Dauug|36 Assembler Reference

Marc W. Abel

December 2020

Contents

Chapter 1

Architecture Introduction

1.1 Machine word structure

The Dauug | 36 word size is 36 bits. The rightmost bit is bit 0 and has significance 2^0. The leftmost is bit 35 with significance 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 (ALU) 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 ith bit of tribble j to the jth bit of tribble i for all i, $j \in \{0...5\}$. The written notation uses the top superscript: if `w` is a word, then $\texttt{w}^\top$ is its transpose, and $\texttt{w}^{\top\top}$ is `w` transposed twice, which is simply `w`.

The bit positions of `w` and $\texttt{w}^\top$ can also be written as 6×6 square matrices. The transpose operation is the ordinary reflection through the main diagonal:

`w`	$\texttt{w}^\top$
`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`

For linear notations, the transpose operation can be written thus. If

`zyxwvu tsrqpo nmlkji hgfedc ba9876 543210` name the bits of `b`, then
`ztnhb5 ysmga4 xrlf93 wqke82 vpjd71 uoic60` are the bits of $\texttt{b}^\top$.

1.2 Register organization

Dauug | 36 minicomputers have 65 536 36-bit registers, allowing up to 128 simultaneously available programs called *users* to have 512 registers each. Within a program, all registers have identical capabilities. To lessen bottlenecks in the CPU, each register has two copies: 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.

1.3 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 Dauug | 36 specification, 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 obvious two-instruction solutions, but the hardware multiplier would be added to make the architecture faster. What is needed is a one-instruction solution.

The solution is a *reverse subtract* arithmetic operation. 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, reverse subtraction has been a standard feature from the outset in order to preserve continuity in the future.

1.4 Organization of data memory

Dauug | 36 program, data, and stack memory are stored on physically separate SRAM chips and have distinct address spaces.

Data memory is memory programs can read to and write from 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. There is no cache, because the architecture already can read to or write from this memory in one CPU cycle without stalling.

Data memory is allocated in pages of 8 192 words (36 864 bytes) and protected via a page table. This page size ensures that a minimal configuration of 1 Mi words of data memory can support 128 users with one small page for each. In principle, certain classes of short programs can run without any data memory whatsoever, because of the large number of available registers. But these programs would have miniscule support from the operating system, because the primary data exchange between the OS and user is via data memory at specified addresses.

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 page the operating system has directed them to. The operating system is responsible for providing a mechanism for programs to query what memory range(s) are valid, because programs may get confused if they stray out of bounds. Wayward programs will not, however, 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 are strictly limited to what the operating system lets it do.

All memory privileges for a given page are identical; the architecture does not support read-only pages. Any page that can be read by a user can be written by that user, and vice versa.

Dauug | 36 uses a load-store architecture, meaning that instructions either do an ALU operation, or do a memory operation. From the programmer's perspective, both never happen in the same instruction. This segregation simplifies the instruction set, keeps all instructions to a single CPU cycle, and frees the ALU during memory operations to aid with offset calculations.

1.5 Organization of program memory

The program memory contains the instructions that are fetched, decoded, and executed in order to run a program. Other than this fetch-decode-execute process, which happens transparently, users have no access at all to their program memory. Enforcement of this instruction works very simply: program memory can only be accessed via specific CPU instructions, and the operating system's loader refuses permit any such instructions in application programs. The rigid machine instruction format (see Section 1.7) makes it straightforward to exclude specific opcodes.

The architecture offers no hardware protection or page table scheme for program memory whatsoever. By using a relocating loader to finalize addresses immediately prior to program execution, programs can be scanned by the loader to ensure that all `CALL` and `JUMP` destinations are legitimate for the user runnin. These instructions are only available in fixed-address formats, so there is no means for evading this filtering. The drawback to this approach is that pointers to functions, `setjmp`-like schemes in C, and pointer-array-based enumerated cases are not available in this architecture. The simplicity and security gained in most cases will greatly outweigh these drawbacks.

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 OS loader is responsible for allowing specific program-external addresses to be called, and disallowing calls to other addresses. These external routines cannot not 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 can proxy additional privileges or memory access as needed.

Although program memory does not use a page table, it is divided into pages of 8 192 words. The reason is that the instruction pointer cannot count sequentially, but uses a linear feedback shift register (LFSR) to step through the 13 least significant bits. The reason is that fast counter ICs with parallel input and output are not manufactured for sale, so the program counter is implemented with XOR gates and D flip flops instead. This behavior by the instruction pointer is transparent to application programmers, and the assembler automatically inserts `JUMP` instructions where page boundaries need to be passed over.

Executable file formats for Dauug | 36 disregard the non-linear instruction pointer, and appear in listings as if the space is linear. The exception is these formats do contain the page boundary `JUMP` instructions. The operating system's loader assumes all responsibility for relocating instructions to their correct destinations and updating `CALL` and `JUMP` targets.

Instructions at addresses that end with 13 zeros are single-instruction infinite loops, because the LFSR cannot progress from zero. These locations can only be reached by `JUMP`, `CALL`, or system resets. But they can be left by branching; for example, a `JUMP` at address `360000‘0` to location `360001‘o` would escape the loop.

1.6 Organization of stack memory

Every user has an instruction pointer stack that is 511 addresses deep. The current instruction pointer is at the top of the stack, and is incremented immediately after the instruction is fetched. Later in the cycle if a subroutine is called, the subroutine's address is pushed onto the stack as the new instruction pointer, with the previous incremented pointer immediately beneath it. When the subroutine returns, the top address is removed, and execution resumes with the next instruction.

Because the architecture does not support stack-based recursion via `CALL`, the assembler is able to ensure that `CALL` loops cannot occur. The assembler will also determine the maximum call depth and ensure that the stack does not overflow. Thus there is no need, and no provision, for hardware detection of stack overflow. There is never any data on the stack, and the only access to the stack contents is via `CALL` and `RETURN`. The assembler

also disallows `RETURN` instructions within the topmost scope, so there is also no need, and no provision, for hardware detection of stack underflow. Note that even if the stack could overflow or underflow, other programs would not be affected. Thus an "evil assembler tool" is not a vector for stack-based system compromises.

1.7 Machine instruction format

The Dauug | 36 architecture implements four instruction formats, numbered after the number of fields used. All fields are a multiple of 9 bits in size, and the opcode field is always the most significant 9 bits.

Format I

opcode	ignored
bits 35–27	bits 26–0

Format I instructions include `CRF` and `NOP`. The Dauug | 36 hardware 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 parmeterized NOP.)

Format II

opcode	branch target in program memory
bits 35–27	bits 26–0

Format II is for `CALL` and `JUMP`. All branch targets are physical addresses: program memory is not protected by hardware and does not use a page table.

Note that program memory addresses cannot exceed 27 bits in this architecture. Most systems built for this architecture are not expected to anywhere close to this much RAM in the near term.

Format III

opcode	register	address in data memory
bits 35–27	bits 26–18	bits 17–0

Format III is for loading and storing registers from hardcoded addresses in data memory. This format may also be used for transferring in immediate values 18 bits at a time. More elegantly but less flexibly, immediate values may be treated as loads from a constant table in data memory. Note the address space for hardcoded addresses is ample at 256 Kiwords (1152 Kibytes).

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`.

Chapter 2

Structure of Dauug|36 Assembler Programs

2.1 Source code character set

Dauug | 36 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 2.1 summarizes some of those uses. It does not include characters that may appear in various operators and operator-like symbols, which appear separately as Table 5.1.

Table 2.1: Peculiar uses for various ASCII characters

Symbol	Use	See section
`0`–`9`	digits 0–9; identifier tails	2.3, 2.4
`a`–`z`	identifiers; digits 10–35	2.4, 2.3
`A`–`Z`	identifiers; digits 36–61	2.4, 2.3
`@ $`	digits 62 and 63	2.3
‘	numeral base if not 10	2.3
_	digit grouping; identifier tails	2.3, 2.4
`.  ’`	identifier tails	2.4
`;`	single-line comments	2.2
`( )`	enclosed comments	2.2
`< >`	type coercion	5.3
`-`	ranges in permutation notations	2.9
tab	same as space	
newline	end of statement	

2.2 Comments

Comments begin with a semicolon and extend to the end of the line. For example:

```
open_secret = 314159        ; ten thousand times pi
```

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 2.1 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 contain no more than one consecutive left parenthesis. Notably but perhaps less usefully, (and) can enclose blocks that contain two consecutive left parentheses, but not isolated ones. In any event, the runs of parentheses that begin and end inline comments can be as long as availabile 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:

```
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.

2.3 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 the value one.

Here are a few ways of writing 19 other than the usual:

```
1110111100111110001111111‘1    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 2.9.

Every number in this book, as well as every number in Dauug | 36 assembly programs, is in base 10 unless expressly stated otherwise in the format provided by this section. For convenience, a radix conversion chart appears as Table 2.2.

Table 2.2: 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

2.4 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, single quotes (ASCII 39), and periods (ASCII 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
```

2.5 Abbreviating keywords

A few frequently-used longer assembler keywords have been given abbreviations that can be used as alternatives. These abbreviations all end with a mandatory period. No spaces are allowed to set off the period; it is part of the abbreviation. On the other hand, a space is mandatory after the period, as it is not considered punctuation. Table 2.3 provides a list of these abbreviations.

Table 2.3: Keyword abbreviations

`c.`	`call`
`j.`	`jump`
`r.`	`return`
`s.`	`signed`
`u.`	`unsigned`

2.6 Undocumented keywords and instructions

This manual is oriented towards how to write Dauug | 36 programs, and not so much how the architecture works internally. Some keywords or instructions are not presently intended for writing programs, are esoteric, and may come and go as system features. In many cases, their documentation would require architectural explanations that are beyond the scope of this document. Table 2.4 lists these keywords and instructions.

Table 2.4: Undocumented keywords and instructions

Name	Comment for architecture maintainers
`eject`	something involving assembly listings
`emit`	bypass assembler with handwritten executable code
`ham3`	third instruction of popcount sequence
`un.a`	simple unary, alpha layer
`un.b`	simple unary, beta layer
`un.g`	simple unary, gamma layer
`stun.a`	stacked unary w/ specific theta, carry in, flags
`stun.b`	stacked unary w/ specific theta, carry in, flags
`stun.c`	stacked unary w/ specific theta, carry in, flags
`stun.d`	stacked unary w/ specific theta, carry in, flags
`stun.e`	stacked unary w/ specific theta, carry in, flags
`stun.f`	stacked unary w/ specific theta, carry in, flags
`stun.g`	stacked unary w/ specific theta, carry in, flags
`stun.h`	stacked unary w/ specific theta, carry in, flags
`stun.i`	stacked unary w/ specific theta, carry in, flags
`stun.j`	stacked unary w/ specific theta, carry in, flags

2.7 Declaring registers

Every running Dauug | 36 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.

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, and as many names as desired. Do not use any commas. Examples:

```
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 5.3 shows how to override a register's declared signedness for a specific need.

2.8 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
```

2.9 Permutation notations

Permutations of tribbles are notated in a 6-digit shorthand. Each digit shows the original position of each bit, numbered from the right starting with zero, in the permuted outcome. Thus if the permutation `321054` 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
; ...
tranpose  = in perm ztnhb5_ysmga4_xrlf93_wqke82_vpjd71_uoic60
left_rev  = in perm i-zh-0
right_rev = in perm z-i0-h
full_rev  = in perm 0-z
; ...
transpose = 0 txor in
```

Note: Using `PERM` here to compute `transpose` is for documentation purposes only. `PERM` is a macro that for most permutations expands to five instructions. The `TXOR` instruction as shown above can transpose a word in one instruction.

Chapter 3

Instruction reference

This chapter tabulates the Dauug | 36 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 chapter 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.

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 as the true result is not zero. Otherwise T is cleared, R is left unchanged, and Z is set of 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	
`a`	unsigned or signed	N	`a` < 0
`cw`	ignored	Z	`a` $= 0$
`c`	unsigned or signed	T	`c` 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 least 36 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	
a	unsigned or signed	N	a < 0
cw	ignored	Z	a = 0
c	unsigned or signed	T	c 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 least 36 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 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 to 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 and 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 ≥ 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-bit sum is 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 and wrap with 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 ≥ 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-bit sum is 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.

BO- — 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. Section 5.3 presents another synopsis.

BOUND Bound

```
bound i < lim
```

Register signedness		Generate interrupt if
`i`	ignored	`i` < 0 or `i` $\geq$ `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 GiB 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 = bzr 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. Section 5.3 presents another synopsis.

CALL Call

```
call subr          call < subr
call -t subr       call <= subr
call +t subr       call == subr
call -r subr       call != subr
call +r subr       call >= subr
                   call > subr
```

No registers used	No flags changed
11 opcodes total	

The `CALL` instructions pushes the current instruction pointer on the call stack, and transfer control to the subroutine with name `subr`. When a `RETURN` instruction is executed by the subroutine later, the instruction 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 dedicated SRAM devices. 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. The stack will never overflow, because the Dauug | 36 architecture does not support recursion via the stack.

Here is a sample:

```
again: nop
call again    ; infinite loop
```

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` 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:

`<`	N is set	`>`	N is clear
`<=`	N or Z is set	`>=`	N is clear or Z is set
`==`	Z is set	`!=`	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 `call = subr` syntactically means to copy a register named `subr` to a register named `call`.

`CALL` may have important pipelining consequences. Stay tuned.

CLO Count leading ones

```
c = clo b
```

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 macro only	Z	all result bits are zero

This builtin macro that expands to four CPU instructions. The number of leading one bits in `b` is counted and written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

CLZ Count leading zeros

```
c = clz b
```

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 macro only	Z	all result bits are zero

This builtin macro that expands to four CPU instructions. The number of leading zero bits in `b` is counted and written to `c`. N and Z are set as if the destination is a signed register. 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. wHere 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 = cto b

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 macro only	Z	all result bits are zero

This builtin macro that expands to four CPU instructions. The number of trailing one bits in `b` is counted and written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

CTZ Count trailing zeros

```
c = ctz b
```

Register signedness		Flag set if	
All	ignored	N	bit 35 of the result is set
	1 macro only	Z	all result bits are zero

This builtin macro that expands to four CPU instructions. The number of trailing zero bits in `b` is counted and written to `c`. N and Z are set as if the destination is a signed register. T and R do not change.

CX Check and extend

```
cw = cx f
```

Register signedness		Flag set if	
`f`	unsigned or signed	N	bit 35 of the result is set
`cw`	ignored	Z	all result bits are zero
	1 opcode only	T	`f` < 0 or `f` > 63
		R	T is set or R is already set

`CX` is used to prepare control words for several instructions, including `MH` and `ML`. This instruction verifies that $0 \leq$ `f` ≤ 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. Section 5.3 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. Section 5.3 presents another synopsis.

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. Section 5.3 presents another synopsis.

FABS Fast absolute value

```
c = fabs b
```

Register signedness		Flag set if	
b	signed	N	never; flag is cleared
c	unsigned or signed	Z	b = 0
	1 opcode only	T	b < −(2^35) or b ≥ 2^35
		R	T is set or R is already set

This is a single-instruction implementation of absolute value. Dauug | 36 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.

FO- 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. Section 5.3 presents another synopsis.

FZ - Find zero

```
c = fzl b       c = fzr 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. Section 5.3 presents another synopsis.

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. Section 5.3 presents another synopsis.

GZ - Grow zero

```
c = gzl b       c = gzr 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. Section 5.3 presents another synopsis.

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

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:

`<`	N is set	`>`	N is clear
`<=`	N or Z is set	`>=`	N is clear or Z is set
`==`	Z is set	`!=`	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. See Section 5.3.

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, implmemented as a two-instruction macro. N and Z are set as if the destination is a signed register. T and R do not change. Section 5.3 presents another synopsis.

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 = lzl b       c = lzr b
c = lzli b      c = 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, implmemented as a two-instruction macro. N and Z are set as if the destination is a signed register. T and R do not change. Section 5.3 presents another synopsis.

MAX Maximum

```
c = a 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 mod 64 ≠ 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 sigificant 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	`c` = 0

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 Section 5.3 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	`c` 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(iphertext) = p(laintext) mix k(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	c = 0

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. See Section 5.3.

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 folowing these instructions.

Because the hardware ignores the 27 operand bits of `NOP`, the Dauug | 36 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 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.

NOT 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. See Section 5.3.

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 0–31 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 $\lfloor \texttt{a} \div 256 = 12\,000 \rfloor$ and we want to force `a` mod $256 = 197 =$ `1100_0101‘b` without knowing it's current value and without changing $\lfloor \texttt{a} \div 256 \rfloor$, 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 `b` 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 Section 1.1. The tribbles of $\texttt{a}^\top$ 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 3.1 and follow the notation of Section 2.9. 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 ith 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 Dauug | 36 assembler provides this service automatically via the `PERM` macro.

PAIT Permute across and inside tribbles

```
c = a pait 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 Dauug | 36 is able to roll these two operations into this single instruction. For instance, `c = a pait 131313131313‘o` mirrors the bits of a 36-bit word (see Table 3.1).

PIT Permute inside tribbles

```
c = a pit 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 3.1 and follow the notation of Section 2.9. All 720 permutations can be reached via 2-instruction compositions of these 64.

Table 3.1: Tribble permutation operations

Oct.	Perm.	Derivation	Oct.	Perm.	Derivation
00	543210	identity	40	425130	count out of circle by 3s
01	054321	rotated identity	41	530241	count out of circle by 3s
02	105432	rotated identity	42	041352	count out of circle by 3s
03	210543	rotated identity	43	152403	count out of circle by 3s
04	321054	rotated identity	44	203514	count out of circle by 3s
05	432105	rotated identity	45	314025	count out of circle by 3s
06	501234	reflected 05	46	421530	count out by 3s backward
07	450123	reflected 04	47	352041	count out by 3s backward
10	345012	reflected 03	50	403152	count out by 3s backward
11	234501	reflected 02	51	514203	count out by 3s backward
12	123450	reflected 01	52	025314	count out by 3s backward
13	012345	reflected identity	53	130425	count out by 3s backward
14	543201	pair swap	54	453201	2-pair rotation
15	543012	pair swap	55	240513	2-pair rotation
16	540213	pair swap	56	013245	2-pair rotation
17	503214	pair swap	57	042315	2-pair rotation
20	043215	pair swap	60	534120	3-pair rotation
21	543120	pair swap	61	521430	3-pair rotation
22	541230	pair swap	62	512340	3-pair rotation
23	513240	pair swap	63	452301	3-pair rotation
24	143250	pair swap	64	315042	3-pair rotation
25	542310	pair swap	65	102354	3-pair rotation
26	523410	pair swap	66	034125	3-pair rotation
27	243510	pair swap	67	021435	3-pair rotation
30	534210	pair swap	70	435021	symmetric half rotation
31	345210	pair swap	71	354102	symmetric half rotation
32	453210	pair swap	72	215430	algorithmically selected
33	103254	movement in pairs	73	124530	algorithmically selected
34	325410	movement in pairs	74	342501	algorithmically selected
35	541032	movement in pairs	75	245031	algorithmically selected
36	531420	2x3 and 3x2 transposes	76	013452	algorithmically selected
37	524130	2x3 and 3x2 transposes	77	234015	algorithmically selected

POPC Popcount

c = popc b

Register signedness		Flag set if	
All	ignored	N	never; flag is cleared
		Z	b = 0

POPC is a builtin assembler macro that expands to three esoteric CPU instructions that aren't in this manual. 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	
`amt`	unsigned or signed	T	`amt` < 0 or `amt` > 63
`cw`	ignored	R	T is set or R is already set
	1 opcode only		

`PRL` prepares a control word for the `ROT` 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	
`amt`	unsigned or signed	T	`amt` < 0 or `amt` > 63
`cw`	ignored	R	T is set or R is already set
	1 opcode only		

This instruction prepares a control word for the `ROT` 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	
`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 `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.

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.

ROT Rotate

```
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. `ROT` will not change T or R, although a preceding `PRL` and `PRR` may.

RS Reverse subtract

```
c = a ~- 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.

RSB Reverse subtract with borrow

```
c = a ~-- 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 and wrap

```
<wrap> c = a ~- 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.

RSWB Reverse subtract and wrap with borrow

```
<wrap> c = a ~-- 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 = 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 = 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	$\texttt{a} - \texttt{b} < 0$
Right	unsigned or signed	Z	$\texttt{a} - \texttt{b} = 0$
Dest.	unsigned or signed	T	`c` cannot fit $\texttt{a} - \texttt{b}$
	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.

SB Subtract with borrow

`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 = 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 `b` 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 and wrap

```
<wrap> c = a - b
```

Register signedness		Flag set if	
Left	ignored	N	$\texttt{a} - \texttt{b} < 0$
Right	ignored	Z	$\texttt{a} - \texttt{b} = 0$
Dest.	ignored	T	$\texttt{a} - \texttt{b} < 0$
	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.

SWB Subtract and wrap with borrow

```
<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 $\mathtt{a}^\top$ using the tribbles of `cw` as swizzle function identifiers. These 64 swizzle functions appear in Table 3.2. See `TXOR` for a synopsis of the transposing operation from `a` to $\mathtt{a}^\top$. 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 3.2: 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 $\mathtt{b}^\top$ by relocating the ith bit of tribble j to the jth bit of tribble i for all $i, j \in \{0...5\}$. See also Section 1.1. The bitwise exclusive-OR of `a` with $\mathtt{b}^\top$ 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(laintext) = c(iphertext) xim k(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 !^ 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 ^ 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 Dauug | 36 architecture 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 2.5 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.[1]

```
; 36-bit LFSR
u. w0
;
w0 = lfsr w0
```

```
; 72-bit LFSR
u. w1 w0
;
w1 = stgr w1
w0 = rtgr w0
w1 = xpoly w1
```

```
; 108-bit LFSR
u. w2 w1 w0
;
w2 = stgr w2
w1 = rtgr w1
w0 = rtgr w0
w3 = xpoly w3
```

```
; 144-bit LFSR
u. w3 w2 w1 w0
;
w3 = stgr w3
w2 = rtgr w2
w1 = rtgr w1
w0 = rtgr w0
w3 = xpoly w3
```

[1] Verify this with Dieharder before publication. There are eight untested cases.

Chapter 4

Glossary

- Define “primary storage.”
- Define “signedness.”
- Define “tribble.”
- Define “overflow” (can be very postive OR very negative).
- Explain ”ordinary assignment.”
- opcode
- operation
- macro, Kibyte, Kiword, Mibyte, Miword, virtual machine

Chapter 5

Unfinished business

5.1 To use later

Table 5.1: Operators and operator-like symbols

—

—

5.2 Bugs to fix and quirks to document

- ASCII nulls in source cause undefined results.
- Inline comments break tokens.
- Add hierarchy: when to use A, AC, AW, AWC
- Need asm test of multi-precision add, subtract w/ all signednesses
- Test corner behavior of Z: Add/subtract with wrap and borrow
- Explain how instructions are implemented; e.g., NAS is actually A.
- Explain why no rotations > 63 and why no negative rotations.
- Jump destinations and calls are hard-coded. For a reason.
- The assembler does not implement the underscore grouping symbol for numbers.

5.3 Sections to write

- SAY is for VM diagnostics and can output numbers in base 10, strings, and characters.

```
u. x letter

x = 12345
letter = 65
say "The value of x is " x ", the numeral 10 is " 10 ", and " (
) the first letter of the alphabet is " #letter "."
sayn "There is no line break right here, "
say "but there is one at the end of the line."
```

- KB gets an ASCII character from stdin. It's not a nonary function, but a keyword in the first token position.
- EMIT for forcible assembly.
- NOW delineates scoped sections. It's not perfectly convenient in terms of positional parameters or brevity of notation, but it's rather optimal for execution speed.

```
now shortmul:
    unsigned big small t result
    t = cx small
    result = big mh t
    t = big ml t
    result = result + t
    return

now main:
    shortmul:big = 123456
    shortmul:small = 7
    call shortmul
    say "123456 * 7 = " shortmul:result

now alternative:
    ; : without prefix uses last lexical scope
    shortmul:big = 123456
    :small = 7
    call
    say "123456 * 7 = " :result
```

- High-level syntax: keywords, instructions, case sensitivity, scope, name of main scope, etc.
- Keyword reference
- labels
- tokenization
- Floating point formats. A binary36 is defined, single precision, base 2, 27 sig bits, 8.13 decimal digits, 9 exponent bits, 76.76 decimal E max, exponent bias 255, E min -254, E max 255, but no implementation.
- IF that works like CMP and uses T as a "true" flag. `if (a = b and c = d) or (e = f and g = h)` can have the result in T in four (!) instructions. If we let the assembler swap operands sometimes, we can do this in 20 opcodes, and allow a 3-deep stack via T, N, and Z (with crazy abuses). The opcodees are {`< <= == !=`} × {plain, and, or, and-or, or-and}. It doesn't short-circuit, but it screams in the pipeline.

  We could go pretty crazy with implementation, using De Morgan's theorem to get NOT, NAND, NOR, LANR, LONR, RANL, RONL going. XOR and XNOR come at higher cost: either a CPU cycle, or potentially a lot more opcodes. We only need XOR, because we can De Morgan out XNOR. Full generality, stack depth 3, and maximum speed would cost $4(3^0 + 3^1 + 3^2 + 3^3) = 160$ opcodes. That's too many. Two deep with full generality would be 52 opcodes. That's over 10% of our opcode universe.

  Another weakness to this proposal is that bitwise boolean might be desired. `if a == b and c & 5` or that kind of thing. We don't have that.

- IMM (register) number
  This is not the desired final syntax. Number can be up to 0777777.

5.4 Immediate tasks and their order

- instruction format (machine)
- instruction format (syntax)
- decide on RETURN payload
- write RETURN
- decide syntax for scope
- write about how scope works and how registers are allocated
- finish up CALL, no conditionals please
-
-

Index