacturer's maximum propagation delay as its actual value. A related problem is that estimated maximum for circuit board track lengths are presumed to be the actual track lengths. For both cases, minimum values are supplied or computed in the source code, but these values are not presently used. Once the minicomputer is working with the simulator as it now exists, some "fuzzier" runs should be done to ensure it is not too sensitive to small variances.

Clock tree skew, the clock waveform's shape, the choice to use or not use termination resistors, and jitter everywhere may all affect whether the minicomputer will or will not work in real life. Unless the design is changed so as to not critically depend on  $t_{KQX} - t_H$  as it presently does (section 3.4), it is very important that skew and jitter be incorporated into the simulation before investing in circuit board track routing. The waveform issues, on the other hand, cannot be modeled by the discrete event simulation. One piece of hopeful news is that traditional mixed-signal simulators such as Ngspice may be able to answer a lot of clock tree, waveform, and termination questions without needing to include the other 97% of the minicomputer in those models.

Another wavelength that my simulation does not see is radiofrequency interference. Perhaps established design heuristics for power, shielding, conformance testing, bypassing, and filtering can adequately treat the RFI risk. Construction and testing may help with some answers, but the simulation as it presently exists will not help.

---

<sup>5</sup>Contact with the manufacturer established that the 2Mi variance is not a typographic error.

# 12

## Opposing viewpoints

The research of this dissertation, relating to both solder-defined minicomputers and the excursion topic of fast SRAM-derived multipliers, confesses an audacious departure from mainstream computing literature and practice. Regrettably but unsurprisingly, the more original a technically sound proposition is, the more opposition it faces to survive or propagate. This is not a new problem for science, and I have seen careers haunted by it. The maxim attributed to Louis Agassiz is not so remarkable because it may have been said [Eddy1889], but for being so popularly requoted for more than a century:

Every great scientific truth goes through three stages. First, people say it conflicts with the Bible. Next, they say it has been discovered before. Lastly, they say they have always believed it.

Other than what people accept as their canon, nothing has changed. As of November 2022, four leading journals have rejected papers I have written describing this work. Two of the journals reviewed a short paper about SRAM ALUs, and the other two reviewed a paper that is now chapter 10 of this dissertation. Additionally, a prominent organization that exists to support fundamental research has declined to fund a proposal that I was invited to submit after a preliminary screening.

Among the most rigorous obstacles faced by many researchers is the imposition of mandatory page limits for journal and conference papers. I treasure succinctness, but

some discoveries are of such complexity, novelty, and/or utility as to overwhelm the reviewers' available time for many journals and possibly most conferences, and page limits are perceived to be the solution. Of the five times my work was rejected, three had reviewers take a position that I had not included enough supporting material, explanation, or context, notwithstanding their editor's promise to reject the work without review had I offered more.

Even without the pressure page limits place on reviewers to reject papers, reviewers intuitively feel discomfort when technical arguments appear to defy established practices. The legal tradition of *stare decisis* has a brother or sister in computer science somewhere. In my experience this has been more of an issue with journals, where my win-lose record is 0–4, than for conferences, where my record is 4–0, notwithstanding that my four conference topics were similarly unconventional. Unfortunately, the conference option is ill-suited for more than I presented in [Abel21], on account of the anemic length limits advertised by most conferences. Other investigators such as [Ousterhout21] have made similar observations and proposed changes.

For one of the five rejections, what I sent did not reach the reviewers at all, but came back with an X in the out-of-scope box as its only feedback. I should have challenged that gatekeeper, while I had an opportunity to do so privately, with a copy of the scope that was claimed and asked for justification.

Because no architecture is separable from the communities it may serve, the anonymous reviewer comments are an essential part of what is presently known about this new technology. The following are word-for-word excerpts from their feedback. To maintain impartiality in presentation, their order was randomized by running them through `shuf` exactly once without further alteration.

- The writing style and structure of the paper should be significantly improved to meet the standards of an academic publication.
- There is not any experimental results.
- The market opportunity is not clear and the proposal does not discuss potential market size.
- The basic concepts in the paper are well known. Only the scenario of building a processor using SRAM brings some novelty. I am not sure that this paper is of much interest for many readers.
- There is no evidence to support claims such as “an emerging concern for VLSI is that complex ICs may be subject to design defects or backdoors, and measures for inspection and audit of these chips are neither practical nor supported by manufacturers.”
- While it may be counterintuitive to provide motivation for solder-defined architectures in a paper that is fundamentally about multiplication techniques, it is necessary when there are no previous works that provide this motivation instead (e.g., the future paper concept of a “general introduction to or survey of the surrounding security landscape” mentioned in the rebuttal does not exist yet) and this motivation is used to support the sole improvement of this work over traditional circuits (i.e., reliability). For comparison, when in-memory computing was first becoming popular several years ago, a significant portion of any work in the field was devoted to motivation; nowadays, after the motivation for in-memory computing has been clearly established in many previous works, it is possible to publish algorithmic works (e.g., on in-memory multiplication) without the need to extensively explain the motivation of in-memory computing by simply providing citations instead. Therefore, my recommendation is to either strengthen the motivation section of this work to also include the validity

of solder-defined architectures, or to release that motivation separately (e.g., in an arXiv version of the aforementioned future paper concept) and then to reference it in this work.

- With additional work, this paper may indeed be on-track to becoming a strong academic publication.
- This project is about developing a new computer architecture that is simpler and more secure. However, it is not clear that it will be powerful enough to support a wide range of applications.
- Is the paper technically sound? No.
- I can understand that this paper is the first to propose an inexpensive (fitting within one circuit board) means of building multipliers without reliance on ASICs, microprocessors, FPGAs, or PLDs. However, I still think the author could compare with the related work, such as discrete component multipliers and VLSI multipliers, to better highlight the proposed work. Exact comparisons can be difficult and allow for some estimation.
- No IP has been filed yet to protect the sustainable competitive advantage.
- Whereas the concept of building a microprocessor using SRAM is interesting, I do not think that the implementation of individual components deserve to make it into a journal paper.
- The revised paper presents interesting and novel techniques for constructing multipliers from SRAM arrays, yet remains insufficient in establishing motivation and results.
- The paper lacks comparison with other related works. Could the author provide a table that shows the comparison with other works?

- The contribution of this paper is interesting but quite modest. It is well known that large products can be computed by combining smaller products. It's also known that some computations can be pre-calculated and stored into addressable memory. Actually, that's how FPGAs work.
- The whole ecosystem needs to be changed which makes adoption very difficult.
- The structure of this paper needs some improvement. This paper focuses on the description of the proposed design but lacks test or simulation results.
- Under the presumption that solder-defined architectures are well-motivated, there still remains several aspects of the evaluation that can be strengthened. Essentially, the proposed work could be compared to several other approaches to reliable multiplication (see below). Ideally, "reliability" could be quantified for a numeric comparison between all of the approaches; as this is likely not possible, then qualitative explanations should be provided at the very least.
- There are many statements in his manuscript that can be shortened.
- The author proposes carry-save and carry-skip two techniques. Could the authors discuss how much performance can be improved with these two techniques?
- In Section III C: "The cost to accommodate mixed signage is negligible, but many CPUs overlook it." I think many CPUs overlook it because it is not needed. In a multiplication process, the system is either unsigned or signed, and mixed signage (one has sign and another is unsigned) is not required since it is not necessary for the applications.
- The percentage of improvement in terms of latency, energy can be added in the abstract.

- It was not clear how users can write/migrate programs to execute on this new architecture.
- Building hierarchical parallel multipliers is not new.
- The most significant problem with this work is the lack of motivation on the need for an “open-source multiprocessor.” The motivation is discussed in Section I, yet there are barely any citations (with citation [2] also being unspecified) and the claims are not reassuring.
- The technical discussion only includes a map of how a general CPU may be architected. A lot of key details are missing.
- [X] Submission is deemed outside of [acronym]’s topical scope.
- The exact technical discussion is not clear. Various components were omitted from Figure 1 to illustrate the whole process.
- The problem of trusting a computing platform is well known, and I propose as an example that Intel-designed 80286 processors are still in use in most civil aircraft as the cost of certifying a more modern microprocessor is too high. This is in my opinion the solution to the problem put forth by the author: if we cannot trust a modern microprocessor, it would make more sense to rely on a simpler design manufactured using old processes. Can the author refute this argument?
- It appears that the performance of such a “minicomputer” will be drastically worse than existing processors. Therefore, the author needs to clearly establish two points: (1) the flaws in existing processors, and (2) why the proposed SRAM-based architecture is not subject to these flaws (why can’t manufacturers conceal “backdoors” in SRAM?).

- The feasibility of ALU component of the idea has only been shown in simulations. No prototype has been built despite prototype is easy to build and is based on inexpensive electronic components.
- Market penetration is very difficult in this domain. It was not clear who the customers.
- The evaluation of the proposed multiplier is highly lacking, and should include a comparison to alternative solutions. For example, how would the proposed multiplier compare to a VLSI circuit that consists of a single multiplier? Wouldn't an isolated VLSI circuit that only performs multiplication not have "backdoors" since it is not complex enough?
- I agree with the author that asking to provide new references in a topic that went out of fashion decades ago is really unfair.
- The main contributions of this paper are: how the author deals with mixed signage, and compressing the partial products. Whereas this is interesting on its own, I do not think it has the required merit for a journal paper.
- This project tackles an important problem, however the proposed approach lacks details about the exact innovation, its business model and adoption.
- Does the paper contribute to the body of knowledge? No.
- Since the author is proposing a "novel computation with memory," the author should discuss that in his manuscript and compare and contrast it with the famous "computation in memory" by citing the two suggested references and other related references with the latest publication dates.
- The author's motivation for using SRAM as a multiplier is not very clear.
- Comparison of the proposed method with state-of-art is missing. Authors may



compare their work with the existing works in terms of latency and energy.

- The references cited clearly explain the author's justification. However, most of them are old and from the past years. Could the author discuss more up to date papers which are comparable to his claim?
- Since the validity of solder-defined architectures has not yet been clearly established, then it is difficult to overlook the drastic performance decrease in multiplication throughput compared to existing architectures. The difference between the 10 million multiplications per second provided in this paper and the 10+ trillion multiplications per second present in modern GPUs/TPUs/etc.. is over \*six\* orders of magnitude apart; this drastic gap is only countered by the unsupported assertion that solder-defined architectures are more reliable than traditional hardware.
- Are the references provided applicable and sufficient? No.
- Section III does not present any new information or ideas interesting for readers of computer arithmetic.
- The hardware components are very simplistic. For example, the ALU is too simple and inefficient, and the CPU does not provide support for powerful and flexible program constructs, e.g., stack, cache, pointer to function, etc. While they can be more secure, they can support very limited set of use cases.
- The claim that computers can be made in the suggested way is difficult to defend because of the large cost, complexity, power consumption, and added likelihood of hardware failure as the number of discrete component increases.

# 13

## Findings, motivation, significance

### 13.1 Major findings to date

Chapter 1, Overview, describes VLSI irregularities that can lead to exploitable defects, and categorizes the irregularities according to how they intersect with designer intent. The chapter notes that maliciously introduced defects in microprocessors, FPGAs, PLDs, ASICs, and other complex logic may not be avoidable without forgoing all use of these components. Due to differences in agenda, there may be disagreements between manufacturers and buyers as to whether a specific behavior is a benefit or a defect. Lists of criteria for acceptable computers, CPUs, and their supply chains are proposed, and a project is inaugurated to construct a computer that satisfies the lists without the use of any microprocessor, FPGA, PLD, or ASIC. An understanding is established that although such a computer would only be practical for certain applications, there are enough use opportunities to justify this research.

Chapter 2, Definitions, does not have any narrative, but strongly implies a need for new terminology for computers that do not contain microprocessors, FPGAs, PLDs, ASICs, etc. New terms are proposed in response to this need such as *complex logic*, *discounted logic*, *internal firewall*, *maker-scale assembly tools*, *minicomputer*, *solder-defined behavior*, *solder-defined hardware*, and *supply-chain firewall*.

Chapter 3, Components, establishes that any practical general-purpose digital

computer will have to employ VLSI RAM. A case is presented that VLSI SRAM ICs are unlikely to contain exploitable defects today, and it will not be practical to maliciously introduce and exploit such defects in the near term. After considering alternative logic families, a proposal is advanced that the best option for building solder-defined minicomputers would be to construct all logic solely from VLSI SRAM and trivial glue logic ICs. The chapter shows that only a limited selection of fast glue logic components is available, and suggests circuits for needs that present commercial ICs do not directly support.

Chapter 4, Logic blocks for SRAM ALUs, examines the effect of using SRAM in logic design, and introduces new circuits for addition, rearrangement of subwords (swizzling), rotation, shifting, and fast multiplication. (Simple lookup tables and substitution-permutation networks based on SRAM are also described, but they are not new.)

Chapter 5, Three-layer ALU structure, shows the existence of a natural superposition for SRAM carry-skip adders, swizzlers, logarithmic shifts, and substitution-permutation networks that can be used to construct robust SRAM ALUs. The nature of this superposition and capacities of prevalent SRAM ICs suggest strongly that the register width of an SRAM minicomputer should be 36 bits.

Chapter 6, Two-layer ALU structure, describes small SRAM ALUs for controllers and other devices that can get by with less speed and/or a smaller word size.

Chapter 7, A three-layer, 36-bit ALU firmware, demonstrates that the set of opcodes implemented by a three-layer SRAM ALU can be highly robust, flexible, and competitive with other logic families. An open-source reference firmware has been released to encourage the use of already-proven features, performance, usefulness, and security in future implementations.

Chapter 8, A solder-defined CPU with protected memory, describes a robust SRAM CPU and protected memory subsystem that fits on a circuit board smaller than the dimensions of this page. Most functionality has been confirmed in electrical

simulations to work, with the remaining portions anticipated to be work by the time you read this. A worst-scenario CPU speed of 16 MIPS appears to be easily reachable, and 20 MIPS has not been ruled out as of November 2022.

Chapter 9, Forthcoming subsystems, argues that the missing subsystems of the chapter 8 minicomputer—preemptive multitasking, firmware loading, and I/O—will also be constructable using solder-defined logic and be useful in their implementations. General descriptions are given of the anticipated designs for these subsystems.

Chapter 10, Fast parallel multipliers, introduces a new general theory for SRAM multipliers, their sections, and their performance. This knowledge is new, because SRAM multipliers differ architecturally from multipliers built from basic gates. SRAM designs are aided by offset-binary representations of signed subproduct terms, by generalizing carry-skip addition to accept more than two addends, and by integrating carry-skip addition of subproducts simultaneously with carry-save addition. Arbitrarily large factors can be multiplied quickly using hierarchical carry-skip methods, although component counts will be correspondingly high. Tables of component and clock cycle counts for SRAM multipliers of anticipated word sizes are presented. Manual design of SRAM multipliers is exceedingly tedious; therefore, an open-source tool has been released for generating and verifying efficient designs.

Chapter 11, Minicomputer implementation, presents the publicly released implementation of the minicomputer thus far [Abel22b]. The dataset includes the firmware for the ALU and other portions of the CPU, assembler for writing programs, virtual machine for testing programs, netlist for the components and their interconnections, software for processing the netlist, and discrete event simulation software that can verify connections, wired logic, firmware, assembler programs, and signal timing as the minicomputer is emulated. The completeness and transparency of the implementation dataset and its documentation throughout this dissertation, in conjunction with the Creative Commons Attribution 4.0 International License they are available under, establish that the architecture of this dissertation is fully open.

Chapter 12, Opposing viewpoints, explores some of the skepticism, reluctance, and other barriers the proposed architecture will encounter along its path to deployment, standardization, and acceptance.

## 13.2 Motivation for this work

Although I enjoy designing computer architectures, I do not consider this to be my native field of research. When I came to Wright State University, I had a thought to focus on communication privacy, and I had a special interest in communication metadata privacy, because *who* and *when* one associates with is at least as revealing as what is being said between the parties. Metadata privacy touches on the foundation of republics in civil rights that are supported by the rule of law, the stability of economies around the world, and the intimacy that should be honored—but is sickeningly violated—between close family or friends who are separated geographically. So as a research computer scientist, it has been my mission to invent technologies for defending civil rights in the face of dystopian threats.

From early years, I began to read James Bamford’s series of extremely dark nonfiction books [Bamford82, Bamford01, Bamford04, Bamford08]. They purport to be about the NSA, but I view them more as a cautionary tale of the opportunities for calamity that come when any eavesdropper is given unlimited computing power, unlimited network access, unlimited funding, or unlimited legal authority.

As an example of the power of metadata, consider the possibility that Bamford wrote a *fifth* book in his series, a book that was never published and that goes unmentioned online. We would have no knowledge as to the contents of this book, but metadata *about* that book may indicate the book’s title or general subject, as well as identify some of the persons involved in its writing, reviewing, and attempted publication. Indeed, this is exactly what happened in 1995, when [Bamford95] was either rejected, withdrawn, or suppressed from publication.

History books such as Bamford’s, technical books such as [Schneier96], a rule-making fiasco known as Clipper, and other events in the news made me a cryptography skeptic while I was still in my twenties. This was also a period when the cost to store data imploded to nearly zero, suggesting to me that traditional cryptography could be supplanted by one-time pads for certain uses, even as all “respectable” literature seemed to advise against them. When I arrived at Wright State the same semester as Junjie Zhang in the fall of 2012, he instructed his Computer and Network Security pupils to do a project and write a “conference-style paper.” My paper was about mechanisms for on-demand distribution of one-time pads within a large user community. By having each user share a large one-time pad with a central hub, the hub could be requested to consume that one-time pad in order to distribute new one-time pads to user pairs as they need to communicate.

Because this was a class assignment and I had just started in the Ph.D. program, my paper’s style and tone (and perhaps contribution) weren’t ready to submit to a real conference, and the paper was a little long. But I defensively posted the work to arXiv so that no one else could set progress back twenty years by patenting the mechanism [Abel12]. And in this paper, I wrote ten columns about the myriad challenges of securing the machines at the endpoints of one-time pad links—because such endpoints are the sole location where a break in the encryption itself can happen. Most of these challenges were a direct outcome of the “enlightened” features we teach in operating system or computer architecture courses, features that leave key material in places seldom thought of. But some of the challenges were in the hardware itself, and I cited the critical role played by the CPU and other VLSI in handling sensitive information.

On July 4, 2014, I published Clique, a network protocol for metadata privacy for short messages, with a reference implementation written in C (and manpages written in `troff`) [Abel14a]. Clique works by arranging all protocol users into a global (as in planetwide) connected graph, where everyone sends UDP packets to everyone on

a fixed schedule, irrespective of who knows who to begin with. The protocol requires that all packets be encrypted so as to be indistinguishable from independent and identically distributed octet sequences. Every listener attempts to decrypt every packet received with every key the listener has on hand, and those packets that decrypt to plaintext are the ones where actual communication is happening. From the perspective of an eavesdropper with unlimited network access, Clique works by flipping the problem of determining who is *talking* to who to the much harder problem of determining who is *listening* to who.

Here again with Clique, I found that the design of the endpoint hardware itself was a critical factor if Clique were to be effective. In the platform notes of Clique’s manpage, I noted:

The Clique protocol is only as secure as its communicating endpoints, inclusive of hardware and operating system flaws. In today’s environment of faithless routers, CPU “management engines” that cannot be disabled, wide-open memory mapped I/O, automatic updates for software (whether closed-source or open-), and an abundance of money for bribes, endpoint security is about as tight as the border between Missouri and Kansas.

My first conference paper relating to the subject of this dissertation also appeared in 2014. It related to a central problem of Clique, which was bandwidth. When setting up a Clique endpoint, the owner must decide how much bandwidth to permanently allocate to Clique. For instance, perhaps 57 600 bits per second out of an inexpensive broadband connection can be afforded for Clique traffic. If the global clique has 10 000 users, then the 57 600 bits will have to be distributed to 9 999 other endpoints, making the throughput to any endpoint 5.7606 bits per second. Clique allows each endpoint to decide (and stick with) its own packet size, but the documentation recommends everyone choose 576 octets so that groups of similarly-configured endpoints do not stand out. The endpoint will have enough bandwidth to send one

packet to each Clique host every 800 seconds, or  $4\frac{1}{2}$  packets per hour. After protocol and cryptographic overhead, each packet can hold 532 octets of chitchat.

So my conference paper [Abel14b] looked at the following question. Compression algorithms that appeared over the years were increasingly well optimized for ever-larger files to compress: **gzip** was good for small stuff, **bzip2** for larger, **xz** for immense. How can the most compression be gotten for 532 octets? Because a single byte more would delay a transmission more than 13 minutes. And if the clique had 1 000 000 users instead of 10 000, the added delay for one byte would exceed 22 hours. So every byte counts with Clique, and my paper describes a preset dictionary scheme that could pack dozens of bytes beyond what plain **gzip** could do. (**bzip2** and **xz** have too much overhead for short datagrams.) But I knew before publication that the whole premise of [Abel14b] depended on hardware security, and that without that, my proposals to improve metadata privacy were not going to amount to much.

My second conference paper that motivated this dissertation's research appeared a year later [Abel15]. This paper was about the metadata privacy of one-time pads in the presence of jamming. I wrote:

OTPs also have non-intuitive bottlenecks in CPU, RAM, disk I/O, and key material consumption, particularly when OTPs protect their own communication metadata (such as the volume of information exchanged) in the presence of random packet injection attacks, link failures, and endpoint outages. This paper explains how these problems arise and presents a set of countermeasures that treats them effectively and scalably.

In that paper also, I was very conscious of the risk hardware security posed to all that I was discovering, and I projected:

As correctly implemented one-time pads have no other vulnerabilities, the communication endpoint becomes the lightning rod for all attack scenarios. We expect that before long, most one-time pad research will be



centered around endpoint security

After I founded Wakefield Cybersecurity in 2018, I traveled with and recorded a talk where I explain what I believe the central problems to be with that day's leading cybersecurity practices [Abel18]. And at the end of my talk, I brought up hardware security yet again, and I talked a little about an idea I had mulled for several years. What if building CPUs could skip the VLSI foundry entirely? I argued:

One technology we could use right now at medium cost, say \$2 000 to \$20 000 per CPU, is to assemble processors directly out of surface-mount transistors, placing and soldering them onto custom boards with relatively inexpensive robots. The traces on the boards are large enough to audit, and you can make a 32-bit processor like the original ARM CPU out of just 25 000 transistors. You might think that's a lot of transistors, but they're a lot only in number. Some of today's surface-mount transistors are so small that you can scoop 25 000 of them up in a single teaspoon. Because of low semiconductor prices and the automation we have available, it's less expensive today to make computers out of individual transistors than at any time in history.

From that time, my thought kept coming back to becoming the architect of just such a machine. Along the way, I read all I could about the experiences others had building their own CPUs without microprocessors. The compilation in [Toomey17] is exciting, not just because of the retro-ness and spunk of the hobbyists, but also informative as to their mistakes, tendency to sometimes think too much alike, and lack of use potential beyond entertainment and personal accomplishment.

I couldn't help but notice the lovely relay computers that rose out of untold hours of solo effort and meticulous workmanship, computers with very few registers and no appreciable primary storage at all. Their only salvation would be SRAM ICs—forbidden, hypocritical components that would undermine the very purpose of

these modern museum pieces. I also hung around with the retro computers I had purchased before they were retro, including my HP-41CX, my TRS-80 Model 100, and I added two sorobans and a Tandy 102. I bought some chips and expanded my Model 100, which I bought in 1985, from 8 Kibyte to 32 Kibyte. But I felt a little regret afterward, like I had somehow turned my back on my past and the truth of what it actually was. And as I played anew with my toy from three decades before, feeling how wonderful its keyboard mechanism worked and thinking about what I could use it for today, it came to me that available RAM was going to be a limiting factor. And although there are kits for attaching large storage to that computer now, their use would further separate me from what I once had. And I thought again of the new computer I might build with those 30 000 transistors. It would not be enough.

It would have to have static RAM ICs.

Around 2019, I started taking a fresh look at what components I could and couldn't use in designing an open, secure computer architecture that did not require a semiconductor foundry, or a lot of trust in one. This is about where chapter 3 continues my story, and I started puzzling over my project in roughly the order of the remaining chapters.

I soon had a first draft of my dissertation topic proposal, but not an advisor yet in our department. To make sure the its contributions would pass into the public domain instead of a patent troll office, I posted its 128 pages on the Wakefield Cybersecurity website on April 11, 2020.

A better repository for preservation would have been arXiv again. Over-moderation had already kept me from publishing my Clique preprint, which has never been made public. Their requirement to find a new recommender every single time one of *their* subject boundaries gets crossed proved to be too much red tape. Another defensive publication, a short white paper about SRAM multipliers, also didn't make it through arXiv moderation and was never posted elsewhere. The moderator thought my write-up was too succinct for academia. So the Wakefield website served as my

personal *IBM Technical Disclosure Bulletin* from the first topic proposal draft until I discovered Harvard Dataverse. In the meantime, arXiv continues to miss out on preprints in every discipline, even while it facilitates amazing progress in these exact same disciplines [Garisto22, vandenHuevel15].

I introduced myself to Travis Doom by email during 2020, and he kindly accepted one of the stray ghosts of our department under his wing. After revising and defending my topic proposal, I got back to work. My next conference paper, and last successful peer-reviewed publication to date, appeared in [Abel21]. It was a reincarnation and extension of the arithmetic logic unit that two journals rejected (chapter 12 has that story), and it now had the entire minicomputer as its topic.

In 2022, I uploaded the implementation for my architecture as it exists thus far into Harvard Dataverse [Abel22b], under the terms of a liberal open-source license. I believe this is where the serious peer review process begins as it applies to my work. For open software and open hardware, the relevant literature is the body of code that is shared, downloaded, contributed to, deployed, and forked. This is a mixed blessing. On one hand, the community does not wait for conferences and journals to approve or curate their efforts. On the other hand, its platform for disseminating new information, if such a platform exists at all, is very decentralized. There are a lot of routes to a project becoming known and finding establishment, but there is no instruction manual for seeking either. Also in 2022, I uploaded software for automating the design of parallel SRAM multipliers for arbitrary combinations of word size and signage, also in Harvard Dataverse [Abel22a].

## 13.3 Security advantages of the architecture

### Open hardware and open firmware for running open software

I have read many forum posts where users complain that some portion of a controller's firmware is closed-source, inaccessible to the owner, and/or encrypted. All firmware

connected with this dissertation is completely 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 using a transparent algorithm. The minicomputer owner has absolute and final authority over every *bit* of the firmware, and the purpose of every 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 the computer.

The hardware is visually and electrically inspectable after manufacture and purchase. The firmware is easily accessible, using appropriate tools, on the serial flash memory for verification and change management.

### **Security perimeter for solder-defined logic**

A security perimeter surrounds the CPU, memory subsystem, firmware loader, and I/O subsystem, inside which there are no purchased complex logic such as microprocessors, FPGAs, PLDs, or ASICs. The only points where this perimeter is crossed are the individual serial buses between the I/O controller and peripherals, with no serial bus attaching to more than one peripheral. The boundaries at these serial buses form an internal firewall, such that any peripheral's defects that may 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 compu-

tational cost on account of the MIX and XIM opcodes. Encryption keys would be installed by the firmware loader from the serial flash memory IC. These keys need only be kept secret from the peripherals they defend against.

## 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 8.6 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 ff a, ff t, and ff c.

In the presence of well-behaved firmware, the only access to stack memory is via the CALL, RETURN, and privileged CALI instructions. No privilege escalation can result from stack overflow, and there is no possibility for stack underflow by programs that the program loader initializes correctly.

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 employ 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 (meaning per-peripheral) basis.

Although memory hygiene for *software* is a hot topic today [NSA22], my archi-

itecture offers nothing new in this domain. I generally think that memory hygiene within a program is a long-solved problem for well-written applications.

## **Control of the CPU**

All nonprivileged CPU 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.

## **Arithmetic**

The ALU 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.

## **Why tamper resistance is out of scope**

Some vendors and some cybersecurity practitioners will attack this architecture on the pretext 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 much tangible personal property 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 perhaps not the top threat of 2022. More likely, the probable adversaries will be the equipment manufacturers or part suppliers themselves, governments of the jurisdictions where the equipment is used, or international criminals.

Buyers who require tamper resistance for this architecture 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 infringes on the buyer's rights.

## 13.4 Drawbacks of the architecture

The architecture of this dissertation is compatible with nothing that exists on the planet. This is by design—one could say there is something rather *backward* about backward compatibility. But freedom from past baggage comes at the cost of needing new. New operating systems, new toolchains, new software, new training, and new documentation for all of it. No one familiar with this architecture is available to hire for help.

Use cases and ease of deployment will be constrained by the size of available SRAM ICs. A fully-loaded circuit board for the present netlist offers 4 Mi and 8 Mi words of code and data memory respectively, equivalent to 54 Mibytes. Should the architecture drive enough demand for larger SRAM ICs, foundries have the technical ability to increase SRAM density and the financial margins to lower SRAM prices.

The peculiar security restrictions of the architecture, such as no data storage on the stack, no pointers to code memory, new features for arithmetic overflow handling, and small memory sizes make the architecture unlikely to ever run Linux, GCC, Clang,

or other prominent megaprograms. This is not entirely bad, but any functionality that developers or system administrators need will have to be created. I hope that this time around, consideration will be given to the size, reliability, longevity, and auditability of the new software and libraries.

In addition to the drawback of limited memory, CPU speeds in excess of 20 MIPS are not on the near-term horizon for this architecture, and speeds for early systems could be as low as 16 MIPS. Peripheral support will be limited to SPI and I<sup>2</sup>C bus devices indefinitely, and I/O bandwidth will be on the low side. (I/O bandwidth may be of small concern, considering there is not much memory in the machine to transfer in the first place.)

The physical size, cost per unit, and energy consumption of this minicomputer will be larger than many alternatives.

I have saved what I regard to be the greatest drawback of solder-defined architectures to note last. The ecological footprint of these devices is comparatively terrible, whether one is considering carbon added to the oceans, conflict minerals, power to operate, hazardous waste, foundry diversion to produce more SRAM ICs, freight, reflow ovens and tools purchased to build a single computer, or other measures of consumption. There may be balancing ecological benefits to consider such as long product life, lack of imposed obsolescence, and losses averted by securing vital assets using this architecture. Might a nuclear reactor meltdown be prevented? Or even a war? I am not an actuary and cannot ethically make any claims, other than it takes more resources to build a solder-defined minicomputer than to build a traditional computer that—while not under attack—can do similar tasks.

## 13.5 Significance of this work

The security problems with today's silicon derive from root causes external to the hardware itself. We can talk about speculative execution, pipelines, cache misses, mal-



formed input, unstable memory schemes, corrupt stacks, arithmetic overflow, dopant-level trojans, closed-source microcode, open system buses, proprietary firmware, counterfeit accessories, and privilege escalation. These technical opportunities superpose with the human circumstances of globalization, divergent interests, addiction to power, situational ethics, and economic inequality. But I believe the root problem with our CPUs is that they are designed, assembled, and distributed without input from the community that uses them. If we hope to find solutions for our security concerns, a thoughtful look in a mirror may be a promising start.

A parallel malady has long existed in the software supply chain: software companies have been writing software for the purpose of selling it, rather than out of their own need to have or use their wares. The symptoms are all too familiar: needless changes and revisions made to already deployed software, premature end of support for widely used systems, easily foreseen vulnerabilities, cloud-based licensing, and subscription-only availability. In many domains, the open-source software movement has afforded relief for some users, although the scale of available software is overwhelming the maintainers who volunteer their time. For closed-source software, much still needs to improve. I believe that on the whole, community-sourced software has brought unprecedented freedom, security, and stability to a world of users, although news reports have proven that this freedom, like any, isn't free [Nichols22]. I believe that a similar approach can be taken with computing machinery itself, offering stability, security, and freedom instead of obsolescence, malfeasance, and monopolism.

This research measured the ability, with presently available components and maker-scale assembly tools, to construct computers that are free of purchased complex logic (microprocessors, FPGAs, PLDs, ASICs), proprietary hardware, closed-source or otherwise inaccessible firmware, and manufacturer aftermarket interference. I found a strong likelihood that a 36-bit, 16 MIPS minicomputer with protected memory, preemptive multitasking, and securely isolated I/O devices on SPI and I<sup>2</sup>C buses can be built.

The components needed are reasonably fungible, although the highest CPU speeds are only attainable through one manufacturer's glue logic ICs. Manufacturer substitution for SRAM ICs is permissible, although the IC carriers would change and necessitate changes near their circuit board footprints. All components are available for the industrial temperature range of  $-40$  to  $+85$  °C.

The minicomputer can fit on one printed circuit board. Once a bare board has been plotted and pressed, assembly is a straightforward soldering task with a reflow oven or hot air tool. All components have leads and need only be mounted on one side of the board. The assembled computer can be visually inspected and checked for pin-to-pin connectivity at any time it is not in use.

For minicomputers that are protected from physical intrusions, the hardware and firmware are intended to be free of security defects. Demonstrating this is true using formal proofs would be an ambitious, constructive, and interesting task. The useful lifetime of the machine is hoped to be at least the data retention of the firmware's serial flash memory, which is specified to be 20 years for the component presently under consideration. The retention can be extended to 20 more years, or if necessary restored, by rewriting the memory. Attached peripherals that are not solder-defined would have their own service requirements and longevity specifications that should not be ignored.

The cost to use an assembly facility will not exclude many builders. Some new reflow ovens cost less than \$300, and yet-smaller-scale soldering methods exist. Even so, automated assembly lines with excellent controls can be hired at reasonable cost and less aggravation than donated time using the cheapest tools.

An eighth-grader with a web browser and a debit card can build a minicomputer. This is a great shift from historic norms, when computers were assembled using wire wrap on boards plugged into backplanes in cabinets on raised floors. Although designs shrank into fewer and fewer discrete packages to reduce assembly costs, assembly costs also ultimately collapsed. The least expensive time in history to build computers “the

hard way”—assembled from separate packages outside of a semiconductor plant—is today and tomorrow.

## 13.6 Future work and timeframe for availability

As I write in mid-November, I expect all features through chapter 8 to work correctly in simulation before you read this. Under the assumption I can be funded to help with the architecture full-time after graduation, the missing subsystems for preemptive multitasking, firmware loading, and SPI and I<sup>2</sup>C bus would take perhaps three months in total to specify and confirm by simulation. A fourth month should be allocated to either manage clock skew effectively or rework the control unit, instruction cycle, and decoder firmware to be immune to any foreseeable amount of skew.

These preconditions would allow layout, prototyping, and evaluation of a real circuit board to begin by month five, with a planned goal of demonstrating a working minicomputer before the end of 2023 that any qualified maker can replicate.

Unless some helpers come around, I do not expect to complete any significant software in 2023 that could either run on or aid in writing software to run on the minicomputer. I do expect that the design and simulation tools, including the assembler, would grow to keep pace with the hardware that is being completed and tested. I also believe that the architecture will be interesting enough that once makers can assemble working machines without trial and error, lots of innovation will appear in short time on the software side of the project.

# A

## Assembly language conventions

This appendix is taken from the assembler reference manual and describes some of how the architecture’s assembly language is planned. Most of what is here has been implemented, although some of the window dressing—particularly certain number bases, underscores as group separators, register declaration after use, and help with permutations—is not yet implemented as of October 2022. As the architecture gets nearer to a fully programmable implementation, this documentation will expand to keep up.

### A.1 Source code character set

Assembly language programs are written in 7-bit ASCII. The only supported symbols are 7 (tab), 10 (newline), and 32–126 (space through ~). In the future, the UTF-8 encoding may be permitted. No other encodings will be supported.

New readers may notice unconventional or bewildering uses for certain characters. Table A.1 summarizes some of those uses.

### A.2 Comments

Comments begin with a semicolon and extend to the end of the line. For example:

```
open_secret = 314159          ; ten thousand times pi
```

**Table A.1:** Peculiar uses for various ASCII characters.

Symbol	Use	See section
0–9	digits 0–9; identifier tails	A.3, A.4
a–z	identifiers; digits 10–35	A.4, A.3
A–Z	identifiers; digits 36–61	A.4, A.3
@ \$	digits 62 and 63	A.3
`	numeral base if not 10	A.3
_	digit grouping; identifier tails	A.3, A.4
. ’	identifier tails	A.4
;	single-line comments	A.2
( )	enclosed comments	A.2
< >	type coercion	A.7
-	ranges in permutation notations	A.8
tab	same as space	
newline	end of statement	

Those are called *single-line comments*, because they cannot span multiple lines. Also supported are *inline comments*, which begin and end explicitly with parentheses. An inline comment can span as many lines as desired and can be used to “comment out” a bunch of code, or to provide space within a source file for documentation. No parsing or syntax checking is done inside inline comments, although only the ASCII symbols authorized by section A.1 should be used. For example:

```
x = (Dare I choose zero here?!?) 0
```

Inline comments do not nest. So how does one comment out a block of code or documentation that contains parentheses itself? This is done using multiple consecutive parentheses. Inline comments don’t technically begin with the ( character, but with a series of one or more consecutive ( characters. They end with an equal number of consecutive ) characters. So (( and )) can enclose blocks that do not contain exactly two consecutive right parentheses. Notably but perhaps less usefully, ( and ) can enclose blocks that contain multiple consecutive left or right parentheses, but not isolated ones. In any event, the runs of parentheses that begin and end inline comments can be as long as available memory permits. For example:

```
diameter = 10
(((
    The circumference is exactly pi times the diameter,
    but this program does not use any floating point
    (so pi is going to be 3).
)))
double_diameter = diameter + diameter
circumference = diameter + double_diameter
```

Inline comments can, of course, allow a line of source code to continue after the closing parenthesis (-es). Inline comments can also be placed around newlines in order to provide line continuations. For example:

```
a = b + (added to) c

twelve_hundred = (
) 1200

fixed_point_thirty_six_bit_approximation_of_pi_divided_by_four (
) = 110010_010000_111111_011010_101000_100010'b
```

All comments, whether single-line or inline, behave as spaces in terms of the assembly language syntax. So identifiers, numbers, symbols, etc. cannot be spliced together through inline comments.

## A.3 Numbers

A *number* is an integer numeric constant for the assembler. The basic format is a single optional + or -, any number of optional leading zeros, a string of *digits* in radix 10 or some indicated radix between 1 and 64, and an optional radix specification.

Digits in radices up to 10 are the familiar 0 through 9. Radices between 11 and 64 use the lowercase letters, uppercase letters, and finally @ and \$ to fill the necessary digits through 63. To preclude surprises, case is always sensitive for digits, implying that a–f are valid hexadecimal digits, but A–F are not.

The symbol ‘ is called a *backtick* and is used to introduce a radix other than 10. It’s ASCII character 96. Following the backtick is either a decimal integer in the range 1–64 (with any number of optional leading zeros), or one of the letters *u*, *b*, *o*, *d*, *h*, or *t* indicating bases 1, 2, 8, 10, 16, or 64 respectively. These letters stand for *unary*, *binary*, *octal*, *decimal*, *hexadecimal*, and *tetrasexagesimal* or *tribble*. For consistency, there is no *x* option to mean hexadecimal: that word doesn’t start with *x*. Note that radix one is a unary number system, with all places having value one.

Here are a few ways of writing 19 other than the usual:

```
111011110011111000111111'b    0010011'b    +201'3    19'10    13'h
```

To aid legibility of long numbers, the string of digits prior to the backtick may contain any number of \_ (underscores) in any position as grouping symbols. For example, the largest 36-bit unsigned number may be written 68\_719\_476\_735 instead of 68719476735 to make it clear the magnitude is about 70 billion. Here are some decimal numbers that are important to the assembler with their base 64 representations:

-34_359_738_368	-w00000't	most negative 36-bit number
0	000000't	zero
34_359_738_367	v\$\$\$\$\$'t	most positive signed 36-bit number
68_719_476_735	\$\$\$\$\$\$'t	most positive unsigned 36-bit number

The base-36 digits are also used in permutation notations, along with hyphens to abbreviate ranges. See also section A.8.

Every number in this book, as well as every number in assembly programs, is in base 10 unless expressly stated otherwise. For convenience, a table of digit symbols appears as table A.2.

**Table A.2:** Digits for bases up to 64.

0	0	8	08	g	16	o	24	w	32	E	40	M	48	U	56
1	1	9	09	h	17	p	25	x	33	F	41	N	49	V	57
2	2	a	10	i	18	q	26	y	34	G	42	O	50	W	58
3	3	b	11	j	19	r	27	z	35	H	43	P	51	X	59
4	4	c	12	k	20	s	28	A	36	I	44	Q	52	Y	60
5	5	d	13	l	21	t	29	B	37	J	45	R	53	Z	61
6	6	e	14	m	22	u	30	C	38	K	46	S	54	@	62
7	7	f	15	n	23	v	31	D	39	L	47	T	55	\$	63

## A.4 Identifiers

Identifiers are “words” used to refer to either registers or branch destinations. They have no limit as to number of characters. They work much like variables and labels do in other programming languages, except in the case of variables, they are always *register* variables. Identifiers are case-sensitive, and the first character—the identifier *head*—must be a letter. The remaining zero or more characters is the *tail* and may contain letters, digits, underscores (ASCII 95), single quotes (39), and periods (46).

The rationale for allowing ‘ is that often in code, some quantity or location is expressed in terms of two sets of units or coordinate systems. The double quote (ASCII 34) is reserved for other uses, so if you want two quotes in a register name, use two single quotes. Underscores are permitted because they are traditional; however, they cannot be the first character, or there would be ambiguity as to whether `_1` refers to a register name or the number one with a strange grouping indicator. Periods are allowed in identifiers as a lightweight means of denoting composite “objects” with more than one member. In actuality, they are distinct variables.

Here are some valid identifiers:

```
j    pt.x    pt.y    F    F'    F''    e.2.71828    another_example    r(adius)
```

The last example is a trick. The identifier is actually `r`, and the text `(adius)` is an inline comment that the assembler disregards. This technique allows the programmer to clarify what is meant when using a short name for convenience. Here’s a longer example, where two variables are conferred advantages of both short and long names:

```
unsigned r(adius) d(iameter)
r = 10
d = r + r
```

## A.5 Abbreviating keywords

A few frequently-used longer assembler keywords have been given abbreviations that can be used as alternatives. All abbreviations end with a mandatory period. The period is part of the token instead of a separator: whitespace is mandatory immediately after the period, but is illegal immediately before it. Table A.3 provides a list of these abbreviations.

**Table A.3:** Keyword abbreviations.

---

c.	call
j.	jump
r.	return
s.	signed
u.	unsigned

---

## A.6 Declaring registers

Every running program is allocated 512 general-purpose registers, allowing a lot of computing without needing to save or load registers to or from primary storage. All registers have the same capability, so there is no need or provision to address them by register number via the assembler. There is also no indirect register access; for instance, a loop can't be written to go through all of the registers.<sup>1</sup>

To allocate a register for a program, it has to be declared with a name and given a default assumption as to whether arithmetic operations should treat it an unsigned or signed quantity. If the register is not intended to store a quantity, it is suggested to specify it as unsigned. To declare registers, simply use the **signed** or **unsigned** keyword anywhere within the code where the register is used. It is not necessary for a register to be declared “prior to use,” so long as the assembler can find the declaration somewhere. The form is simply the signedness with as many names as desired. Do not use any commas. Examples:

---

<sup>1</sup>Such a loop can be written using self-modifying code, which is only possible for privileged programs.



```

unsigned five ten
signed           ; any # of names, including zero, is allowed

five = 5
ten = 10
fifteen = five + ten
twenty = ten + ten    ; ERROR: register "twenty" is not declared

signed fifteen       ; okay to declare after use

```

A register's *signedness* refers to whether it is unsigned or signed. So in the example above, the signedness of `five` is said to be unsigned, and the signedness of `fifteen` is signed. Don't feel locked in by a particular signedness declaration. Signedness actually applies to the operations being performed, not the physical register itself. Section A.7 shows how to override a register's declared signedness for a specific need.

## A.7 Overriding register signedness

A register's signedness affects the range of numbers it can represent without information loss. Through awareness of the signedness of their source and destination registers, operations such as addition and arithmetic shift are able to set error flags in the event of an out-of-range result.

Sometimes the signedness of a register that has been declared with the `signed` or `unsigned` keyword may not represent the programmer's intent under an exceptional circumstance. To overcome this, `<signed>` and `<unsigned>` casts are provided. These casts don't actually affect registers, but instead alter the register type information provided to operations like subtraction or assignment. For instance:

```

signed s
unsigned u z

z = 0           ; constant zero
s = -5          ; no problem
u = z - s       ; no problem; u is now 5
u = s - z       ; underflow; sets the T and R flags
u = z + s       ; underflow; sets the T and R flags
u = <unsigned> s ; no problem; u is now 2**36 - 5
<signed> u = s   ; no problem; u is now 2**36 - 5
s = u           ; overflow; sets the T and R flags

```

## A.8 Permutation notations

Permutations of tribbles are notated in a 6-digit shorthand. Each digit shows the original position of each bit, numbered from the left starting with zero, in the per-

muted outcome. Thus if the permutation 234501 is applied to the tribble 010011'b, the result is 001101'b.

Permutations of 36-bit words use the same notation. All 10 numerals and 26 lowercase letters are necessary. In assembler source code, the few places where 36-bit permutations appear are written without spaces, quotation marks, or the '36 suffix. Underscores are allowed in any position for legibility. Hyphens provide a shorthand for contiguous ascending or descending ranges of bit positions. Here are some examples:

```
unsigned in transpose left_rev right_rev full_rev
; ...
transpose = in perm 06ciou_17djpv_28ekqw_39flrx_4agmsy_5bhntz
left_rev (reverse most significant bits) = in perm h-0i-z
right_rev (reverse least significant bits) = in perm 0-hz-i
full_rev (reverse all bits) = in perm z-0
; ...
transpose = 0 txor in
```

Note: The efficient way to transpose a word is to use the TXOR instruction, because the PERM macro may expand to as many as five instructions. Using PERM here to compute `transpose` is for documentation purposes only.



# B

## Instruction reference

This appendix is taken from the assembler reference manual and tabulates opcodes and macros alphabetically by mnemonic. What these entries have in common is, they all translate directly into machine-language instructions that are fetched from program memory, decoded, and executed as a program runs. This appendix does not list keywords like `unsigned`, casts like `<wrap>`, and other language features that may control the assembler and affect program execution, but do not themselves emit executable instructions into the object file.

This appendix and its subject matter are still being actively written and revised.

# A

# Add

**c = a + b**

Register signedness		Flag set if	
Left	unsigned or signed	N	$a + b < 0$
Right	unsigned or signed	Z	$a + b = 0$
Dest.	unsigned or signed	T	c cannot fit $a + b$
8 opcodes total		R	T is set or R is already set

This is the instruction for ordinary addition of 36-bit numbers. It does not have a carry input or carry output. It is fully range-checked, so the T and R flags will indicate when the sum does not fit in 36 bits.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities, which then are added to produce a 38-bit signed sum that will not overflow. The N and Z flags are set based on the original 38-bit sum. The 36 least significant bits of the sum are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full sum does not fit, otherwise T is cleared and R is left unchanged.

# ABS

# Absolute value

**c = abs b**

Register signedness		Flag set if	
b	signed	N	never; flag is cleared
c	unsigned or signed	Z	$b = 0$
2 macros total		T	c cannot fit $ b $
		R	T is set or R is already set

This is a builtin macro that expands to two CPU instructions. The absolute value of **b** is written to **c**. This macro has full range checking: if **b** is  $-(2^{35})$  and **c** is signed, the result will not fit, and flags T and R will be set, and Z will be cleared because the true result is not zero. Otherwise T is cleared, R is left unchanged, and Z is set if the result is zero. N always is cleared.

If  $-(2^{30}) \leq b < 2^{30}$  can be guaranteed, **FABS** is a faster alternative to **ABS**.

**ABS** is not implemented for and will not assemble if **b** is unsigned.

## AC

## Add with carry

$c = a ++ b$

Register signedness		Flag set if	
Left	unsigned or signed	N	$a + b + T < 0$
Right	unsigned or signed	Z	$a + b + T = 0$
Dest.	unsigned or signed	T	c cannot fit $a + b + T$
8 opcodes total		R	T is set or R is already set

This is the final instruction for multiple-precision addition of integers larger than 36 bits. It uses the T flag as a carry input, but has no carry output. It is fully range-checked, so the T and R flags will indicate when the multiple-precision sum does not fit.

This instruction is preceded by **AW** for 72-bit addition, or by **AWC** for 108-bit and larger addition. It is never preceded by **A**, because **A** conflicts for range checking.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities, which then are added along with the T flag to produce a 38-bit signed sum that will not overflow. The N and Z flags are set based on the original 38-bit sum. The 36 least significant bits of the sum are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full sum does not fit, otherwise T is cleared and R is left unchanged.

## AND

## AND

$c = a \& b$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
1 opcode only		Z	all result bits are zero

**AND** sets the destination to the bitwise AND of its operands. N and Z are set as if the destination is a signed register. T and R do not change.

# ASL

# Arithmetic shift left

`c = a asl cw`

Register signedness		Flag set if	
<code>a</code>	unsigned or signed	N	<code>a &lt; 0</code>
<code>cw</code>	ignored	Z	<code>a = 0</code>
<code>c</code>	unsigned or signed	T	<code>c</code> cannot fit full result
4 opcodes total		R	T is set or R is already set

The 2-instruction `PSL` and `ASL` sequence enables safe, range-checked power-of-two multiplication, despite its many signedness combinations and corner cases. `ASL` multiplies left operand `a` by a non-negative power of two, and writes the product's 36 least significant bits to destination `c`. If the full result does not fit in `c` without loss of information, the T and R flags will be set, otherwise T is cleared and R does not change.

The `PSL` instruction is used to convert the desired exponent of two into control word `cw`. This exponent is clamped to a maximum of 36, and then copied into all tribbles.

## ASR

## Arithmetic shift right

`c = a asr cw`

Register signedness		Flag set if	
<code>a</code>	unsigned or signed	N	<code>a &lt; 0</code>
<code>cw</code>	ignored	Z	<code>a = 0</code>
<code>c</code>	unsigned or signed	T	<code>c</code> cannot fit full result
4 opcodes total		R	T is set or R is already set

The 2-instruction **PSR** and **ASR** sequence enables safe, range-checked division by powers of two, despite its many signedness combinations and corner cases. **ASR** divides left operand `a` by a non-negative power of two with rounding towards  $-\infty$ , and writes the quotient's 36 least significant bits to destination `c`. If the rounded quotient does not fit in `c` without loss of information, the T and R flags will be set, otherwise T is cleared and R does not change.

Note that because of the rounding towards  $-\infty$ , **ASR** is not (for most purposes) a stand-alone means for dividing numbers that are or may be negative by powers of two.

The **PSR** instruction is used to convert the desired exponent of two into control word `cw`. If the exponent is 36 or more, it is clamped to 36. If the exponent is zero, it is left as zero. Otherwise, the exponent is subtracted from 36 in order to represent it from the internal hardware perspective of a left rotation. After this clamping, leaving as zero, or subtracting, the exponent is copied to all tribbles.



## AW

## Add with wrap

<wrap> c = a + b

Register signedness		Flag set if	
Left	ignored	N	never; flag is cleared
Right	ignored	Z	$a + b = 0$
Dest.	ignored	T	$a + b \geq 2^{36}$
	1 opcode only	R	flag does not change

This is the first instruction for multiple-precision addition of integers larger than 36 bits. It has no carry input, and uses the T flag as a carry output. It does not require range checking and therefore has no effect on the R flag.

This instruction is followed by AC for 72-bit addition. For 108-bit and larger integers, it is followed by AWC.

The three registers are treated as unsigned without regard to how they are declared. The operands are added, and the 36 least significant bits of the sum are stored in the destination. Flag N is cleared. Flag Z will be set if the left and right operands are both zero, and cleared otherwise because the sum, however truncated, cannot truly be zero. Flag T is set if a carry is generated, and cleared otherwise. Flag R does not change.

**AWC**

## Add with wrap and carry

<wrap> c = a ++ b

Register signedness		Flag set if	
Left	ignored	N	never; flag is cleared
Right	ignored	Z	$a + b + T = 0$
Dest.	ignored	T	$a + b + T \geq 2^{36}$
	1 opcode only	R	flag does not change

This is the intermediate instruction for multiple-precision addition of integers larger than 72 bits. It uses the T flag as a carry input and carry output. It does not require range checking and therefore has no effect on the R flag.

In 108-bit addition, this instruction is preceded by **AW** and followed by **AC**. For 144-bit and larger integers, it is preceded by **AW** or **AWC** and followed by **AWC** or **AC** depending on its position.

The three registers are treated as unsigned without regard to how they are declared. The operands are added along with the T flag, and the 36 least significant bits of the sum are stored in the destination. Flag N is cleared. Flag Z will be set if both operands and the incoming T flag are all zero, and cleared otherwise because the sum, however truncated, cannot truly be zero. Flag T is set if a carry is generated, and cleared otherwise. Flag R does not change.

## B0 –

## Brighten ones

```
c = bol b      c = bor b
c = boli b     c = bori b
```

Register signedness		Flag set if
All	ignored	N bit 35 of the result is set
	4 opcodes	Z all result bits are zero

BOL, BOLI, BOR and BORI are builtin macros that expand to two CPU instructions each. BOL/BOR ignores the leading/trailing ones of `b`, replaces the adjacent bit with one, replaces all other bits with zeros, and writes the result to `c`. BOLI/BORI is BOL/BOR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## BOUND

## Bound

```
bound i < lim
```

Register signedness		Generate interrupt if
i	ignored	i < 0 or i ≥ lim
lim	unsigned or signed	
	2 opcodes total	

This instruction provides a two-sided array boundary check in one CPU cycle. The array presumably has less than  $2^{35}$  elements, which is guaranteed to be the case if less than 144 Gbytes of RAM is installed. This allows index `i` to have any signedness, because it will be unconditionally out of bounds—either because negative or excessively positive—whenever the leftmost bit is set.

The upper limit `lim` may be signed or unsigned, and represents the number of elements that may be safely accessed. If `lim` ≤ 0, index `i` is always out of bounds, because there is no safe memory location for access. Otherwise, the maximum permitted index is `lim` − 1. If `i` is out of bounds, this instruction generates an interrupt, otherwise this instruction does nothing. In any event, no registers are written to, and no flags change.

The required < in the syntax is to remind the programmer of the operand positions.

## BZ –

## Brighten zeros

`c = bzl b`

`c = bzs b`

`c = bzli b`

`c = bzri b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	4 opcodes	Z	all result bits are zero

BZL, BZLI, BZR and BZRI are builtin macros that expand to two CPU instructions each. BZL/BZR replaces the leading/trailing zeros of `b` with ones, replaces the adjacent bit with one, replaces all other bits with zeros, and writes the result to `c`. BZLI/BZRI is BZL/BZR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

# CALL

# Call

<code>call subr</code>	<code>call &lt; subr</code>
<code>call -t subr</code>	<code>call &lt;= subr</code>
<code>call +t subr</code>	<code>call == subr</code>
<code>call -r subr</code>	<code>call != subr</code>
<code>call +r subr</code>	<code>call &gt;= subr</code>
	<code>call &gt; subr</code>

No registers used	No flags changed
11 opcodes total	

The **CALL** instructions pushes the current instruction pointer on the call stack, and transfers control to the subroutine with name **subr**. When a **RETURN** instruction is executed by the subroutine later, the instruction address on the stack will be popped, and the program will continue with the instruction following **CALL**.

For security reasons, the call stack is stored in a dedicated SRAM IC. The sole electrical access to the stack SRAM's data connects to the instruction pointer exclusively. The only instructions that can access this memory's contents are the **CALL** and **RETURN** instructions. Thus the stack is not used for other purposes in the manner of other architectures, such as for local variables. Because the architecture does not support recursion via the stack, stack overflow is unlikely and in any event cannot cause privilege escalation.

Calls can be conditioned on the N, R, T, or Z flags. The **-t** and **+t** designators cause the call to occur only if T is clear or set, respectively. If the call does not occur, execution continues with the instruction that immediately follows. The **-r** and **+r** designators do the same using the R flag.

The **<**, **<=**, **==**, **!=**, **>=**, and **>** designators operate as if a **cmp a b** instruction immediately preceded the **call**, with the call taken if **a** is less than, less than or equal, equal to, not equal to, greater than or equal to, or greater than **b**, respectively. Equivalently, the call will be taken only under the following corresponding N and Z flag states:

<b>&lt;</b> N is set	<b>&gt;</b> N is clear
<b>&lt;=</b> N or Z is set	<b>&gt;=</b> N is clear or Z is set
<b>==</b> Z is set	<b>!=</b> Z is clear

Note that N and Z are never simultaneously set.

It's important to use **==** instead of merely **=** for the equality test, because **call = subr** syntactically means to copy a register named **subr** to a register named **call**.

## CL0 Count leading ones

`c = clo b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 macro only	Z	all result bits are zero

This builtin macro that expands to three CPU instructions. The number of leading one bits in `b` is counted and written to `c`. N is cleared, and Z is set iff `b = 0`. T and R do not change.

## CLZ Count leading zeros

`c = clz b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 macro only	Z	all bits of <code>b</code> are ones

This builtin macro that expands to three CPU instructions. The number of leading zero bits in `b` is counted and written to `c`. N is cleared, and Z is set iff `b = 777777777777'o` (all ones). T and R do not change.

## CMP

## Compare

`cmp a - b`

Register signedness		Flag set if	
Left	unsigned or signed	N	$a < b$
Right	unsigned or signed	Z	$a = b$
4 opcodes total			

This instruction subtracts the right operand from the left and sets the N and Z flags according to the result. These flags will be correct for any combination of inputs; there is no overrange situation that can occur. The result is discarded, and the T and R flags do not change.

## CRF

## Clear range flag

`crf`

No registers used	Flag set if
1 opcode only	T R was previously set
	R never; flag is cleared

This is the only instruction that can clear the R (Range) flag. The R flag is copied to T, and then R is cleared. Here is a code example for how to save and restore the R flag:

```
unsigned save_R

; save R flag
crf                                ; previous R now in T
save_R = 0 ++ 0                    ; add with carry saves T

; restore R flag
crf                                ; R is now clear
save_R = 0 - save_R                ; possible overflow restores R
```

## CTO

## Count trailing ones

`c = clo b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 macro only	Z	all result bits are zero

This builtin macro that expands to three CPU instructions. The number of trailing one bits in `b` is counted and written to `c`. N is cleared, and Z is set iff `b = 0`. T and R do not change.

## CTZ

## Count trailing zeros

`c = clz b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 macro only	Z	all bits of <code>b</code> are ones

This builtin macro that expands to three CPU instructions. The number of trailing zero bits in `b` is counted and written to `c`. N is cleared, and Z is set iff `b = 777777777777'o` (all ones). T and R do not change.



# CX

## Check and extend

`cw = cx f`

Register signedness		Flag set if	
<code>f</code>	unsigned or signed	N	bit 35 of the result is set
<code>cw</code>	ignored	Z	all result bits are zero
	1 opcode only	T	<code>f &lt; 0</code> or <code>f &gt; 63</code>
		R	T is set or R is already set

CX is used to prepare control words for several instructions, including MH and ML. This single-instruction macro verifies that  $0 \leq f \leq 63$ , and then replicates the least significant tribble of `f` to all of the others. The T and R flags are set if the range check fails, otherwise T is cleared and R does not change.

# DSL

## Double shift left

`c = a dsl b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction adds the T flag with wrapping to `a`, and then shifts the sum left six bits. The six bits shifted in at the right are the six leftmost bits of `b`. The result is written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

This is a key instruction for long multiplication, providing in one CPU cycle what would otherwise take five cycles. Listing 7.2 has sample code. The N and Z flags are not used for long multiplication, but are available in case someone identifies a use for them later.

## EO –

## Erase ones

```
c = eol b      c = eor b
c = eoli b     c = eori b
```

Register signedness		Flag set if
All	ignored	N bit 35 of the result is set
	4 macros	Z all result bits are zero

EOL, EOLI, EOR and EORI are builtin macros that expand to two CPU instructions each. EOL/EOR replaces any leading/trailing ones of **b** with zeros, and write the result to **c**. EOLI/EORI is EOL/EOR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## EZ –

## Erase zeros

```
c = ezl b      c = ezr b
c = ezli b     c = ezri b
```

Register signedness		Flag set if
All	ignored	N bit 35 of the result is set
	4 macros	Z all result bits are zero

EZL, EZLI, EZR and EZRI are builtin macros that expand to two CPU instructions each. EZL/EZR replaces any leading/trailing zeros of **b** with ones, and writes the result to **c**. EZLI/EZRI is EZL/EZR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## FABS

## Fast absolute value

`c = fabs b`

Register signedness		Flag set if	
b	signed	N	never; flag is cleared
c	unsigned or signed 1 opcode only	Z	$b = 0$
		T	$b < -(2^{35})$ or $b \geq 2^{35}$
		R	T is set or R is already set

This is a single-instruction implementation of absolute value. The architecture only supports this operation if the six leftmost bits of the operand are either all ones or all zeros. This instruction is fully range-checked, so if the operand is not within the supported range, the T and R flags will both be set, and N and Z will both be cleared.

If `b` is within the supported range, its absolute value is written to `c`. T and N are cleared, and R is left unchanged. Z is set if `b` is zero, and cleared otherwise.

If  $-(2^{30}) \leq b < 2^{30}$  is too restrictive for the application, ABS offers a full-word “slow” absolute value operation using two instructions.

FABS is not implemented for and will not assemble if `b` is unsigned.

## F0 -

## Find one

`c = fol b`                      `c = for b`  
`c = foli b`                    `c = fori b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	4 macros	Z	all result bits are zero

FOL, FOLI, FOR and FORI are builtin macros that expand to two CPU instructions each. FOL/FOR scans `b` for the leftmost/rightmost one bit, sets all other bits to zero, and writes the result to `c`. If `b` does not contain any ones, the output is all zeros. FOLI/FORI is FOL/FOR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## FZ –

## Find zero

```
c = fzl b      c = fzs b
c = fzli b     c = fzri b
```

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	4 macros	Z	all result bits are zero

FZL, FZLI, FZR and FZRI are builtin macros that expand to two CPU instructions each. FZL/FZR scans **b** for the leftmost/rightmost zero bit, sets that bit to one, sets all other bits to zero, and writes the result to **c**. If **b** does not contain any zeros, the output is all zeros. FZLI/FZRI is FZL/FZR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## GO –

## Grow one

```
c = gol b      c = gor b
c = goli b     c = gori b
```

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	4 macros	Z	all result bits are zero

GOL, GOLI, GOR and GORI are builtin macros that expand to two CPU instructions each. GOL/GOR replaces the leftmost/rightmost zero of **b** with one, and writes the result to **c**. If **b** does not contain any zeros, the output is all ones. GOLI/GORI is GOL/GOR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## GZ –

## Grow zero

`c = gzl b`                      `c = gزر b`  
`c = gzli b`                      `c = gzri b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	4 macros	Z	all result bits are zero

GZL, GZLI, GZR and GZRI are builtin macros that expand to two CPU instructions each. GZL/GZR replaces the leftmost/rightmost one of `b` with zero, and writes the result to `c`. If `b` does not contain any ones, the output is all zeros. GZLI/GZRI is GZL/GZR with all output bits inverted. N and Z are set as if the destination is a signed register. T and R do not change.

## HALT

## Halt

`halt`

No registers used	No flags changed
1 opcode only	

As of October 2020, this is a vestigial instruction that causes the virtual machine to quit. Instructions for yielding the CPU and possibly waiting for interrupts will be worked out in the next few weeks. This entry will be removed or seriously altered.

## IPSR

## Instruction pointer shift register

`c = ipsr b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

**As of September 2022, this instruction has been removed. The instruction pointer now uses a linear counter.**

This instruction sets `c` to the successor of `b` according to the Galois linear feedback shift register corresponding to  $x^{13} + x^{10} + x^4 + x + 1$ . This is done by shifting `b` right one position, and then clearing bit 12. If `b` was odd prior to the shift (that is, a one was shifted out), then the quantity is XORed with 11011'o, otherwise no XOR is done. The final result is written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

This LFSR is used by the operating system and programming tools to emulate the behavior of the instruction pointer 13 least significant bits, which do not simply increment. If `b` is initially 1, the next five terms generated are 4617, 6925, 8079, 7630, 3815. The predecessor of 1 is 2, so 2 is the last output before the cycle loops every 8 191 terms. This LFSR follows the paging scheme of the instruction pointer, in that the 23 most significant bits do not change. Note that if the least 13 significant bits are all zero, `c` will be equal to `b`, and the LFSR will not progress.

# JUMP

# Jump

jump dest	jump < dest
jump -t dest	jump <= dest
jump +t dest	jump == dest
jump -r dest	jump != dest
jump +r dest	jump >= dest
	jump > dest

No registers used	No flags changed
11 opcodes total	

The **JUMP** instructions transfer control to the instruction at label **dest**. Nearly always, this label should be within the present scope. Here is a sample:

```
again: nop
      jump again    ; infinite loop
```

Jumps can be conditioned on the N, R, T, or Z flags. The **-t** and **+t** designators cause the jump to occur only if T is clear or set, respectively. If the jump does not occur, execution continues with the instruction that immediately follows. The **-r** and **+r** do the same using the R flag.

The **<**, **<=**, **==**, **!=**, **>=**, and **>** designators operate as if a **cmp a b** instruction immediately preceded the **jump**, with the jump taken if **a** is less than, less than or equal, equal to, not equal to, greater than or equal to, or greater than **b**, respectively. Equivalently, the jump will be taken only under the following corresponding N and Z flag states:

<b>&lt;</b> N is set	<b>&gt;</b> N is clear
<b>&lt;=</b> N or Z is set	<b>&gt;=</b> N is clear or Z is set
<b>==</b> Z is set	<b>!=</b> Z is clear

Note that N and Z are never simultaneously true.

It's important to use **==** instead of merely **=** for the equality test, because **jump = dest** syntactically means to copy a register named **dest** to a register named **jump**.

**JUMP** may have important pipelining consequences. Stay tuned.

Here is a simple double loop with 6300 inner iterations:

```

                                unsigned i j
                                i = 0
                                j = 0
outer:  cmp i 90
                                jump >= outer_done
inner:  cmp j 70
                                jump >= inner_done
                                say "i = " i " and j = " j
                                j = j + 1
                                jump inner
inner_done: i = i + 1
                                jump outer
outer_done: nop

```

## LANR

## Left and not right

`c = a & !b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

LANR sets the destination to the bitwise AND of the left operand with the bitwise complement of the right operand. N and Z are set as if the destination is a signed register. T and R do not change.



# LAS

# Logical assignment

<wrap> c = b

<wrap> c = <left> a

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

LAS copies the operand to the destination without range checking. N and Z are set as if the destination is a signed register. T and R do not change.

By default, the operand is taken from the right copy of the register file. To force use of the left copy, use the <left> cast as shown.

## LFSR

## Linear feedback shift register

`c = lfsr b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction sets `c` to the successor of `b` according to the Galois linear feedback shift register corresponding to  $x^{36} + x^{31} + x^{13} + x^7 + x^6 + x^5 + x^3 + x^2 + 1$ . This is done by shifting `b` right one position, with bit 35 filled with zero. If `b` was odd prior to the shift (that is, a one was shifted out), then the shifted amount is XORed with 410000\_010166'o, otherwise no XOR is done. The final result is written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

LFSRs are useful, because they can key fast pseudorandom number generators (PRNGs) implemented with MIX. See that instruction for sample code. The period of this LFSR is  $2^{36} - 1$  for any nonzero initial value. The period of the resulting PRNG is guaranteed to be at least  $2^{36} - 1$ , but has not been adequately validated for longer lengths.

The polynomial chosen for this LFSR can also be used, with powers raised by 36, 72, or 108, as the most significant word of 72-bit, 108-bit, and 144-bit LFSRs. This turns out to be the *only* polynomial with eight taps or fewer which has this property. Such LFSRs are useful for producing PRNGs with longer guaranteed periods. See also XPOLY.

## LO -

## Light ones

`c = lol b`                      `c = lor b`  
`c = loli b`                     `c = lori b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	2 opcodes, 2 macros	Z	all result bits are zero

LOL/LOR ignores the leading/trailing ones of `b`, replaces all other bits with zeros, and writes the result to `c`. LOLI/LORI is LOL/LOR with all output bits inverted, implemented as a two-instruction macro. N and Z are set as if the destination is a signed register. T and R do not change.

## LONR

## Left or not right

`c = a | !b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

LONR sets the destination to the bitwise OR of the left operand with the bitwise complement of the right operand. N and Z are set as if the destination is a signed register. T and R do not change.

## LSL

## Logical shift left

`c = a lsl cw`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction shifts the bits of register `a` left. Bits shifted out are discarded, and bits shifted in are zeros. The number of positions to shift may not be negative, and must be copied into every tribble of control word `cw`. The PSL instruction can convert any unsigned value into a suitable control word. N and Z are set as if the destination is a signed register. T and R are not changed by LSL or any preceding PSL.

## LSR

## Logical shift right

`c = a lsr cw`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction shifts the bits of register `a` right. Bits shifted out are discarded, and bits shifted in are zeros. The number of positions to shift may not be negative, is specified from the perspective of a left rotation (sic), and must be copied into every tribble of control word `cw`. The PSR instruction can convert any unsigned value into a suitable control word. N and Z are set as if the destination is a signed register. T and R are not changed by LSR or any preceding PSR.

## LZ -

## Light zeros

$c = \text{lzl } b$                        $c = \text{lzr } b$   
 $c = \text{lzli } b$                        $c = \text{lzri } b$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	2 opcodes, 2 macros	Z	all result bits are zero

LZL/LZR replaces the leading/trailing zeros of **b** with ones, replaces all other bits with zeros, and writes the result to **c**. LZLI/LZRI is LZL/LZR with all output bits inverted, implemented as a two-instruction macro. N and Z are set as if the destination is a signed register. T and R do not change.

## MAX

## Maximum

$c = a \text{ max } b$

Register signedness		Flag set if	
Left	unsigned or signed	N	maximum < 0
Right	unsigned or signed	Z	maximum = 0
Dest.	unsigned or signed	T	c cannot fit maximum
	8 opcodes total	R	T is set or R is already set

The maximum of the two operands is determined, and the N and Z flags are set accordingly. The 36 least significant bits of the maximum are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full maximum does not fit, otherwise T is cleared and R is left unchanged.

## MH

## Multiply high

`c = a mh b`

Register signedness		Flag set if	
Left	ignored	N	never; flag is cleared
Right	ignored	Z	$c = 0$
Dest.	ignored	T	$c \bmod 64 \neq 0$
	1 opcode only	R	T is set or R is already set

This is a key instruction for unsigned “short” multiplication where one of the factors fits into six bits, and the product fits into 36 bits. The smaller of the factors must be copied into all of the tribbles via `CX` or an assembler constant. `MH` multiplies the tribbles of `a` and `b` pairwise, but the six 12-bit results cannot fit the 6-bit spaces afforded by the tribbles of `c`. Instead, `MH` retains only the six most significant bits of each 12-bit result. `ML` is the complementary instruction that retains the six least significant bits of each.

To meaningfully add the output of `MH` and `ML`, their place values must be aligned consistently, meaning that `MH` needs a 6-position left shift, and that the result can spill to as many as 42 bits (which will not fit in a 36-bit register) as a result of that shift. The solution is that instead of shift, `MH` rotates its result six bits left. If the six bits rotated into the rightmost places are not all zeros, the T and R flags are set because the eventual product will not fit in 36 bits. Otherwise T is cleared, R is left unchanged, and the output of `MH` can be directly added to `ML` to obtain the 36-bit product. Z will be set if the output of `MH` is all zeros. N is always cleared.

Here is an unsigned short multiplication example with full range checking, and an always-accurate Z flag at the end whether or not overflow occurs. Four cycles are needed. The `CX` can be optimized out when multiplying by a small constant.

```
unsigned big small t result    ; will multiply big * small
; ...
t = cx small                    ; copy small into all tribbles
result = big mh t               ; high bits of product
t = big ml t                    ; low bits of product
result = result + t             ; result is now big * small
```

## MHNS

## Multiply high no shift

`c = a mhns b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 opcode only	Z	<code>c = 0</code>

This is a key instruction for unsigned long multiplication, where two 36-bit factors are multiplied as 6-bit tribbles and eventually sum to produce a 72-bit result. **MHNS** multiplies the tribbles of **a** and **b** pairwise, but the six 12-bit results cannot fit the 6-bit spaces afforded by the tribbles of **c**. Instead, **MHNS** retains only the six most significant bits of each result. The tribbles are output in their original positions, instead of being rotated left as with **MH**. The Z flag is set if the outcome of **MHNS** is all zeros, and cleared otherwise. N is always cleared, and T and R do not change. See listing 7.2 for sample code.

## MIN

## Minimum

`c = a min b`

Register signedness		Flag set if	
Left	unsigned or signed	N	minimum < 0
Right	unsigned or signed	Z	minimum = 0
Dest.	unsigned or signed	T	<code>c</code> cannot fit minimum
	8 opcodes total	R	T is set or R is already set

The minimum of the two operands is determined, and the N and Z flags are set accordingly. The 36 least significant bits of the minimum are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full minimum does not fit, otherwise T is cleared and R is left unchanged.

## MIRD

## Mirrored decrement

`c = mird b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction subtracts one from `b` mod  $2^{36}$  as if `b`'s place values are in reverse order.

For example, if `b = 00000_00000_000011_110101_000011_011110'2`  
then `mird b = 11111_11111_111101_110101_000011_011110'2`.

Because no range checking is done, the T flag is cleared and R is left unchanged. N is not “mirrored,” but is a copy of the leftmost output bit. In the example above, N is set. Z is set if all output bits are set, and cleared otherwise.

## MIRI

## Mirrored increment

`c = miri b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction adds one to `b` mod  $2^{36}$  as if `b`'s place values are in reverse order.

For example, if `b = 11111_11111_111101_110101_000011_011111'2`  
then `mird b = 00000_00000_000011_110101_000011_011111'2`.

Because no range checking is done, the T flag is cleared and R is left unchanged. N is not “mirrored,” but is a copy of the leftmost output bit. In the example above, N is cleared. Z is set if all output bits are set, and cleared otherwise.



## MIX

## Mix

$$c(\text{iphertext}) = p(\text{laintext}) \text{ mix } k(\text{ey})$$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

MIX passes 36-bit word **p** through an invertible substitution-permutation network keyed by 36-bit word **k**. The inverse operation of MIX is XIM. Testing shows that on average, one-bit changes to the value of **p/k** cause **c** to change by 15.37/16.47 bits. An ideal mixing function would cause half of the bits of **c**—half is 18 bits—to change. N and Z are set as if the destination is a signed register. T and R do not change. Here are sample uses for hashing an object, pseudorandom numbers, and cryptography:

```

; Compute hash of 4-word object
unsigned hash word.1 word.2 word.3 word.4
; ...
hash = o3THvL't mix word.1      ; use 36-bit const as initial value
hash = hash mix word.2          ; use progressive words of object
hash = hash mix word.3
hash = hash mix word.4

; Pseudorandom number generator
unsigned state1 state2 output
state1 = zRN6x1't                ; 36-bit seed #1
state2 = mPC$TB't                ; 36-bit seed #2
; ...
get_next_rand:                  ; period of PRNG >= (2**36) - 1
    state2 = lfsr state2        ; rekey in just one instruction
    state1 = state1 mix state2   ; new value
    output = state1             ; keep caller from changing state
    return

; Toy example cipher - insecure!
unsigned in out key.1 key.2
key.1 = EtdvIv't                ; bits 0-35 of key
key.2 = yKoM2j't                ; bits 36-71 of key
; ...
encrypt:
    out = in mix key.1           ; ECB mode with 36-bit block size
    out = out mix key.2         ; two rounds total
    return
;
decrypt:
    out = in xim key.2           ; "in" here is "out" from encrypt
    out = out xim key.1         ; "unwrap" key in reverse order
    return

```

## ML

## Multiply low

`c = a ml b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 opcode only	Z	<code>c = 0</code>

This is a key instruction for unsigned “short” multiplication where one of the factors fits into six bits, and the product fits into 36 bits. The smaller of the factors must be copied into all of the tribbles via **CX** or an assembler constant. **ML** multiplies the tribbles of **a** and **b** pairwise, but the six 12-bit results cannot fit the 6-bit spaces afforded by the tribbles of **c**. Instead, **ML** retains only the six least significant bits of each 12-bit result. The **Z** flag is set if the outcome of **ML** is all zeros, and cleared otherwise. **N** is always cleared, and **T** and **R** do not change. See **MH** for more information and sample code.

## NAND

## NAND

`c = a !& b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

**NAND** sets the destination to the bitwise NAND of its operands. **N** and **Z** are set as if the destination is a signed register. **T** and **R** do not change.

## NAS

## Numeric assignment

`c = b`

`c = <left> a`

Register signedness		Flag set if	
Left	unsigned or signed	N	operand < 0
Right	unsigned or signed	Z	operand = 0
Dest.	unsigned or signed	T	dest. cannot fit operand
8 variants total		R	T is set or R is already set

NAS copies the 36 operand bits to the 36 destination bits exactly, and then confirms that a change in signedness has not changed the resulting quantity. N and Z are set to indicate if the operand is negative or zero respectively. T and R are set if the operand does not fit in the destination. Otherwise, T is cleared, and R does not change.

By default, the operand is taken from the right copy of the register file. To force use of the left copy, use the `<left>` cast as shown.

## NOP

## No operation

`nop`

No registers used	No flags changed
1 opcode only	

This instruction sits out the current CPU cycle without writing to any register or changing any flags. If the control transfer instructions **CALL**, **JUMP**, and **RETURN** require flushing the instruction pipeline, **NOP** is a simple and fail-safe option to place immediately following these instructions.

Because the hardware ignores the 27 operand bits of **NOP**, the emulation software uses **NOP** to encode diagnostic instructions such as **SAY** and **SAYS** that have no hardware support. This allows the programs assembled for the virtual machine to run unmodified on physical hardware.

## NOR

`c = a !| b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

NOR sets the destination to the bitwise NOR of its operands. N and Z are set as if the destination is a signed register. T and R do not change.

## NOR

## NOT

`c = !b`

`c = <left> !a`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

NOT sets the destination to the bitwise NOT of the operand. N and Z are set as if the destination is a signed register. T and R do not change.

By default, the operand is taken from the right copy of the register file. To force use of the left copy, use the `<left>` cast as shown.

## NOT

# NUDGE

# Nudge

`c = a nudge b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

NUDGE replaces the rightmost 0–35 bits of `a` with the same number of bits from `b`, and writes the result to `c`. The number of bits replaced is the number that appear to the right of the leftmost one in `b`. In essence, the first one bit as `b` is scanned from left to right is the *start bit*, with all bits following that bit replacing the bits in the corresponding positions of `a`. For example, `1101_0110'b nudge 0010_1101'b` would be `1100_1101'b`.

NUDGE is useful for altering the modulus of a number relative to some power of two. For example if  $\lfloor a \div 256 = 12\,000 \rfloor$  and we want to force  $a \bmod 256 = 197 = 1100_0101'b$  without knowing its current value and without changing  $\lfloor a \div 256 \rfloor$ , we would set `c = a nudge 1_1100_0101'b`.

NUDGE is also useful for efficiently converting a pointer to a member of a power-of-two-sized structure to a pointer to any other member of the same structure, without needing to know which member was indicated by the original pointer.

# OR

# OR

`c = a | b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

OR sets the destination to the bitwise OR of its operands. N and Z are set as if the destination is a signed register. T and R do not change.

## PARTY

## Parity

`c = party b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 opcode only	Z	parity of <code>b</code> is even

This instruction determines whether the number of ones in `b` is odd, and if so sets `c` to 1 and clears the Z flag. Otherwise, `c` is zeroed and Z is set. The N and T flags are cleared, and R is left unchanged.

## PAT

## Permute across tribbles

`c = a pat b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction transposes `a` per figure 5.3. The tribbles of  $\mathbf{a}^\top$  then undergo permutation operations selected by the corresponding tribbles of `b`. The permuted outcome is then transposed back and written to `c`. Due to the 6-bit encoding limit of 64 operations, only 64 of the possible  $6! = 720$  permutations of six bits can be supported by this instruction. These 64 permutations appear in table B.1 and follow the notation of section A.8. All 720 permutations can be reached via 2-instruction compositions of these 64.

In essence, PAT does permutations *between* tribbles, where tribble  $i$  of `b` specifies a permutation among the  $i$ th bits of each tribble of `a`. PIT and PAT can be used in combination to produce any permutation of 36 bits in five instructions or fewer. As computing operands for these instructions manually would be tedious and error-prone, the assembler provides this service automatically via the PERM macro.

**Table B.1:** Tribble permutation operations.

Oct.	Perm.	Derivation	Oct.	Perm.	Derivation
00	012345	identity	40	031524	count out of circle by 3s
01	123450	rotated identity	41	142035	count out of circle by 3s
02	234501	rotated identity	42	253140	count out of circle by 3s
03	345012	rotated identity	43	304251	count out of circle by 3s
04	450123	rotated identity	44	415302	count out of circle by 3s
05	501234	rotated identity	45	520413	count out of circle by 3s
06	432105	reflected 05	46	035124	count out by 3s backward
07	321054	reflected 04	47	140253	count out by 3s backward
10	210543	reflected 03	50	251304	count out by 3s backward
11	105432	reflected 02	51	302415	count out by 3s backward
12	054321	reflected 01	52	413520	count out by 3s backward
13	543210	reflected identity	53	524031	count out by 3s backward
14	102345	pair swap	54	102354	2-pair rotation
15	210345	pair swap	55	315042	2-pair rotation
16	312045	pair swap	56	542310	2-pair rotation
17	412305	pair swap	57	513240	2-pair rotation
20	512340	pair swap	60	021435	3-pair rotation
21	021345	pair swap	61	034125	3-pair rotation
22	032145	pair swap	62	043215	3-pair rotation
23	042315	pair swap	63	103254	3-pair rotation
24	052341	pair swap	64	240513	3-pair rotation
25	013245	pair swap	65	453201	3-pair rotation
26	014325	pair swap	66	521430	3-pair rotation
27	015342	pair swap	67	534120	3-pair rotation
30	012435	pair swap	70	120534	symmetric half rotation
31	012543	pair swap	71	201453	symmetric half rotation
32	012354	pair swap	72	034512	algorithmically selected
33	452301	movement in pairs	73	035421	algorithmically selected
34	014523	movement in pairs	74	105243	algorithmically selected
35	230145	movement in pairs	75	130542	algorithmically selected
36	024135	$2 \times 3$ , $3 \times 2$ transposes	76	254310	algorithmically selected
37	031425	$2 \times 3$ , $3 \times 2$ transposes	77	510432	algorithmically selected



# PAIT Permute across and inside tribbles

`c = a paity b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

In unusual circumstances where PIT is applied to the result of PAT using the same operands, the ALU can roll these two operations into this single instruction. For instance, `c = a paity 131313131313'o` mirrors the bits of a 36-bit word (see table B.1).

# PIT Permute inside tribbles

`c = a pity b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

Each tribble of `a` undergoes a permutation operation selected by the corresponding tribble of `b`, with the result written to `c`. Due to the 6-bit encoding limit of 64 operations, only 64 of the possible  $6! = 720$  permutations of six bits can be supported by this instruction. These 64 permutations appear in table B.1 and follow the notation of section A.8. All 720 permutations can be reached via 2-instruction compositions of these 64.

## POPC

## Popcount

`c = popc b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
		Z	<code>b = 0</code>

POPC is a builtin assembler macro that expands to two esoteric CPU instructions that aren't in this appendix. At the conclusion of the sequence, `c` is equal to the number of one bits in `b`. This operation is sometimes called *population count*, *popcount*, or *Hamming weight* in other literature. N and Z are set as if the destination is a signed register. Because the result only uses the six rightmost bits, N will not be set. T and R do not change.

A third register is required for intermediate results during the computation; this register is provided transparently by the assembler and is shared with other builtin macros. Register `b` is never overwritten by this macro.

## PRL

## Prepare to rotate left

`cw = prl amt`

Register signedness		Flag set if	
<code>amt</code>	unsigned or signed	T	<code>amt &lt; 0</code> or <code>amt &gt; 63</code>
<code>cw</code>	ignored	R	T is set or R is already set
	1 opcode only		

PRL prepares a control word for the ROL instruction by copying the least significant tribble into all others. For instance, if `amt = 5`, then `cw = 050505050505'8`. Left rotations of 0 through 63 bits are supported. Rotating 36–63 bits is equivalent to rotating 0–27 bits. If `amt` is outside the supported range, the T and R flags are set. Otherwise, T is cleared and R does not change.

PRL is the same instruction as CX, but PRL is preferred for legibility.

# PRR Prepare to rotate right

`cw = prr amt`

Register signedness		Flag set if	
<code>amt</code>	unsigned or signed	T	<code>amt &lt; 0</code> or <code>amt &gt; 63</code>
<code>cw</code>	ignored 1 opcode only	R	T is set or R is already set

This instruction prepares a control word for the `ROL` instruction by first altering the least significant tribble to be from the perspective of a left rotation, then copying that tribble to all others. For instance, if `amt = 29`, then `cw = 070707070707'8`. Right rotations of 0 through 63 bits are supported. Rotating 36–63 bits is equivalent to rotating 0–27 bits. If `amt` is outside the supported range, the T and R flags are set. Otherwise, T is cleared and R does not change.

# PSL Prepare to shift left

`cw = psl amt`

Register signedness		Flag set if	
<code>amt</code>	unsigned	N	bit 35 of the result is set
<code>cw</code>	ignored 1 opcode only	Z	all result bits are zero

This instruction prepares a control word for an `ASL` or `LSL` instruction by copying the number of positions to shift into all tribbles. For instance, if `amt = 4`, then `cw = 040404040404'8`. Left shifts of any non-negative number of positions are supported. Shifts of more than 35 bits are all equivalent to 36-bit shifts and set `cw = 444444444444'8` (each tribble is 36). N and Z are set as if `cw` is a signed register. T and R do not change.

## PSR

## Prepare to shift right

`cw = psr amt`

Register signedness		Flag set if	
amt	unsigned	N	bit 35 of the result is set
cw	ignored	Z	all result bits are zero
	1 opcode only		

This instruction prepares a control word for an **ASR** or **LSR** instruction copying the number of positions to shift into all tribbles. This amount to shift is specified from the perspective of a left rotation (sic). For instance, if `amt = 34`, then `cw = 020202020202'8`. Right shifts of any non-negative number of positions are supported. Shifts of more than 35 bits are all equivalent to 36-bit shifts and set `cw = 444444444444'8` (each tribble is 36). N and Z are set as if `cw` is a signed register. T and R do not change.

## RANL

## Right and not left

`c = !a & b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

**RANL** sets the destination to the bitwise AND of the right operand with the bitwise complement of the left operand. N and Z are set as if the destination is a signed register. T and R do not change.

## ROL

## Rotate left

`c = a rot cw`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction rotates the bits of `a`. The number of positions to rotate must be specified from the perspective of a left rotation, and copied into every tribble of control word `cw`. The `PRL` and `PRR` instructions offer a range-checked mechanism to set up the control word for left and right rotations. `N` and `Z` are set as if the destination is a signed register. `ROL` will not change `T` or `R`, although a preceding `PRL` and `PRR` may.

## RONL

## Right or not left

`c = !a | b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

`RONL` sets the destination to the bitwise OR of the right operand with the bitwise complement of the left operand. `N` and `Z` are set as if the destination is a signed register. `T` and `R` do not change.

# RS

## Reverse subtract

$$c = a \sim - b$$

Register signedness		Flag set if	
Left	unsigned or signed	N	$b - a < 0$
Right	unsigned or signed	Z	$b - a = 0$
Dest.	unsigned or signed	T	c cannot fit $b - a$
8 opcodes total		R	T is set or R is already set

This is the instruction for reverse subtraction of 36-bit numbers. It does not have a borrow input or borrow output. It is fully range-checked, so the T and R flags will indicate when the difference does not fit in 36 bits.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities. The left operand is subtracted from the right to produce a 38-bit signed difference that will not overflow. The N and Z flags are set based on the original 38-bit difference. The 36 least significant bits of the difference are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full difference does not fit, otherwise T is cleared and R is left unchanged.

RSC

## Reverse subtract with carry

$$c = a \sim - b$$

Register signedness		Flag set if	
Left	unsigned or signed	N	$b - a - T < 0$
Right	unsigned or signed	Z	$b - a - T = 0$
Dest.	unsigned or signed	T	c cannot fit $b - a - T$
8 opcodes total		R	T is set or R is already set

This is the final instruction for multiple-precision reverse subtraction of integers larger than 36 bits. It uses the T flag as a borrow input, but has no borrow output. It is fully range-checked, so the T and R flags will indicate when the multiple-precision difference does not fit.

This instruction is preceded by **RSW** for 72-bit reverse subtraction, or by **RSWB** for 108-bit and larger reverse subtraction. It is never preceded by **RS**, because **RS** conflicts for range checking.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities. The T flag and left operand are subtracted from the right to produce a 38-bit signed difference that will not overflow. The N and Z flags are set based on the original 38-bit difference. The 36 least significant bits of the difference are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full difference does not fit, otherwise T is cleared and R is left unchanged.

**RSW**

## Reverse subtract with wrap

<wrap>  $c = a \sim - b$

Register signedness		Flag set if	
Left	ignored	N	$b - a < 0$
Right	ignored	Z	$b - a = 0$
Dest.	ignored	T	$b - a < 0$
	1 opcode only	R	flag does not change

This is the first instruction for multiple-precision reverse subtraction of integers larger than 36 bits. It has no borrow input, and uses the T flag as a borrow output. It does not require range checking and therefore has no effect on the R flag.

This instruction is followed by **RSB** for 72-bit reverse subtraction. For 108-bit and larger integers, it is followed by **RSWB**.

The three registers are treated as unsigned without regard to how they are declared. The T flag and left operand are subtracted from the right, and the 36-bit difference is stored in the destination. Flag Z will be set if the left and right operands are equal, and cleared otherwise. Flags N and T are set if a borrow is generated, and cleared otherwise. Flag R does not change.



## RSWC

## Reverse subtract and wrap and carry

<wrap>  $c = a \sim - b$

Register signedness		Flag set if	
Left	ignored	N	$b - a - T < 0$
Right	ignored	Z	$b - a - T = 0$
Dest.	ignored	T	$b - a - T < 0$
	1 opcode only	R	flag does not change

This is the intermediate instruction for multiple-precision reverse subtraction of integers larger than 72 bits. It uses the T flag as a borrow input and borrow output. It does not require range checking and therefore has no effect on the R flag.

In 108-bit reverse subtraction, this instruction is preceded by RSW and followed by RSB. For 144-bit and larger integers, it is preceded by RSW or RSWB and followed by RSWB or RSB depending on its position.

The three registers are treated as unsigned without regard to how they are declared. The T flag and left operand are subtracted from the right, and the 36-bit difference is stored in the destination. Flag Z will be set if the 36-bit difference is zero and no borrow is generated. Flags N and T are set if a borrow is generated, and cleared otherwise. Flag R does not change.

## RTGL

## Rotate T going left

$c = \text{rtgl } b$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero
		T	bit 35 of $b$ is set
		R	R is already set

This instruction rotates  $b$  left one position, fills bit 0 with the existing T flag, and writes the result to  $c$ . The bit rotated out from bit 35 is moved to the T flag. R is left unchanged. N and Z are set as if the destination is a signed register.

## RTGR

## Rotate T going right

$c = \text{rtgr } b$

Register signedness		Flag set if	
All	ignored	N	incoming T flag is set
	1 opcode only	Z	all result bits are zero
		T	bit 0 of $b$ is set
		R	R is already set

This instruction rotates  $b$  right one position, fills bit 35 with the incoming T flag, and writes the result to  $c$ . The bit shifted out from bit 0 is copied to the T flag. R is left unchanged. N and Z are set as if the destination is a signed register.

## S

## Subtract

$c = a - b$

Register signedness		Flag set if	
Left	unsigned or signed	N	$a - b < 0$
Right	unsigned or signed	Z	$a - b = 0$
Dest.	unsigned or signed	T	$c$ cannot fit $a - b$
	8 opcodes total	R	T is set or R is already set

This is the instruction for ordinary subtraction of 36-bit numbers. It does not have a borrow input or borrow output. It is fully range-checked, so the T and R flags will indicate when the difference does not fit in 36 bits.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities. The right operand is subtracted from the left to produce a 38-bit signed difference that will not overflow. The N and Z flags are set based on the original 38-bit difference. The 36 least significant bits of the difference are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full difference does not fit, otherwise T is cleared and R is left unchanged.

## SC

## Subtract with carry

$c = a - b$

Register signedness		Flag set if	
Left	unsigned or signed	N	$a - b - T < 0$
Right	unsigned or signed	Z	$a - b - T = 0$
Dest.	unsigned or signed	T	$c$ cannot fit $a - b - T$
8 opcodes total		R	T is set or R is already set

This is the final instruction for multiple-precision subtraction of integers larger than 36 bits. It uses the T flag as a borrow input, but has no borrow output. It is fully range-checked, so the T and R flags will indicate when the multiple-precision difference does not fit.

This instruction is preceded by **SW** for 72-bit subtraction, or by **SWB** for 108-bit and larger subtraction. It is never preceded by **S**, because **S** conflicts for range checking.

The 36-bit unsigned and/or signed operand registers are extended into 38-bit signed quantities. The T flag and right operand are subtracted from the left to produce a 38-bit signed difference that will not overflow. The N and Z flags are set based on the original 38-bit difference. The 36 least significant bits of the difference are stored in the destination register, which may be signed or unsigned. Flags T and R are set if the full difference does not fit, otherwise T is cleared and R is left unchanged.

## STGL

## Shift T going left

$c = \text{stgl } b$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
1 opcode only		Z	all result bits are zero
		T	bit 35 of $b$ is set
		R	R is already set

This instruction shifts  $b$  left one position, fills bit 0 with a zero, and writes the result to  $c$ . The bit shifted out from bit 35 is copied to the T flag. R is left unchanged. N and Z are set as if the destination is a signed register.

## STGR

## Shift T going right

`c = stgr b`

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
	1 opcode only	Z	all result bits are zero
		T	bit 0 of <code>b</code> is set
		R	R is already set

This instruction shifts `b` right one position, fills bit 35 with a zero, and writes the result to `c`. The bit shifted out from bit 0 is copied to the T flag. R is left unchanged, and N is cleared. Z is set if the result is all zeros, and cleared otherwise.

## SW

## Subtract with wrap

`<wrap> c = a - b`

Register signedness		Flag set if	
Left	ignored	N	$a - b < 0$
Right	ignored	Z	$a - b = 0$
Dest.	ignored	T	$a - b < 0$
	1 opcode only	R	flag does not change

This is the first instruction for multiple-precision subtraction of integers larger than 36 bits. It has no borrow input, and uses the T flag as a borrow output. It does not require range checking and therefore has no effect on the R flag.

This instruction is followed by **SB** for 72-bit subtraction. For 108-bit and larger integers, it is followed by **SWB**.

The three registers are treated as unsigned without regard to how they are declared. The T flag and right operand are subtracted from the left, and the 36-bit difference is stored in the destination. Flag Z will be set if the left and right operands are equal, and cleared otherwise. Flags N and T are set if a borrow is generated, and cleared otherwise. Flag R does not change.

# SWC Subtract with wrap and carry

<wrap> c = a - b

Register signedness		Flag set if	
Left	ignored	N	$a - b - T < 0$
Right	ignored	Z	$a - b - T = 0$
Dest.	ignored	T	$a - b - T < 0$
	1 opcode only	R	flag does not change

This is the intermediate instruction for multiple-precision subtraction of integers larger than 72 bits. It uses the T flag as a borrow input and borrow output. It does not require range checking and therefore has no effect on the R flag.

In 108-bit subtraction, this instruction is preceded by SW and followed by SB. For 144-bit and larger integers, it is preceded by SW or SWB and followed by SWB or SB depending on its position.

The three registers are treated as unsigned without regard to how they are declared. The T flag and right operand are subtracted from the left, and the 36-bit difference is stored in the destination. Flag Z will be set if the 36-bit difference is zero and no borrow is generated. Flags N and T are set if a borrow is generated, and cleared otherwise. Flag R does not change.

# SWIZ Swizzle

c = a swiz cw

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

This instruction replaces tribbles of  $a^T$  using the tribbles of cw as swizzle function identifiers. These 64 swizzle functions appear in table B.2. See TXOR for a synopsis of the transposing operation from a to  $a^T$ . N and Z are set as if the destination is a signed register. T and R do not change, because no range checking is done.

A routine use for SWIZ is to select a tribble from a register and replicate it to all of the other tribbles. For example, if  $a = 01\ 34\ 67\ 90\ 23\ 56\ '8$  and  $cw = 02\ 02\ 02\ 02\ 02\ 02\ '8$ , the result will be  $90\ 90\ 90\ 90\ 90\ 90\ '8$ .

**Table B.2:** Swizzle operations.

Slot	Operation
0	copy from tribble 0
1	copy from tribble 1
2	copy from tribble 2
3	copy from tribble 3
4	copy from tribble 4
5	copy from tribble 5
6–63	reserved

TXOR

Transposing XOR

`c = a txor b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

The bits of `b` are transposed to compute  $\mathbf{b}^\top$  by relocating the  $i$ th bit of tribble  $j$  to the  $j$ th bit of tribble  $i$  for all  $i, j \in \{0\dots5\}$ . The bitwise exclusive-OR of `a` with  $\mathbf{b}^\top$  is then written to destination `c`. N and Z are set as if the destination is a signed register. T and R do not change.

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

## XIM

## Undo mix

$p(\text{laintext}) = c(\text{iphertext}) \text{ xim } k(\text{ey})$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

XIM is the inverse operation of MIX. XIM passes 36-bit word  $c$  through an inverted substitution-permutation network keyed by 36-bit word  $k$ . N and Z are set as if the destination is a signed register. T and R do not change. See MIX for more specifics.

Testing shows that on average, one-bit changes to the value of  $c/k$  cause  $p$  to change by 15.36/16.48 bits. Note these measurements are distinguishable from those of MIX, and could be indicative of S-box imbalances.

## XNOR

## XNOR

$c = a \text{ !}^{\wedge} b$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

XNOR sets the destination to the bitwise XNOR of its operands (the opposite of XOR). N and Z are set as if the destination is a signed register. T and R do not change.

# XOR

# XOR

$$c = a \wedge b$$

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
1	opcode only	Z	all result bits are zero

XOR sets the destination to the bitwise XOR of its operands. N and Z are set as if the destination is a signed register. T and R do not change.



# XPOLY

# XOR polynomial if T is set

`c = xpoly b`

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 opcode only	Z	all result bits are zero

If the T flag is set, this instruction XORs `b` with `410000_010166'o` and writes the result to `c`. Otherwise `b` is copied directly to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

XPOLY is used to implement 72-bit, 108-bit, and 144-bit Galois linear feedback shift registers with periods of  $2^{72} - 1$ ,  $2^{108} - 1$ , and  $2^{144} - 1$  respectively. Because it would take the CPU at least millions of years to cycle through the 72-bit sequence, applications that require 108 or 144 bits would be rare. But 36-bit LFSRs implemented by the LFSR instruction can restart their cycles in less than a day, so 72-bit LFSRs would find application. The XPOLY polynomial is not suitable for (and may produce very short sequences if used for) LFSRs of more than 144 bits.

Here are sample implementations for all four LFSR sizes. The word registers must be initialized before the sequence starts, with at least one register in each case not zero. Note from section A.5 that `u.` is a supported shorthand meaning `unsigned`. When an LFSR is used to key MIX for a pseudorandom number generator, any single register from the LFSR is adequate.<sup>1</sup>

<code>; 36-bit LFSR</code>	<code>; 72-bit LFSR</code>	<code>; 108-bit LFSR</code>	<code>; 144-bit LFSR</code>
<code>u. w0</code>	<code>u. w1 w0</code>	<code>u. w2 w1 w0</code>	<code>u. w3 w2 w1 w0</code>
<code>;</code>	<code>;</code>	<code>;</code>	<code>;</code>
<code>w0 = lfsr w0</code>	<code>w1 = stgr w1</code>	<code>w2 = stgr w2</code>	<code>w3 = stgr w3</code>
	<code>w0 = rtgr w0</code>	<code>w1 = rtgr w1</code>	<code>w2 = rtgr w2</code>
	<code>w1 = xpoly w1</code>	<code>w0 = rtgr w0</code>	<code>w1 = rtgr w1</code>
		<code>w3 = xpoly w3</code>	<code>w0 = rtgr w0</code>
			<code>w3 = xpoly w3</code>

<sup>1</sup>This should be verified with Dieharder before finalizing this section. There are eight untested cases.

# C

## Advance corrections

When I documented specifics of the implementation [Abel22b], there were several places where I found the implementation needs minor updates, in order to improve the future usability of both the implementation and this dissertation. These circuit changes, although small, call for careful execution, testing, and some prerequisite improvements to the simulation environment. This dissertation was finished before the pending implementation changes could be made.

To make this dissertation as useful as possible to future readers, it has been written as if near-term minor edits to the implementation have already been released. This appendix contains a short description of each place where, as of November 2022, the implementation is still catching up to the text of this dissertation.

### C.1 Instruction format fields to move

The field locations for some of the instruction formats are moving. The implementation [Abel22b] uses these formats:

#### Immediate value instructions

opcode	immediate value	dest. register
bits 35–27	bits 26–9	bits 8–0

### ALU instructions

opcode	left register	right register	dest. register
bits 35–27	bits 26–18	bits 17–9	bits 8–0

The implementation will be corrected to conform with this dissertation, which specifies:

### Immediate value instructions

opcode	dest. register	immediate value
bits 35–27	bits 26–18	bits 17–0

### ALU instructions

opcode	dest. register	left register	right register
bits 35–27	bits 26–18	bits 17–9	bits 8–0

The instruction word output for Listing 11.8’s code fetches was altered to conform to the intended field order.

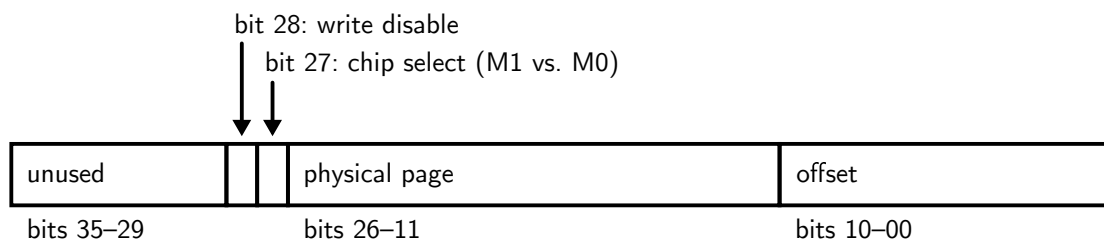
## C.2 ALU operation to be deleted

The implementation includes a  $\gamma$  layer operation  $\gamma.imb$  (immediate both), which allows an 18-bit immediate value in a CPU instruction to be repeated into a 36-bit immediate value. The  $\beta$  layer does the necessary swizzle, but the bits come out in the wrong positions.  $\gamma.imb$  exchanges the necessary three-bit fields to produce the correct immediate value.

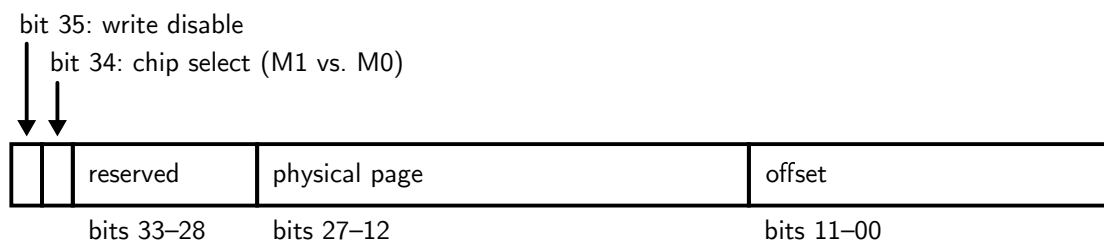
After the immediate value repositioning of section C.1,  $\beta$ ’s swizzle will place all subwords in their correct order, and  $\gamma.imb$  can be replaced with any identity-capable  $\gamma$  operation such as  $\gamma.add$ . The  $\gamma.imb$  operation will be purged from the implementation.

## C.3 Physical address format to change

The implementation's physical address format for data memory is:



The implementation is being changed to use this format:



There are two reasons for this change. The visually obvious one is that the first format will not allow the physical page field to grow contiguously if larger SRAM ICs enter the market. In the new format, the write disable and chip select bits are moved out of the way, and the intermediate bits are reserved to support a larger physical page field.

The less obvious change is, the offset field is growing from 11 to 12 bits, meaning that the physical pages themselves are increasing from 2048 to 4096 words. I had meant for the the address format work for the largest-available 36-wide ICs, which are  $4\text{Mi} \times 36$ . But I made a mistake: the virtual page and offset together only came to 22 bits—enough for one SRAM, but I forgot that the board accommodates a second SRAM. By increasing the page size, a fully-expanded system (based on presently available components) with 8Mi words of data memory can be addressed.

Another reason to increase the offset field from 11 to 12 bits is so that page table bypass flip-flop  $ff\ b$  can be eliminated at a future time in favor of using RAMs  $\alpha_0$  and  $\alpha_1$ . Because the  $\alpha$  RAMs operate on six-bit slices, using a six-bit multiple for the offset field is helpful. This limited  $\alpha$  RAM use in parallel with the page table may support an additional addressing mode for certain reads. Section 8.6.2 describes this suggestion further.

# D

## What’s where in the source tree

Because the minicomputer implementation actively being developed, the contents and arrangement of the source tree are expected to change. This appendix lists a coarse inventory of the source code as of November 2022.

**Table D.1:** `code/` Implementation directory root.

directory	description
<code>asm/</code>	assembler and virtual machine
<code>consts/</code>	constants that represent opcodes and operations
<code>firmware/</code>	modules to compute SRAM firmware
<code>logic-solver/</code>	synthesize optimal SN74AUC-series glue logic
<code>misc/</code>	helper functions, constants, and macros
<code>netlist/</code>	“electrical source code” of the minicomputer and conversion software
<code>netsim/</code>	“electrical object code” of the minicomputer and simulation software
<code>vm/</code>	testing for assembler and ALU firmware

This directory also contains the only `makefile` for the implementation.

**Table D.2:** code/asm/ Assembler and virtual machine.

file	contents
alu.c	ALU for the virtual machine, including colored trace output
asm.c	outermost routines for assembler and virtual machine
asm.h	API for assembler and virtual machine
context.h	non-shared struct for the API's opaque pointer
errors.c	de-duplicates and reports errors found by assembler
lex.c	assembler terminals and low-level parsing
no-bu	list of files that GNU Tar need not save
symc.c	symbol table implementation for assembler
syntax.c	assembler high-level parsing and instruction encoding

**Table D.3:** code/consts/ Constants that represent opcodes and operations.

file	contents
opcode.h	assigned numbers and source code names for opcodes
slots.h	assigned numbers and source code names for ALU operations

**Table D.4:** code/firmware/ Modules to compute SRAM firmware.

file	contents
alpha.c	ALU $\alpha$ layer
audited.c	short functions exempted from separately-written test cases
beta.c	ALU $\beta$ layer
decoder.c	non-ALU portion of control decoder RAMs D0 and D1
firmware.c	firmware memory allocation and loading for simulations
firmware.h	local header file for firmware modules
gamma.c	ALU $\gamma$ layer
gen-sbox.py	compute S-boxes derived from $\sqrt{2} \div 2$
instruct.c	ALU portion of control decoder RAMs D0 and D1
no-bu	list of files that GNU Tar need not save
pieces.c	firmware-computing functions that don't fit neatly elsewhere
sbox.c	automatically generated by gen-sbox.py
theta.c	ALU $\theta$ RAM
unary.c	stacked unary operations (simple unary are in audited.c)
uninit.c	obsolete stubs for "initializing" non-firmware RAMs
zeta.c	ALU $\zeta$ RAM

**Table D.5:** code/logic-solver/ Synthesize optimal SN74AUC-series glue logic.

file	contents
alpha-lock-permutations	what-if scenarios for various control signal meanings
freebie demonstration.odt	optimal boolean functions of 3 variables for SRAM enable inputs
logic solver demonstration.odt	optimal boolean functions of 3 variables
no-bu	list of files that GNU Tar need not save
size	database of size-optimized boolean functions of 4 variables
size-freebies	“size” recomputed for use as SRAM enable inputs
solve.c	solver program for fastest/smallest SN74AUC circuits
speed	database of speed-optimized boolean functions of 4 variables
speed-freebies	“speed” recomputed for use as SRAM enable inputs
tables.py	Python batch job to compute glue logic for node lockouts

**Table D.6:** code/misc/ Helper functions, constants, and macros.

file	contents
misc.c	general support functions, mostly not architecture-specific
misc.h	general and architecture-specific definitions
unused.c	unused code emulating the $\beta$ layer transposition



**Table D.7:** code/netlist/ “Electrical source code” of the minicomputer and software to process it.

<b>file</b>	<b>contents</b>
2021 topology.ods	27 spreadsheets of circuit details; possibly out of sync with timing study.ods
feb.netlist	first-attempt CPU netlist started February 2, 2021 (human-written)
floorplan.svg	latest component floorplan generated by <code>run.netlist.py</code>
frozen.netlist	version of <code>feb.netlist</code> that generated <code>netsim/frozen.ns</code>
generated.bom	bill of materials, component list, and run statistics from <code>run.netlist.py</code>
generated.net	<code>run.netlist.py</code> output in a format that KiCad 5.0.2 can load
net-everything.svg	net connectivity markings on top of <code>floorplan.svg</code>
no-bu	list of files that GNU Tar need not save
run.netlist.py	main netlist processing program; its inputs are <code>feb.netlist</code> and <code>tiles</code>
tiles	circuit board locations for all components (human-written)
timing study.ods	possible unintentional fork of 2021 topology.ods; needs check for merge

**Table D.8:** `code/netsim/` “Electrical object code” of the minicomputer and software to simulate it.

<b>C source code</b>	<b>description</b>
<code>compo.h</code>	structs that represent components and pins
<code>comps/c_1g74.c</code>	simulated D flip-flop with preset and clear
<code>comps/c_244.c</code>	simulated quad 4-bit buffer with output enable
<code>comps/c_2gxx.c</code>	simulated AND, BUF, INV, NAND, NOR, OR, XOR
<code>comps/c_374.c</code>	simulated dual 8-bit D flip-flop
<code>comps/c_io.c</code>	simulated component for generic test hook
<code>comps/c_osc.c</code>	simulated crystal oscillator
<code>comps/c_pull.c</code>	simulated pull resistor
<code>comps/c_reset.c</code>	simulated voltage monitor (power-on reset)
<code>comps/c_sram.c</code>	simulated synchronous SRAM
<code>comps/c_switch.c</code>	simulated DIP switch
<code>connect.c</code>	deals with components, connections, short circuits, etc.
<code>load.c</code>	parser for simulation scripts
<code>ns.c</code>	root source file and <code>main()</code> for the simulator
<code>pin.c</code>	simulates component input, output, and clock input pins
<code>queue.c</code>	heap-based generic priority queue (wrapped by <code>quser.c</code> )
<code>quser.c</code>	time-based priority queue for netlist simulation
<code>util.c</code>	minor tidbit functions not used elsewhere
<b>other file</b>	<b>description</b>
<code>connectivity.ns</code>	connections and timings computed by <code>run.netlist.py</code>
<code>fib.a</code>	assembly: infinite loop of overflowing Fibonacci numbers
<code>fib.ns</code>	script: load <code>frozen.ns</code> and run <code>fib.a</code>
<code>fibclean.a</code>	assembly: compute xth Fibonacci number, $0 \leq x \leq 53$
<code>fibclean.ns</code>	script: load <code>frozen.ns</code> and run <code>fibclean.a</code>
<code>fire.report</code>	overcurrent/short-circuit measurements from last <code>ns</code> run
<code>frozen.ns</code>	<code>connectivity.ns</code> as computed using <code>frozen.netlist</code>
<code>last-test.ns</code>	symbolic link to last script run from <code>ns</code>
<code>no-bu</code>	list of files that GNU Tar need not save
<code>nop.a</code>	assembly: 8 NOPs and a HALT for system power-up tests
<code>nop.ns</code>	script: load <code>frozen.ns</code> and run <code>nop.a</code>
<code>ns</code>	binary executable for electrical simulation
<code>regress/</code>	directory of forgotten regression tests (may still work)
<code>writeprotect.ns</code>	script: disable SRAM writes for purpose of testing
<code>zero-pc.ns</code>	script: force program counters to zero for purpose of testing

**Table D.9:** `code/vm/` Testing for assembler and ALU firmware.

<b>file</b>	<b>contents</b>
<code>asm-tests.c</code>	<code>asm-tests/</code> folder converted to static C strings
<code>asm-tests/abs.at</code>	assembly: absolute value regression test
<code>asm-tests/dist.test</code>	assembly: mixed-signage arithmetic example
<code>asm-tests/ham2.at</code>	assembly: two-cycle popcount regression test
<code>asm-tests/imm.test</code>	assembly: manual verification of immediate opcodes
<code>asm-tests/lmul.at</code>	assembly: 64-bit multiplication regression test
<code>asm-tests/rpop.at</code>	assembly: R(ange) flag save and restore regression test
<code>asm-tests/smul.at</code>	assembly: absolute value regression test
<code>gen-asm-tests.py</code>	tool to convert <code>asm-tests/*.at</code> to <code>asm-tests.c</code>
<code>no-bu</code>	list of files that GNU Tar need not save
<code>perms.c</code>	regression test code for permutations
<code>stats.c</code>	statistical tests for MIX and XIM instructions
<code>tests.c</code>	regression tests for 109 ALU opcodes
<code>vm</code>	tool to test assembler and ALU firmware in virtual machine
<code>vm.c</code>	<code>main()</code> routine for <code>vm</code>

# References

- [Abel12] Marc W. Abel. 2012. Practical, scalable alternative session encryption using one-time pads. arXiv:1212.5086. doi: 10.48550/arXiv.1212.5086
- [Abel14a] Marc W. Abel. 2014. Clique network protocol and reference implementation. Retrieved Nov. 12, 2022 from <https://clique4.us>
- [Abel14b] Marc. W. Abel and Soon. M. Chung. 2014. Computing preset dictionaries from text corpora for the compression of messages. In *2014 Int. Conf. Data Software Eng. (ICoDSE)*, Nov. 26–27, 2014, Bandung, Indonesia. IEEE, New York, NY, USA, 5 pages. doi: 10.1109/ICODSE.2014.7062490
- [Abel15] Marc. W. Abel and Soon. M. Chung. 2015. Defending one-time pad cryptosystems from denial-of-service attacks. In *2015 Int. Conf. Data Software Eng. (ICoDSE)*, Nov. 25–26, 2015, Yogyakarta, Indonesia. IEEE, New York, NY, USA, 77–82. doi: 10.1109/ICODSE.2015.7436975
- [Abel18] Marc W. Abel. 2018. Apocalypse not: Practical security against state-sponsored shenanigans. Audio. 46 minutes. <https://talk.wakesecure.com>
- [Abel21] Marc W. Abel. 2021. Solder-defined architectures for trusted computing. In *NAECON 2021—IEEE Nat. Aerosp. Electron. Conf.*, Aug. 16–19, 2021, Dayton, OH, USA. IEEE, New York, NY, USA, 246–253. doi: 10.1109/NAECON49338.2021.9696432
- [Abel22a] Marc W. Abel. 2022. Parallel Multiplier Synthesis Software. Harvard Dataverse. (Sep. 4, 2022). doi: 10.7910/DVN/DABIBJ
- [Abel22b] Marc W. Abel. 2022. 36-Bit Minicomputer Using SRAM ICs as Logic Elements. Harvard Dataverse. (Nov. 12, 2022). doi: 10.7910/DVN/SOHO2F

- [Afshar08] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. 2008. Improving synthesis of compressor trees on FPGAs via integer linear programming. In *Proc. Conf. Des. Automat. Test Europe*, March 10–14, 2008, Munich, Germany. IEEE, New York, NY, USA 1256–1261. doi: 10.1109/DATE.2008.4484851
- [Appelbaum13] Jacob Appelbaum, Laura Poitras, Marcel Rosenbach, Christian Stocker, Jorg Schindler and Holger Stark. 2013. Documents reveal top NSA hacking unit. *Spiegel*, Dec. 29, 2013. Retrieved Apr. 4, 2020 from <https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html>
- [Appleby16] Austin Appleby. 2016. SMHasher is a test suite designed to test the distribution, collision, and performance properties of non-cryptographic hash functions. Retrieved Oct. 15, 2022 from <https://github.com/aappleby/smhasher>
- [Aumasson12] Jean-Philippe Aumasson and Daniel Julius Bernstein. 2012. SipHash: A fast short-input PRF. In *13th Int. Conf. Cryptology India*, Dec. 9–12, 2012, Kolkata, India. *Lect. Notes Comput. Sci.* 7668. Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-34931-7\_28
- [Bailleux16a] Olivier Bailleux. 2016. A CPU made of ROMs. Watched Apr. 4, 2020 from <https://www.youtube.com/watch?v=J-pyCxMg-xg>
- [Bailleux16b] 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>
- [Bamford82] James Bamford. 1982. *The Puzzle Palace: Inside the National Security Agency, America's Most Secret Intelligence Organization*. Houghton Mifflin, Boston, MA, USA.
- [Bamford95] James Bamford and Wayne Madsen. 1995. *The Puzzle Palace* (2nd. ed.). Penguin Books. Never released, but a citation can be found in [Schneier96]. I sought and obtained further verification that the second edition was in fact written.
- [Bamford01] James Bamford. 2001. *Body of Secrets: Anatomy of the Ultra-Secret National Security Agency*. Doubleday, New York, London, Toronto, Sydney, Auckland.
- [Bamford04] James Bamford. 2004. *A Pretext for War: 9/11, Iraq, and the Abuse of America's Intelligence Agencies*. Doubleday, New York, London, Toronto, Sydney, Auckland.

- [Bamford08] James Bamford. 2008. *The Shadow Factory: The Ultra-Secret NSA from 9/11 to the Eavesdropping on America*. Doubleday, New York, London, Toronto, Sydney, Auckland.
- [Baugh73] Charles R. Baugh and Bruce A. Wooley. 1973. A two's complement parallel array multiplication algorithm. *IEEE Trans. Comput.* C-22, 12 (Dec. 1973), 1045–1047. doi: 10.1109/T-C.1973.223648
- [Becker13] 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
- [Bhargavan16] Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS '16)*. ACM, New York, NY, USA, 456–467. doi: 10.1145/2976749.2978423
- [Booth51] Andrew D. Booth. 1951. A signed binary multiplication technique. *Quart. J. Mech. Appl. Math.* 4, 2 (Jan. 1951), 236–240. doi: 10.1093/qjmam/4.2.236
- [Bratus12] 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).
- [Brewer13] Cynthia A. Brewer, ed. 2013. ColorBrewer 2.0: Color Advice for Cartography. Geography, Pennsylvania State University. University Park, PA, USA. <https://colorbrewer2.org>
- [Brown20] 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>
- [Butterfield16] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr, eds. 2016. *A Dictionary of Computer Science* (7th ed.). Oxford University Press, Oxford, England.
- [Buzbee10] Bill Buzbee. 2010. Magic-1 is a completely homebuilt minicomputer. Retrieved Sep. 6, 2022 from <http://www.homebrewcpu.com>
- [Collet21] Yann Collet. 2021. xxHash Library. Retrieved Oct. 15, 2022 from <https://github.com/Cyan4973/xxHash>

- [CTS18] CTS Labs. 2018. Severe Security Advisory on AMD Processors. CTS Labs, Tel Aviv, Israel.
- [Dadda65] Luigi Dadda. 1965. Some schemes for parallel multipliers. *Alta Freq.* 34, 5 (May 1965), 349–356.
- [Dannenberg10] 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
- [Domas18] Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. At *Black Hat USA 2018*, (Las Vegas, NV), white paper.
- [Dovgalyuk19] Pavel Dovgalyuk. 2019. Relay computer computes 7 digits of pi. Video retrieved Apr. 4, 2020 from <https://www.youtube.com/watch?v=b00Cfx2EN10>
- [Eddy1889] Mary Baker G. Eddy. 1889. *Science and Health with Key to the Scriptures* (40th ed.), p. 254. J. A. J. Wilcox, Boston, MA. Retrieved Jan. 3, 2023 from [https://en.wikisource.org/wiki/Science\\_and\\_Health\\_with\\_Key\\_to\\_the\\_Scriptures\\_\(1889\)](https://en.wikisource.org/wiki/Science_and_Health_with_Key_to_the_Scriptures_(1889))
- [Ermolov17] 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.
- [Ermolov20] 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>
- [Gambrell20] Jon Gambrell and Josef Federman. 2020. Fireworks, ammonium nitrate likely fueled Beirut explosion. Associated Press, Aug. 5, 2020. Retrieved Aug. 6, 2020 from <https://apnews.com/cbeb3263d6fc30a63a0300f588e7207b>
- [Glaisher1889] James Whitbread Lee Glaisher. 1889. The method of quarter-squares. *Nature* 40 (Oct. 10, 1889), 573–576. doi: 10.1038/040573c0
- [Galassi21] Mark Galassi et al. 2021. GNU Scientific Library Reference Manual (3rd. ed.). <https://www.gnu.org/software/gsl>
- [Garisto22] Daniel Garisto. 2022. ArXiv.org reaches a milestone and a reckoning: Runaway success and underfunding have led to growing pains for the preprint server. (Jan. 10, 2022). <https://www.scientificamerican.com/article/arxiv-org-reaches-a-milestone-and-a-reckoning/>

- [Greenemeier17] 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/>
- [Han19] Jin-Woo Han, Myeong-Lok Seol, Dong-Il Moon, Gary Hunter, and M. Meyyappan. 2019. Nanoscale vacuum channel transistors fabricated on silicon carbide wafers. *Nat. Electr.* 2 (Aug. 26, 2019), 405–411. doi: 10.1038/s41928-019-0289-z
- [Hatamian86] Medhi Hatamian and Glenn L. Cash. 1986. A 70-MHz 8-bit  $\times$  8-bit parallel pipelined multiplier in 2.5- $\mu$ m CMOS. In *IEEE J. Solid-State Circuits* 21, 4 (Aug. 1986), 505–513. doi: 10.1109/JSSC.1986.1052564
- [Holler17] 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
- [IBM22] IBM Security and Ponemon Institute LLC. 2022. *Cost of a Data Breach Report 2022*. Retrieved Sep. 6, 2022 from <https://www.ibm.com/downloads/cas/3R8N1DZJ>
- [IEEE19] Institute of Electrical and Electronic Engineers. 2019. IEEE 764-2019. *IEEE Standard for Floating-Point Arithmetic*. doi: 10.1109/IEEESTD.2019.8766229
- [ISO18] 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](https://standards.iso.org/ittf/PubliclyAvailableStandards/c073906_ISO_IEC_27000_2018_E.zip)
- [Iwata06] Tetsu Iwata. 2006. New blockcipher modes of operation with beyond the birthday bound security. In *Fast Software Encryption 2006 (FSE 2006)*, March 15–17, 2006, Graz, Austria. *Lect. Notes Comput. Sci.* 4047 (July 6, 2006), Springer, Berlin, Heidelberg. doi: 10.1007/11799313\_20
- [Jansky33] Karl G. Jansky. 1933. Electrical disturbances apparently of extraterrestrial origin. *Proc. Inst. Radio Eng.* 21, 10 (Oct. 1933), 1387–1398. doi: 10.1109/JRPROC.1933.227458



- [Janushkevich20] 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>
- [Johnson72] Nigel Johnson. 1973. Improved binary multiplication system. *Electron. Lett.* 9, 1 (Jan. 1973), 6–7. doi: 10.1049/el:19730005
- [Johnson80] Everett L. Johnson. 1980. A digital quarter square multiplier. *IEEE Trans. Comput.* C-29, 3 (Mar. 1980), 258–261. doi: 10.1109/TC.1980.1675558
- [KiCad22] KiCad project. 2022. KiCad EDA: A Cross Platform and Open Source Electronics Design Automation Suite. Retrieved Sep. 7, 2022 from <https://www.kicad.org/>
- [Knuth81] Donald Ervin Knuth. 1981. *The Art of Computer Programming*, Vol. 2: Seminumerical Algorithms (2nd. ed.), 1–5. Addison-Wesley, Reading, MA, USA.
- [Kobayashi81] Hideaki Kobayashi. 1981. A fast multi-operand multiplication scheme. In *1981 IEEE 5th Symp. Comput. Arith. (ARITH)*, May 16–19, 1981, Ann Arbor, MI, USA. IEEE, New York, NY, USA, 246–250. doi: 10.1109/ARITH.1981.6159279
- [Kocher19] 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
- [Lehman61] Meir M. Lehman and Naphtali Burla. 1961. Skip techniques for high-speed carry-propagation in binary arithmetic units. *IRE Trans. on Electron. Comput.* EC-10, 4 (Dec. 1961), 691–698. doi: 10.1109/TEC.1961.5219274
- [Ling90] Huey Ling. 1990. An approach to implementing multiplication with small tables. *IEEE Trans. Comput.* 39, 5 (May 1990), 717–718. doi: 10.1109/12.53588
- [Lipp18] 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

- [Love11] 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
- [Schlaepfer16] Eric Schlaepfer. 2016. The MOnSter 6502. Retrieved Sep. 6, 2022 from <https://monster6502.com>
- [Mora05] Higinio Mora Mora, Jerónimo Manuel Mora Pascual, José Luis Sánchez Romero and Francisco Antonio Pujol López. 2005. Partial product reduction based on look-up tables. *19th Int. Conf. VLSI Des. & 5th Int. Conf. on Embed. Syst. Des. (VLSID '06)*, Jan. 3–7, 2006, Hyderabad, India. IEEE, New York, NY, USA, 6 pages. doi: 10.1109/VLSID.2006.130
- [Mora08] Higinio Mora Mora, Jerónimo Manuel Mora Pascual, José Luis Sánchez Romero and Juan Manuel García-Chamizo. 2008. Partial product reduction by using look-up tables for  $M \times N$  multiplier. *Integration* 41, 4 (July 2008), 557–571. doi: 10.1016/j.vlsi.2008.01.005
- [Mutlu19] Onur Mutlu and Jeremie S. Kim. 2019. RowHammer: A retrospective. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*. doi: 10.1109/TCAD.2019.2915318
- [Nichols22] Steven J. Vaughan-Nichols. 2022. Securing open-source code isn't going to be cheap. (Feb. 9, 2022). Retrieved Sep. 21, 2022 from [https://www.theregister.com/2022/02/09/secure\\_open\\_source\\_software](https://www.theregister.com/2022/02/09/secure_open_source_software)
- [NSA08] 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](https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf)
- [NSA22] 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](https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF)
- [Obama13] Barack Obama. 2013. Statement by the President. Jun. 7, 2013, (San Jose, CA), transcript. Retrieved Apr. 4, 2020 from <https://obamawhitehouse.archives.gov/the-press-office/2013/06/07/statement-president>
- [Oklobdzija96] Vojin G. Oklobdzija, David Villeger and Simon S. Liu. 1996. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Trans. Comput.* 45, 3 (Mar. 1996), 294–306. doi: 10.1109/12.485568

- [Ousterhout21] John Kenneth Ousterhout. 2021. Eliminate page limits for final paper versions. Retrieved Nov. 11, 2022 from <https://web.stanford.edu/~ouster/cgi-bin/pageLimits.php>
- [Patterson83] David Andrew Patterson. 1983. Microprogramming. *Sci. Am.* 248, 3 (Mar. 1983), 50–57. Retrieved Oct. 12, 2022 from <https://www.jstor.org/stable/24968851>
- [Paul09] Bipul C. Paul, Shinobu Fujita, and Masaki Okajima. 2009. ROM-based logic (RBL) design: a low-power 16 bit multiplier. *IEEE J. Solid-State Circuits* 44, 11 (Nov. 2009), 2935–2942. doi: 10.1109/JSSC.2009.2028928
- [Pompeo20] 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>
- [Portnoy17] 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>
- [Private21] Manufacturer representative. 2021. Private communication. Jan. 25, 2021.
- [Rice53] 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
- [Russell78] Richard M. Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (Jan. 1978), 63–72. doi: 10.1145/359327.359336
- [Rutkowska15a] 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/x86_harmful.pdf)
- [Rutkowska15b] Joanna Rutkowska. 2015. State considered harmful: A proposal for a stateless laptop. (Dec. 2015). Retrieved Sep. 26, 2022 from [https://blog.invisiblethings.org/papers/2015/state\\_harmful.pdf](https://blog.invisiblethings.org/papers/2015/state_harmful.pdf)
- [Schneier96] Bruce Schneier. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, Chichester, Brisbane, Toronto, Singapore.

- [Schwartz08] 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
- [Seacord14] 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.
- [Spafford89] Eugene Howard Spafford. 1989. The Internet worm: Crisis and aftermath. *Commun. ACM* 32, 6 (June 1989), 678–687. doi: 10.1145/63526.63527
- [Stoll88] Clifford Paul Stoll. 1988. Stalking the wily hacker. *Commun. ACM* 31, 5 (May 1988), 484–497. doi: 10.1145/42411.42412
- [Stenzel77] William J. Stenzel, William J. Kubitz and Gilles H. Garcia. 1977. A compact high-speed parallel multiplication scheme. *IEEE Trans. Comput.* C-26, 10 (Oct. 1977), 948–957. doi: 10.1109/TC.1977.1674730
- [Tatar18] 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.
- [TI75] Texas Instruments Inc. 1975. *TMS 1000 Series MOS/LSI One-Chip Microcomputers Programmer’s Reference Manual*. Texas Instruments Inc., Dallas, TX.
- [TI98] Texas Instruments Inc. 1998. *AVC Logic Family Technology and Applications*. Texas Instruments Inc., Dallas, TX. Retrieved Sept. 23, 2022 from <https://www.ti.com/lit/pdf/scea006>
- [TI02] Texas Instruments Inc. 2002. *AUC: Advanced Ultra-Low-Voltage CMOS Logic (product bulletin)*. Texas Instruments Inc., Dallas, TX. Retrieved Sept. 23, 2022 from <https://www.ti.com/lit/pdf/sceb011>
- [TienLin73] Tien-Lin Chang. 1973. Binary read-only-memory multiplier. *Electron. Lett.* 9, 25 (Dec. 1973), 580–581. doi: 10.1049/el:19730429
- [Toomey17] Warren Toomey (Ed.). 2017. Homebrew computers web-ring. Retrieved Sep. 6, 2022 from <https://www.homebrewcypuring.org>
- [VanBulck20] 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

- [vandenHuevel15] Jan van den Huevel. 2015. The arXiv cannot replace traditional publishing without addressing the standards of research assessment. (Oct. 23, 2015). <https://blogs.lse.ac.uk/impactofsocialsciences/2015/10/23/open-repositories-arxiv-scientific-publishing/>
- [Verma07] Ajay K. Verma and Paolo Ienne. 2007. Automatic Synthesis of Compressor Trees: Reevaluating Large Counters. In *2007 Des. Automat. Test Europe Conf. Exhib.*, Apr. 16–20, 2007, Nice, France. IEEE, New York, NY, USA, 6 pages. doi: 10.1109/DATE.2007.364632
- [Waksman14] 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
- [Wallace64] Christopher Stewart Wallace. 1964. A suggestion for a fast multiplier. *IEEE Trans. Electr. Comput.* EC-13, 1 (Feb. 1964), 14–17. doi: 10.1109/PGEC.1964.263830
- [Ward96] William Grady Ward. 1996. Moby Thesaurus II, distributed by Zeke Sikelianos, <https://moby-thesaurus.org>
- [Webster13] *Webster's Revised Unabridged Dictionary*. 1913. G. & C. Merriam Co., Springfield, MA, USA.
- [Weinberger58] Arnold Weinberger and Jay L. Smith. 1958. A logic for high-speed addition. *Natl. Bur. Stand. Circ.* 591, 1 (Feb. 14, 1958), 3–12. doi: 10.6028/NBS.CIRC.591
- [Weinberger81] Arnold Weinberger. 1981. 4:2 carry-save adder module. *IBM Tech. Discl. Bull.* 23, 8 (Jan. 1981), 3811–3814.
- [Weste11] Neil H. E. Weste and David Money Harris. 2011. *CMOS VLSI Design: A Circuits and Systems Perspective* (4th. ed.). Addison-Wesley, New York, NY, USA, 475–476.