BINAR™ Opcode Reference

Preliminary - Version 0.0

March 1, 1990

Value	Opcode	Value	Opcode	Value	Opcode	Value	Opcode
0x00	NOP	0x20	?branch	0x3F	OVER	0x6A	CONFIG!
0x01	<interrupt>	0x21	@	0x40	OVER_QUICK	0x6B	ENABLE
0x03	<trap>	0x22	AND	0x41	R>	0x6C	DISABLE
0x04	<cold>	0x23	AND_QUICK	0x42	R>_QUICK	0x6D	1
0x05	HALT	0x24	ASR	0x43	ROT	0x6E	1+
0x06	NOP_QUICK	0x25	branch	0x44	RTI	0x6F	1-
0x07	0	0x26	C!	0x45	SWAP	0x70	2*
0x08	0<	0x27	C@	0x46	SWAP_QUICK	0x71	2*_QUICK
0x09	NOT	0x28	B@	0x47	U>	0x72	4+
0x0A	0=	0x29	D!	0x48	UM*	0x73	+!
0x0B	2/	0x2A	D+	0x51	UM/MOD	0x74	<do>
0x0C	!	0x2B	D>R	0x56	XOR	0x75	DDROP
0x0D	SP!	0x2C	D@	0x57	XOR_QUICK	0x76	DDUP
0x0E	SP@	0x2D	DASR	0x58	ASIC!	0x77	NIP
0x0F	RP!	0x2E	docon	0x59	LIT_ASIC!	0x78	NIP_QUICK
0x10	RP@	0x2F	dovar	0x5A	ASIC@	0x79	TUCK
0x11	+	0x30	DR>	0x5B	LIT_ASIC@	0x7A	=
0x12	+_QUICK	0x31	DROP	0x5C	DBASE!	0x7B	<>
0x13	-1	0x32	DROP_QUICK	0x5D	DBASE@	0x7C	?DUP
0x13	TRUE	0x33	DUP	0x5E	SPLIM!	0x7D	ABS
0x14	-	0x34	DUP_QUICK	0x5F	SPLIM@	0x7E	DLSL
0x15	<lit>	0x35	EXECUTE	0x60	FP!	0x7F	DOVER
0x16	<short>	0x36	H!	0x61	FP@	0x80	DSWAP
0x17	LIT_+	0x37	H@	0x62	MRAM!	0x81	DROT
0x18	<short>_QUICK	0x38	R@	0x63	MRAM@	0x83	S>D
0x19	LIT_+_QUICK	0x39	R@_QUICK	0x64	RPLIM!	0x84	NEGATE
0x1A	<loop>	0x3A	I_+	0x65	RPLIM@	0x85	DNEGATE
0x1C	PICK	0x3B	I'	0x66	SBASE!	0x86	D-
0x1D	ROLL	0x3C	J	0x67	SBASE@	0x87	/STRING
0x1E	>	0x3D	OR	0x68	WAIT	0x8A	CMOVE
0x1F	>R	0x3E	OR_QUICK	0x69	CONFIG@	0x8B	MOVE

Value	Opcode
0x8C	FILL
0x8D	WFILL
0x8E	COUNT
0x8F	M+
0x90	0=_NOT
0x91	0>
0x92	2_PICK
0x93	3_ROLL
0x94	-ROT
0x95	4-_!
0x96	4-_@
0x97	?NEGATE
0x98	not?branch
0x99	=branch
0x9A	@_@
0x9B	@_!
0x9C	@_+
0x9D	@_LIT+
0x9E	ADC
0x9F	AND!
0xA0	OR!
0xA1	C+!
0xA2	C_OR!
0xA3	?DNEGATE
0xA5	D=
0xA6	DLSR
0xA7	LIT_NIP
0xA8	DUP_0<
0xA9	DUP_@
0xAA	SWAP_OVER_!
0xAD	LIT_AND

0xAC LIT-+ fast

Value	Opcode
0xAE	LIT_AND_QUICK
0xAF	LIT_OR
0xB0	LIT_OR_QUICK
0xB1	LIT_!
0xB2	LIT_@
0xB3	LSLN
0xB4	LIT_LSLN
0xB5	LSR
0xB6	LSRN
0xB7	LIT_LSRN
0xB8	OVER_!
0xB9	OVER_@
0xBA	OVER_+
0xBB	RLC
0xBC	RRC
0xBD	R@_!
0xBE	R@_@
0xBF	R>_!
0xC0	R>_@
0xC1	LIT_R+!
0xC2	LIT_ROT_+
0xC3	BIT-CLEAR
0xC4	+_FLAGS
0xC5	DS@
0xC6	DS!
0xC7	TEST_UNDER_MASK
0xC8	FETCH_AND_ADD
0xC9	LIT_FETCH_AND_ADD

Value	Opcode
0xCA	TEST_AND_SET
0xCC	LIT_LOC_@
0xCD	LIT_LOC_C@
0xCE	LIT_LOC_B@
0xCF	LIT_LOC_@_+
0xD0	LIT_LOC_!
0xD1	LIT_LOC_C!
0xD2	LIT_LOC_@_@
0xD3	LIT_LOC_@_!
0xD4	LIT_LOC_+!
0xD5	SBASE+_@
0xD6	SBASE+_!
0xD7	DBASE+_@
0xD8	DBASE+_!
0xD9	UNORMALIZE
0xDA	UDNORMALIZE
0xDB	LOAD_DS
0xDC	LOAD_RS
0xDD	STORE_DS
0xDE	STORE_RS
0xE0	?EXIT
0xE1	<next>
0xE2	I'_!
0xE3	I'_@
0xE4	R@_4-_!
0xE5	I'_4-_!
0xE6	SWAP_!
0xE7	SWAP_-
0xE8	FP+!
0xE9	LIT_FP+!
0xEA	SBASE+!

Value	Opcode
0xEB	DBASE+!
0xEC	FRAME_PUSH
0xED	FRAME_POP
0xEE	LIT_@_inc
0xEF	LIT_C@_inc
0xF0	LIT_!_inc
0xF1	LIT_C!_inc
0xF2	LEAVE
0xF3	+_@
0xF4	LIT_+_@
0xF5	+_C@
0xF6	LIT_+_C@
0xF7	SWAP_-_@
0xF8	SWAP_-_C@
0xF9	+_!
0xFA	+_C!
0xFB	LIT_+_!
0xFC	LIT_+_C!
0xFD	SWAP_-_!
0xFE	SWAP_-_C!
0xFF	-3_ROLL

This is a preliminary document. While reasonable effort has been made to ensure its accuracy, it probably has minor errors, and is subject to change.

Please report any errors or omissions so that they may be corrected in the final version.

This document lists the BINAR instruction set as used with the target-resident Forth compiler. The opcodes are designed to represent the operation of the machine conveniently for Forth programmers. The "assembly language" instruction set for use with other language will use different naming conventions and syntax.

Notation:

Addresses are represented by "addr".

Signed 32-bit integers are represented by "n". 32-bit integers treated as unsigned quantities are represented by "un".

Signed 64-bit integers are represented by "d". They are placed on the stack as a pair of 32-bit values, with the more significant 32 bits on the top of the stack with respect to the least significant 32 bits. 64-bit integers treated as unsigned quantities are represented by "ud".

Signed 16-bit integers are represented by "h". 16-bit integers treated as unsigned quantities are represented by "uh".

Signed 8-bit integers are represented by "b". 8-bit integers treated as unsigned quantities (*e.g.* characters) are represented by "c".

Logical flags are represented by "flag". Flags input to an opcode are true if non-zero. Flags output by opcodes are 0 if false, -1 if true.

All quantities are 32-bit signed integers unless otherwise specified. Bits are numbered from lowest to highest, with bit 0 being the lowest order bit, and bit 31 being the highest order bit of a 32-bit quantity (the sign bit).

Each opcode starts at the top of a new page. The mnemonic is the Forth notation for the opcode. Below the header is a stack picture which shows the inputs and outputs of the opcode. As an example, the word with the following stack picture:

(n1 b2 addr3 → n4 un5)

takes three inputs on the data stack: a signed 32-bit integer n1, a signed 8-bit integer b2, and an address addr3, with addr3 on the top of the stack, b2 as the second element on the stack, and n1 as the third element on the stack. The word returns two quantities on the data stack: a signed 32-bit integer n4 as the second element on the stack, and an unsigned 32-bit integer un5 on the top of the stack. In words which affect the return stack, the notation "RS(→)" is used in a similar manner. The notation "Immediate" to the right of a stack notation means that that stack picture applies to the immediate operand mode of the instruction, using the literal field to supply a constant value.

Some opcodes require address or literal fields for proper operation. When the encoding notation includes "/ lit", then the short or long literal instruction fields must be used with the opcode, mandating use of the JNEXT, EXIT, or 2OPS instruction formats. Literal fields are used as sign-extended integers. When the encoding notation includes "/ call", then the CALL instruction format must be used, and bits 2-22 of the word aligned address of the call target (typically a conditional branch target) must be placed in bits 2-22 of the CALL instruction. Encodings that are annotated "(2OPS format)" may *only* be used in the

2OPS format, but provide single-cycle instruction execution.

Some opcodes have variants that allow the use of either a literal field or a data stack value for operation. In these cases, the description is written for the variant that uses the data stack, with appropriate annotation for the behavior of the literal field.

Words that have an effect on the Return Stack should only be used in JNEXT or 2OPS instruction formats unless otherwise noted. Return Stack manipulation words can interfere with the operation of subroutine calls and returns in subtle ways that are not obvious to the casual user.

Alignment:

All word addresses are forced to be word aligned by ignoring the bottom two bits (*i.e.*, bits 0 and 1) and substituting zeros.

All half-word addresses are forced to be half-word aligned by ignoring the bottom bit (*i.e.*, bit 0) and substituting a zero.

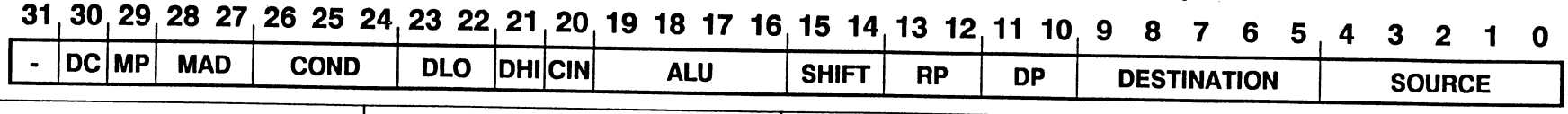
Cautions:

The instruction set and its implementation can and *will* be changed somewhat on the next version of the silicon. Therefore, code that exploits the particular numeric value of an opcode or any unusual properties of its interpretation must be avoided. Code should be written so that replacing an opcode with an equivalent subroutine call will have no adverse effects. This is extremely easy to do on a stack machine, and so should present no real problem.

The "conventional" mnemonics are very preliminary, and are subject to change without notice during the implementation of the RTX assembler. The "Forth" mnemonics are reasonably stable, and track words available in the target-resident Forth compiler being developed

by Harris.

BINAR QUICK REFERENCE • Harris Semiconductor Proprietary Information



DECODE CONTROL (30):
0 nop (default)
1 DECODE

INCREMENT MPC (29):
0 nop[MPC] (default)
1 INC[MPC]

MICRO-ADDRESS (27-28):
0 JMP=00x
1 JMP=01x
2 JMP=10x
3 JMP=11x

CONDITION CODE (24-26):
0 JMP=xx0 always 0
1 JMP=xxC Carry out
2 JMP=xxZ Zero
3 JMP=xxS Sign
4 JMP=xxL Lowest bit of DLO
5 JMP=xxV oVerflow
6 JMP=xxP Pending interrupt
7 JMP=xx1 always 1

DLO CONTROL (22-23):
0 nop[DLO] (default)
1 SR[DLO]
2 SL[DLO]
3 DEST=DLO

DHI (21):
0 DHI[0] (default)
1 DHI[1]

CIN (20):
0 CIN=0 (default)
1 CIN=1

ALU FUNCTION (16-19):

	CIN=0	CIN=1
0	A (default)	A
1	A or B	A or B
2	A + A	A + A + 1
3	-1	-1
4	A and B	A and B
5	B	B
6	A + B	A + B + 1
7	A + 0	A + 1
8	A - 1	A - B
9	A xor B	A xor B
10	not B	not B
11	A nand B	A nand B
12	0	0
13	A - B - 1	unused
14	A nor B	A nor B
15	not A	not A

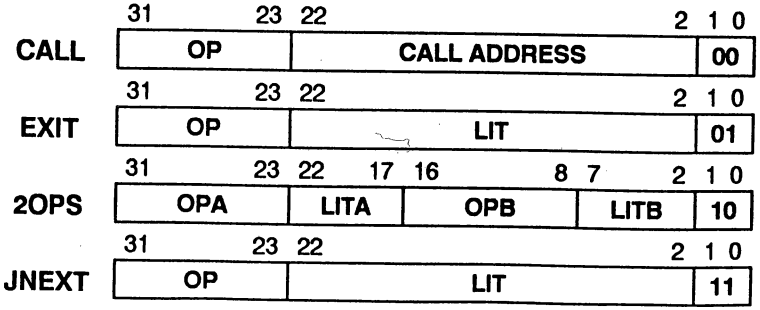
ALU SHIFT (14-15):
0 PASS[ALU] (default)
1 SR[ALU]
2 SL[ALU]
3 FPU (pass FPU output)

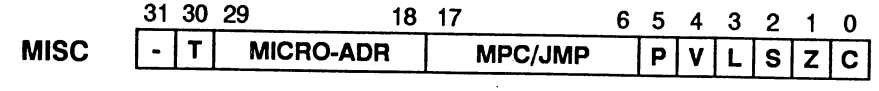
RP CONTROL (12-13):
0 DEC[RP]
1 INC[RP]
2 nop[RP] (default)
3 DEST=RP

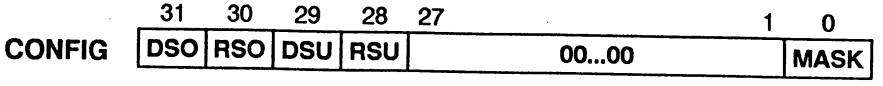
DP CONTROL (10-11):
0 DEC[DP]
1 INC[DP]
2 nop[DP] (default)
3 DEST=DP

DESTINATION (5-9):
0 none (default)
1 DEST=HOST
2 DEST=DS
3 DEST=DP-LIMIT
4 DEST=RS
5 DEST=RP-LIMIT
6 DEST=DHI[0]
7 DEST=DHI[1]
8 DEST=MRAM
9 DEST=MICRO-ADR
10 DEST=DS-FROM-DHI
11
12 DEST=SBASE
13 DEST=DBASE
14 DEST=FP
15 RAM-RMW
16 RAM-C@
17 RAM-W@
18 DEST=CONFIG
19 ASIC-@
20 DEST=RAM-C!
21 DEST=RAM-W!
22 DEST=RAM-!
23 DEST=ASIC-!
24
25 LATCH-INSTRUCTION
26 CYCLE-RAM
27 ADDR=BUS-4(CYCLE)
28 ADDR=BUS+0(CYCLE)
29 ADDR=BUS+SBASE(CYCLE)
30 ADDR=BUS+DBASE(CYCLE)
31 ADDR=BUS+FP(CYCLE)

SOURCE (0-4):
0 SOURCE=HOST
1 SOURCE=LIT
2 SOURCE=PAGE/NAR/CTL
3 SOURCE=ADDR
4 SOURCE=MRAM
5 SOURCE=MROM
6 SOURCE=SBASE
7 SOURCE=DBASE
8 SOURCE=FP
9 SOURCE=MISC
10 SOURCE=RP
11 SOURCE=DP
12 SOURCE=RS
13 SOURCE=RETURN-SAVE
14 SOURCE=CONFIG
15 SOURCE=I-LATCH
16 SOURCE=DHI[0]
17 SOURCE=DHI[1]
18 SOURCE=RP-LIMIT
19 SOURCE=DP-LIMIT
20 SOURCE=DLO
21 SOURCE=-1
22 SOURCE=0
23 SOURCE=4
24 SOURCE=unused-24
25 SOURCE=DS
26 MULTIPLY-STEP
27 DIVIDE
28 SOURCE=RD
29 SOURCE=RD-SIGNED
30 SOURCE=unused-30
31 SOURCE=unused-31







!

Store

(n1 addr2 →)

(n1 →) Immediate

Encoding:

0x0C	4 cycles
0xB1 / lit	4 cycles

Operation:

Store n1 at address addr2. The store immediate variant uses the literal field to provide the value of addr2.

Implementation:

```
opcode: !
0: SOURCE=DHI
          ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS   INC[DP]    DEST=RAM-! ;;
2: ;;
3: SOURCE=DS   INC[DP]    ALU=B
          DECODE ;;

opcode: LIT_!
0: SOURCE=LIT       ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI       DEST=RAM-! ;;
2: ;;
3: SOURCE=DS   INC[DP]  ALU=B     DECODE ;;
```


!_INC

Incrementing Store

(n1 addr2 → addr3) Immediate

Encoding:

0xF0 / lit	4 cycles

Operation:

Store value n1 at location addr2, then add the compiled instruction literal field to addr2, leaving addr3.

Implementation:

```
opcode: !_INC
0: SOURCE=DHI
     ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS   INC[DP]     DEST=RAM-! ;;
2: ;;
3: SOURCE=LIT   ALU=A+B   DECODE ;;
```

+

Add

(n1 n2 → n3)

(n1 → n3) Immediate

Encoding:

0x11	2 cycles
0x12	1 cycle (2OPS format)
0x17 / lit	2 cycles
0x19 / lit	1 cycle (2OPS format)

Operation:

Add n1 and n2, giving n3= n1+n2. In the immediate variant, the value n2 is provided by the literal field.

Implementation:

```
opcode: +
0: SOURCE=DS  INC[DP]  ALU=A+B ;;
1: DECODE ;;

opcode: +_QUICK
0: SOURCE=DS  INC[DP]  ALU=A+B  DECODE ;;

opcode: LIT_+
0: SOURCE=LIT  ALU=A+B ;;
1: DECODE ;;

opcode: LIT_+_QUICK
0: SOURCE=LIT  ALU=A+B   DECODE ;;
```


+!

Add To Memory

(n1 addr2 →)

Encoding:

0x73	7 cycles

Operation:

Performs an atomic increment of the word at addr2 by value n1.

Implementation:

```
opcode: +!
0:  SOURCE=DHI   ADDR=BUS+0(CYCLE) ;;
1:  RAM-RMW ;;
2:  SOURCE=DS INC[DP]    ALU=B ;;
3:  SOURCE=RD    ALU=A+B   CYCLE-RAM ;;
4:  SOURCE=DHI   DEST=RAM-! ;;
5:  SOURCE=DS INC[DP]    ALU=B ;;
6:  DECODE ;;
```

+_!

Indexed Store

(n1 n2 addr3 →)

(n1 addr3 →) Immediate

Encoding:

0xF9	5 cycles
0xFB / lit	5 cycles

Operation:

Add the offset n2 to addr3, then store the value n1 in that same memory location. In the immediate variant, the value n2 is taken from the literal field.

Implementation:

```
opcode: +_!
0: SOURCE=DS  INC[DP]   ALU=A+B ;;
1: SOURCE=DHI        ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS  INC[DP]    DEST=RAM-! ;;
3: ;;
4: SOURCE=DS INC[DP]  ALU=B     DECODE ;;

opcode: LIT+_!
0: SOURCE=LIT           ALU=A+B ;;
1: SOURCE=DHI        ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS  INC[DP]    DEST=RAM-! ;;
3: ;;
4: SOURCE=DS INC[DP]  ALU=B   DECODE ;;
```


+_@

Indexed Load

(n1 addr2 → n3)

(addr2 → n3) Immediate

Encoding:

0xF3	5 cycles
0xF4 / lit	5 cycles

Operation:

Add the offset n1 to addr2, then fetch the value n3 from that memory location. In the immediate variant, the value n1 is provided by the literal field.

Implementation:

```
opcode: +_@
0:  SOURCE=DS  INC[DP]   ALU=A+B ;;
1:  SOURCE=DHI   ADDR=BUS+0(CYCLE) ;;
2: ;;
3: ;;
4:  SOURCE=RD   ALU=B    DECODE ;;

opcode: LIT+_@
0: SOURCE=LIT   ALU=A+B ;;
1: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
2: ;;
3: ;;
4: SOURCE=RD   ALU=B    DECODE ;;
```

+_C!

Indexed Store Character

(b1 n2 addr3 →)

(b1 addr3 →) Immediate

Encoding:

0xFA	5 cycles
0xFC / lit	5 cycles

Operation:

Add the offset n2 to addr3, then store the value b1 in that memory location. The highest 24 bits of the stack value for b1 are ignored. In the immediate variant, the value n2 is provided by the literal field.

Implementation:

```
opcode: +_C!
0: SOURCE=DS INC[DP]    ALU=A+B ;;
1: SOURCE=DHI         ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS INC[DP]    DEST=RAM-C! ;;
3: ;;
4: SOURCE=D  INC[DP]    ALU=B     DECODE ;;

opcode: LIT+_C!
0: SOURCE=LIT             ALU=A+B ;;
1: SOURCE=DHI         ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS  INC[DP]     DEST=RAM-C! ;;
3: ;;
4: SOURCE=DS  INC[DP]    ALU=B    DECODE ;;
```


+_C@

Indexed Load Character

(n1 addr2 → c3)

(addr2 → c3) Immediate

Encoding:

0xF5	5 cycles
0xF6 / lit	5 cycles

Operation:

Add the offset n1 to addr2, then fetch the value c3 from that memory location. c3 is zero-extended to form a 32-bit stack word. In the immediate variant, the value n1 is provided by the literal field.

Implementation:

```
opcode: +_C@
0: SOURCE=DS INC[DP]    ALU=A+B ;;
1: SOURCE=DHI    ADDR=BUS+0(CYCLE) ;;
2: RAM-C@ ;;
3: ;;
4: SOURCE=RD     ALU=B     DECODE ;;

opcode: LIT+_C@
0: SOURCE=LIT    ALU=A+B ;;
1: SOURCE=DHI    ADDR=BUS+0(CYCLE)   ;;
2: RAM-C@ ;;
3: ;;
4: SOURCE=RD    ALU=B     DECODE ;;
```

+_FLAGS

Add With Flags

(n1 n2 → n3 MISC)

Encoding:

0xC4 2 cycles

Operation:

Perform an addition of n1 and n2 giving n3=n1+n2. Additionally, return the value of the MISC bus source, the lowest bits of which contain the ALU flags resulting from the addition (bit 0 = C, bit 1 = Z, bit 2 = S, bit 4 = V).

Implementation:

```
opcode: +_FLAGS
0: SOURCE=DS     ALU=A+B ;;
1: DS-FROM-DHI
   SOURCE=MISC   ALU=B    DECODE ;;
```


-

Subtract

(n1 n2 → n3)

Encoding:

0x14 2 cycles

Operation:

Subtract n2 from n1, giving n3= n1-n2.

Implementation:

```
opcode: -
0: SOURCE=DS       ALU=B     DS-FROM-DHI ;;
1: SOURCE=DS INC[DP]  ALU=A-B  DECODE ;;
```

-3_ROLL

Put Top As Fourth

(n1 n2 n3 n4 → n4 n1 n2 n3)

Encoding:

0xFF 5 cycles

Operation:

Perform a "backwards ROLL" of the data stack involving 4 elements, moving the top stack element into the fourth-from-top position. This is an inverse operation to 3_ROLL.

Implementation:

```
opcode: -3_ROLL
0: SOURCE=DS INC[DP]   DEST=DLO ;;
1: SOURCE=DS INC[DP]  DHI[1]  ALU=B ;;
2: DHI[0]  DS-FROM-DHI
   SOURCE=DS  ALU=B ;;
3: DHI[0]  DEC[DP]  DS-FROM-DHI
   SOURCE=DLO   ALU=B ;;
4: DHI[1]  DEC[DP]  DS-FROM-DHI
     DECODE ;;
```


-ROT

Put Top as Third

(n1 n2 n3 → n3 n1 n2)

Encoding:

0x94 3 cycles

Operation:

Perform a "backwards ROLL" of the data stack involving 3 elements, moving the top stack element into the third-from-top position. This is an inverse operation to ROT.

Implementation:

```
opcode: -ROT
0: SOURCE=DS  INC[DP]   DHI[1]  ALU=B ;;
1: DHI[0] DS-FROM-DHI
   SOURCE=DS  ALU=B ;;
2: DEC[DP]  DHI[0]  DS-FROM-DHI
   SOURCE=DHI[1]  ALU=B          DECODE ;;
```

/STRING

Chop String

(addr1 cnt2 n3 → addr4 cnt5)

Encoding:

0x87 4 cycles

Operation:

Zen-Forth support word for truncating the left-most n3 characters of a string represented by addr1 and cnt2. Addr4 is computed as addr1+n3. Cnt5 is computed as cnt2-n3.

Implementation:

```
opcode: /STRING
0: DHI[0]  ALU=notA  DEST=DHI[1] ;;
1: SOURCE=DS INC[DP]
       DHI[1]  ALU=A+B+1 ;;
2: SOURCE=DS  ALU=A+B ;;
3: DHI[0]  DS-FROM-DHI
   SOURCE=DHI[1]  ALU=B     DECODE ;;
```


0<

Test For Less Than Zero

(n1 → flag2)

Encoding:

0x08 2 cycles

Operation:

Flag2 is true if n1 is less than zero (*i.e.*, if sign bit is set).

Implementation:

```
opcode: 0<
0: JMP=01S ;;

2: ALU=0   DECODE ;;
3: ALU=-1  DECODE ;;
```

0=

Test For Equal To Zero

(n1 → flag2)

Encoding:

0x0A 2 cycles

Operation:

Flag2 is true if n1 equals zero.

Implementation:

```
opcode: 0=
0: JMP=01Z ;;

2: ALU=0   DECODE ;;
3: ALU=-1  DECODE ;;
```


0=_NOT

Test For Not Equal To Zero

(n1 → flag2)

Encoding:

0x90 2 cycles

Operation:

Flag2 is true if n1 is not equal to zero. Useful for forcing a clean flag value of -1 or 0 if n1 is a "dirty" flag value.

Implementation:

```
opcode: 0=_NOT
0: JMP=01Z ;;

2: ALU=-1  DECODE ;;
3: ALU=0   DECODE ;;
```

0>

Test For Greater Than Zero

(n1 → flag2)

Encoding:

0x91 3 cycles

Operation:

Flag2 is true if n1 is greater than zero. This involves both a check of the sign bit and a test for exactly equal to zero.

Implementation:

```
opcode: 0>
0: JMP=01Z ;;

2: ( <>0 )  JMP=10S ;;
3: ( =0 )   JMP=101 ;;

4: ( >0 )   ALU=-1  DECODE ;;
5: ( <=0 )   ALU=0   DECODE ;;
```


1+

Increment

(n1 → n2)

Encoding:

0x6E 2 cycles

Operation:

Increments n1, producing n2 = n1+1

Implementation:

```
opcode: 1+
0:  ALU=A+1 ;;
1:  DECODE ;;
```

1-

Decrement

(n1 → n2)

Encoding:

0x6F 2 cycles

Operation:

Decrements n1, producing n2 = n1 - 1

Implementation:

```
opcode: 1-
0:  ALU=A-1 ;;
1:  DECODE ;;
```


2*

Logical Shift Left

(n1 → n2)

Encoding:

0x70	2 cycles
0x71	1 cycle (2OPS format)

Operation:

Shifts n1 left by 1 bit, producing n2. This is equivalent to multiplication by 2.

Implementation:

```
opcode: 2*
0:  ALU=A+A ;;
1:  DECODE ;;

opcode: 2*_QUICK
0:  ALU=A+A   DECODE ;;
```

2/

Divide by 2

(n1 → n2)

Encoding:

0x0B 3 cycles

Operation:

Performs truncated signed division by two. This is true division, not simply a shift right operation.

Implementation:

```
opcode: 2/
0: ALU=notA  DEST=DHI[1]  JMP=01S ;;

( Input non-negative )
2: ALU=A  CIN=0  SR[ALU]  JMP=110 ;;
6: DECODE ;;

( Input negative )
3: JMP=10Z ;;   ( test for -1 )

( Input was not -1  )
4: ALU=A+1  SR[ALU] DECODE ;;
( Input was -1 )
5: ALU=0            DECODE ;;
```


2_PICK

Push 3rd Element On Stack

(n1 n2 n3 → n1 n2 n3 n1)

Encoding:

0x92 3 cycles

Operation:

Copies the third element on the stack to the top of stack.

Implementation:

```
opcode: 2_PICK
0: INC[DP] ;;
1: SOURCE=DS    DEC[DP]  DHI[1]  ALU=B ;;
2: DEC[DP]  DS-FROM-DHI  DHI[0]
   SOURCE=DHI[1]  ALU=B     DECODE ;;
```

3_ROLL

Get Fourth Element On Stack

(n1 n2 n3 n4 → n2 n3 n4 n1)

Encoding:

0x93 5 cycles

Operation:

Moves the fourth element on the stack to the top of stack.

Implementation:

```
opcode: 3_ROLL
0: SOURCE=DS INC[DP]   DEST=DLO ;;
1: SOURCE=DS INC[DP]  DHI[1]  ALU=B ;;
2: DHI[1]  DS-FROM-DHI
   SOURCE=DS  ALU=B ;;
3: SOURCE=DLO  DEC[DP]  DEST=DS ;;
4: DEC[DP]  DHI[0] DS-FROM-DHI
   SOURCE=DHI[1]   ALU=B     DECODE ;;
```


4+

Increment By 4

(n1 → n2)

Encoding:

0x72 2 cycles

Operation:

Adds 4 to n1, giving n2 = n1+4.

Implementation:

```
opcode: 4+
0:  SOURCE=4   ALU=A+B ;;
1:  DECODE ;;
```

4-_!

Store At Address-4

(n1 addr2 →)

Encoding:

0x95 4 cycles

Operation:

Stores n1 at address "addr2-4".

Implementation:

```
opcode: 4-_!
0: SOURCE=DHI        ADDR=BUS-4(CYCLE) ;;
1: SOURCE=DS  INC[DP]    DEST=RAM-! ;;
2: ;;
3: SOURCE=DS  INC[DP]   ALU=B   DECODE ;;
```


4-_@

Load From Address-4

(addr1 → n2)

Encoding:

0x96 4 cycles

Operation:

Fetches value n1 from address "addr1-4".

Implementation:

```
opcode: 4-_@
0:  SOURCE=DHI   ADDR=BUS-4(CYCLE) ;;
1: ;;
2: ;;
3:  SOURCE=RD   ALU=B    DECODE ;;
```

< +loop >

Loop, Count By N

(n1 →)
RS(limit1 index2 → limit1 index3
... →)

Encoding:

0x1F0 / call 7/10 cycles

Operation:

Adds n1 to the loop index index2, then checks to see if the new index value index3 exceeds the loop limit limit1. If n1 is non-negative, then "exceeds" is defined as index3 > limit1. If n1 is negative, then "exceeds" is defined as index3 <= limit1. If index3 exceeds limit1, the index and limit are popped off the return stack and execution continues in-line. If index3 does not exceed limit1, the new index value is stored on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. The opcode takes 7 clock cycles if the branch is taken, 10 clock cycles if not taken. This opcode assumes that index3 and limit1 will never differ in value more than 0x80000000. In other words, it does not check for overflow when doing the comparison.

Notes:

This opcode is broken in the BINAR ROM. The Implementation given below is correct, and is loaded as an MRAM opcode during Forth cold-start.

Implementation:

```
opcode: <+loop>
0: SOURCE=RETURN-SAVE   DEST=DLO ;;
1: DEC[DP]  SOURCE=DHI DEST=DS
                        INC[RP] ;;
2:  SOURCE=RS   INC[RP]   ALU=A+B ;;
3:  SOURCE=RS   ALU=A-B   DEST=DHI[1] ;;
( If result sign same as incremnt, done )
4: SOURCE=DS INC[DP]  DHI[1] ALU=AxorB ;;
5: SOURCE=DHI    DEC[RP] DEST=RS
                 JMP=11S ;;

( Signs different, continue loop )
7:  SOURCE=DS  INC[DP]   ALU=B  DECODE ;;

( Signs match, abort loop )
6:  SOURCE=DLO   ADDR=BUS+0(CYCLE)
                 INC[MPC]   JMP=101 ;;
NEXT opcode
5:  INC[RP] ;;
6:  INC[RP]           LATCH-INSTRUCTION ;;
7:  SOURCE=DS  INC[DP]   ALU=B  DECODE ;;
```

< >

Test For Not Equal

(n1 n2 → flag3)

Encoding:

0x7B 3 cycles

Operation:

Flag3 is true if n1 is not equal to n2.

Implementation:

```
opcode: <>
0: SOURCE=DS INC[DP]  ALU=A-B ;;
1: JMP=01Z ;;

2: ALU=-1  DECODE ;;  ( Result not0, <> )
3: ALU=0   DECODE ;;  ( Result    0, = )
```


<cold>

Cold Start

(→)

Encoding:

0x04 8 cycles

Operation:

Opcode executed when the RESET pin is released. Use at any other time is not recommended

Implementation:

```
opcode: <cold>
0: ;;
1: ;;
2: ;;
3: ;;
4: ;;
5: ;;
6: ;;
7: ALU=0  DECODE ;;
```

<do>

Start Loop

(limit1 start2 →)
RS(→ limit1 index2)

Encoding:

0x74 3 cycles

Operation:

The limit1 and start2 value for a loop are moved from the data stack to the return stack (where the start value becomes the initial index value). This opcode is compatible with all looping instructions except NEXT.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: <do>
0: SOURCE=DS  INC[DP]  DEC[RP] DEST=RS ;;
1: SOURCE=DHI          DEC[RP] DEST=RS ;;
2: SOURCE=DS  INC[DP] ALU=B    DECODE ;;
```


<interrupt>

Interrupt

(→ n1)
RS(→ addr2)

Encoding:

0x01 2 cycles

Operation:

This is the opcode executed when an interrupt is recognized. The n1 value returned is the configuration register value before the interrupt was processed. This value contains the interrupt mask bit in bit 0 (*before* the interrupt mask bit was set by processing the interrupt), as well as the four bits indicating data or return stack interrupts. The interrupt restart address plus 4 is pushed as value addr2 onto the return stack (in order to restart after an interrupt, the value addr2-4 is used as the restart address).

Notes:

This opcode is automatically invoked by the hardware when processing an interrupt. Its definition is subject to change in future chip revisions, so using it for other purposes would be dim-witted.

Implementation:

```
opcode: <interrupt>
0: DEC[RP]
   ( Required to catch return address! )
```

```
   DEC[DP]  DS-FROM-DHI
   SOURCE=CONFIG  ALU=B ;;
1: DECODE ;;
```


<lit>

Push Long Immediate

(→ n1)

Encoding:

0x15 / call 4 cycles

Operation:

Returns an in-line 32-bit literal value n1. The <LIT> opcode is compiled with a CALL instruction whose address field points to the next instruction to be executed, typically (but not necessarily) at the address of the <LIT> instruction plus 8. The memory word after the <LIT> instruction (at <LIT> instruction plus 4 in memory) contains the 32-bit value returned as n.

Implementation:

```
opcode: <lit>
0: SOURCE=RETURN-SAVE
                  ADDR=BUS+0(CYCLE) ;;
1: DEC[DP]  DS-FROM-DHI    INC[RP] ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;
```

<loop>

Loop

(→)
RS(limit1 index2 → limit1 index3
... →)

Encoding:

0x1A / call 6/9 clocks

Operation:

Adds 1 to the loop index index2, then checks to see if the new index value index3 exceeds (is greater than or equal to) the loop limit limit1. If index3 exceeds limit1, the index and limit are popped off the return stack and execution continues in-line. If index3 does not exceed limit1, the new index value is stored on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. The opcode takes 6 clock cycles if the branch is taken, 9 clock cycles if not taken. This opcode assumes that index3 and limit1 will never differ in value more than 0x80000000. In other words, it does not check for overflow when doing the comparison.

Implementation:

```
opcode: <loop>
0: DEC[DP]  DS-FROM-DHI
   SOURCE=RETURN-SAVE     DEST=DLO ;;
1:              INC[RP]    ALU=0 ;;
2: SOURCE=RS    INC[RP]    ALU=A+B+1 ;;
3: SOURCE=RS     ALU=A-B    DEST=DHI[1] ;;
4: SOURCE=DHI    DEC[RP] DEST=RS
                         JMP=11S ;;

( Negative result, continue loop )
7: SOURCE=DS INC[DP]    ALU=B     DECODE ;;
( Non-Negative comparison, abort loop )
6: SOURCE=DLO    ADDR=BUS+0(CYCLE)
                  INC[MPC]    JMP=101 ;;
NEXT opcode
5: INC[RP] ;;
6: INC[RP]  LATCH-INSTRUCTION ;;
7: SOURCE=DS  INC[DP]    ALU=B  DECODE ;;
```

<next>

Count Down Loop

(→)
RS(count1 → count2
... →)

Encoding:

0xE1 4 cycles

Operation:

Decrements loop counter count1 (at top of return stack) by 1, then checks for new count2 value equal to zero. If count2 is zero, then it is popped off the return stack and execution continues in-line. If the count2 is non-zero, it is placed on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. Note that an initial count value of -1 performs 4 giga-iterations.

Implementation:

```
opcode: <next>
( Remember that 1st cycle will get pre-call RS
      value )
0: SOURCE=RS   DHI[1]  ALU=B ;;
1: SOURCE=RS INC[RP]
        ADDR=BUS+0(CYCLE)
        DHI[1]  ALU=A-1   JMP=01Z ;;

( Zero flag, fall through )
3: INC[RP] ;;
```



```
4: LATCH-INSTRUCTION ;;
5: DECODE ;;

( Non-zero flag, take the branch )
2: SOURCE=DHI[1]   DEST=RS   JMP=110 ;;
6: ;;
7: DECODE ;;
```

<short>

Push Short Immediate

(→ n1) Immediate

Encoding:

0x16 / lit	2 cycles
0x18 / lit	1 cycle (2OPS format)

Operation:

Pushes the compiled instruction lit field onto the stack as n1.

Notes:

Use the <LIT> opcode if a full 32-bit literal value is required.

Implementation:

```
opcode: <short>
0: SOURCE=LIT ALU=B
     DEC[DP]  DS-FROM-DHI ;;
1: DECODE ;;

opcode: <short>_QUICK
0: SOURCE=LIT ALU=B
     DEC[DP] DS-FROM-DHI  DECODE ;;
```


<trap>

Illegal Opcode Trap

(→ addr1 n2)

Encoding:

0x03	5 cycles

Operation:

Perform a subroutine call to address $FFFFFFFC (regardless of instruction type). The address just beyond the address of the instruction containing this opcode (*i.e.*, the instruction's address plus 4) is returned as addr1, and the value of the MISC register (which, among other things, contains the micro-address field indicating the value of the opcode causing the fault) is returned as n2. This opcode is invoked by hardware whenever an illegal opcode value is interpreted at execution time. It may also be used as a software interrupt if desired with appropriate trap handling software.

Implementation:

```
opcode: <trap>
0: DEC[DP]   DS-FROM-DHI
    SOURCE=RETURN-SAVE   ALU=B  ;;
1: SOURCE=0  ADDR=BUS-4(CYCLE)  ;;
2: DEC[DP]   DS-FROM-DHI
    SOURCE=MISC ALU=B  ;;
3: LATCH-INSTRUCTION ;;
4: DECODE ;;
```

=

Test For Equal

(n1 n2 → flag3)

Encoding:

0x7A 3 cycles

Operation:

Flag3 is true if n1 and n2 are equal.

Implementation:

```
opcode: =
0: SOURCE=DS INC[DP]  ALU=A-B ;;
1: JMP=01Z ;;

2: ALU=0   DECODE ;;  ( Result not0, <> )
3: ALU=-1  DECODE ;;  ( Result    0, = )
```


=branch

Jump If Equal

(n1 n2 →)

Encoding:

0x99 / call 5 cycles

Operation:

Branch if n1 and n2 are equal. The =branch opcode is compiled with a CALL instruction having an address field pointing to the branch target.

Implementation:

```
opcode: =branch
0: SOURCE=DS  INC[DP]  ALU=A-B  ;;
1: SOURCE=RS  ADDR=BUS+0(CYCLE) ;;
2: INC[RP]   JMP=10Z ;;

( Non-Zero flag, fall through )
4: LATCH-INSTRUCTION          JMP=111 ;;
7: SOURCE=DS INC[DP]  ALU=B   DECODE ;;
( Zero flag, take the branch )
5: ;;
6: SOURCE=DS INC[DP]  ALU=B   DECODE ;;
```

>

Test For Greater Than

(n1 n2 → flag3)

Encoding:

0x1E 4 cycles

Operation:

Flag3 is true if n1 is greater than n2.

Implementation:

```
opcode: >
0: SOURCE=DS  INC[DP]  ALU=A-B  ;;
1: JMP=01V ;;

2: JMP=10S ;;
3: JMP=11S ;;

4: ALU=0  DECODE ;;   ( not S  not V )
5: ALU=-1 DECODE ;;   (     S  not V )
6: ALU=-1 DECODE ;;   ( not S      V )
7: ALU=0  DECODE ;;   (     S      V )
```


>R

Transfer Dstack to Rstack

(n1 →)
RS(→ n1)

Encoding:

0x1F 2 cycles

Operation:

Pop n1 from the data stack, placing it on the return stack. This opcode also accomplishes the <for> function for use with the <next> looping opcode in the "FOR ... NEXT" construct.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with >R as the first opcode and NOP_QUICK as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: >R
0: DS-FROM-DHI    SOURCE=DS   ALU=B  ;;
1: SOURCE=DS  INC[DP]
      DEC[RP]   DEST=RS   DECODE ;;
```

?branch

Jump If Zero

(flag1 →)

Encoding:

0x20 4 cycles

Operation:

Perform a branch if flag1 is zero. ?BRANCH must be compiled as the opcode in a CALL instruction that has the branch target as its next address field.

Implementation:

```
opcode: ?branch
0: SOURCE=RETURN-SAVE
        ADDR=BUS+0(CYCLE) ;;
1: INC[RP]    JMP=01Z ;;

( Non-zero flag, fall through )
2: LATCH-INSTRUCTION    JMP=111 ;;
7: SOURCE=DS  INC[DP] ALU=B      DECODE ;;
( Zero flag, take the branch )
3: ;;
4: SOURCE=DS  INC[DP] ALU=B      DECODE ;;
```


?DNEGATE

Double Precision Conditional Negate

(d1 n2 → d3)

Encoding:

0xA3 5 cycles

Operation:

If n2 is non-negative, return d3 = d1. If n2 is negative, return d3 as the two's complement of d1.

Implementation:

```
opcode: ?DNEGATE
0: SOURCE=DS INC[DP]
        DHI[1]  ALU=notB  JMP=01S ;;

( Negative )
3: SOURCE=DS    ALU=notB ;;
4: ALU=A+1 ;;
5: DHI[0]  DS-FROM-DHI
    SOURCE=DHI[1]  ALU=B      JMP=11C ;;
6: ALU=A        DECODE ;;
7: ALU=A+1      DECODE ;;

( Non-Negative )
2: SOURCE=DHI[1]  ALU=notB
       INC[MPC] JMP=101 ;;
next opcode
( Nop padding for consistent timing )
5: ;;
6: ;;
7: DECODE ;;
```

?DUP

Conditional Push

(n1 → n1 n1

... → n1)

Encoding:

0x7C 2 cycles

Operation:

Push a second copy of n1 onto the stack if it is non-zero, else leave it alone.

Implementation:

```
opcode: ?DUP
0: JMP=01Z ;;

( Push )
2: DEC[DP]  DS-FROM-DHI   DECODE  ;;
( Don't Push )
3:                        DECODE  ;;
```


?EXIT

Conditional Exit

(flag1 →)

Encoding:

0xE0 4 cycles

Operation:

Perform a subroutine exit if flag1 is non-zero.

Notes:

This opcode should only be used as a JNEXT opcode or as the second opcode of a 2OPS instruction.

Do not use this as the very first opcode of a subroutine, or improper operation will result (a 5-clock version could be written that does not have this limitation).

Implementation:

```
opcode: ?EXIT
0: SOURCE=RS   ADDR=BUS+0(CYCLE) ;;
1: JMP=01Z ;;

( don't exit )
2:                              JMP=100 ;;
( exit )
3: LATCH-INSTRUCTION  INC[RP]  JMP=100 ;;
4: SOURCE=DS INC[DP]   ALU=B    DECODE ;;
```

?NEGATE

Conditional Negation

(n1 n2 → n3)

Encoding:

0x97 2 cycles

Operation:

If n2 is less than zero, perform a two's complement negation on integer n1, returning integer n3. Otherwise, return n1 as n3.

Implementation:

```
opcode: +-
0: SOURCE=DS INC[DP]   DHI[1]
      ALU=notB     JMP=01S ;;

( non-negative )
2: DHI[1]  ALU=notA DEST=DHI[0] DECODE ;;
( negative )
3: DHI[1]  ALU=A+1 DEST=DHI[0]  DECODE ;;
```


@

Load

(addr1 → n2)

(→ n2) Immediate

Encoding:

0x21 4 cycles
0xB2 / lit 4 cycles

Operation:

Fetches the 32-bit value at addr1, returning n2. In the immediate variant, the address addr1 is supplied by the literal field. Note that in this case the address is a byte-addressed literal, *not* the address field found in a CALL instruction.

Implementation:

```
opcode: @
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;

opcode: LIT_@
0: SOURCE=LIT   ADDR=BUS+0(CYCLE)  ;;
1: SOURCE=DHI   DEC[DP]  DEST=DS  ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```

@_!

Store Indirect

(n1 addr2 →)

Encoding:

0x9B 7 cycles

Operation:

Performs an indirect store of n1 through addr2. In other words, the value n1 is stored at the address contained in the memory word addressed by addr2.

Implementation:

```
opcode: @_!
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD   ADDR=BUS+0(CYCLE) ;;
4: SOURCE=DS  INC[DP]   DEST=RAM-! ;;
5: ;;
6: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```


@_+

Fetch Then Add

(n1 addr2 → n3)

Encoding:

0x9C 4 cycles

Operation:

Add the value contained in memory location addr2 to n1, giving result n3.

Implementation:

```
opcode: @_+
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=DS INC[DP]  ALU=B  ;;
3: SOURCE=RD ALU=A+B   DECODE ;;
```

@_@

Load Indirect

(addr1 → n2)

Encoding:

0x9A 7 cycles

Operation:

Performs an indirect fetch through addr1. In other words, the value n2 is fetched from the address contained in the memory word addressed by addr1.

Implementation:

```
opcode: @_@
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD ADDR=BUS+0(CYCLE) ;;
4: ;;
5: ;;
6: SOURCE=RD  ALU=B   DECODE ;;
```


@_INC

Incrementing Fetch

(addr1 → addr2 n3) Immediate

Encoding:

0xEE / lit 4 cycles

Operation:

Perform a fetch operation from addr1, returning the value n3. Also, add the compiled literal value to addr1 after the fetch is performed to return addr2.

Implementation:

```
opcode: @_INC
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: SOURCE=LIT  ALU=A+B ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```

@_LIT+

Fetch Then Add Immediate

(addr2 → n2) Immediate

Encoding:

0x9D / lit 4 cycles

Operation:

Add the value contained in memory at the address given by the compiled instruction literal field to n1, giving result n2.

Implementation:

```
opcode: @_LIT+
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=LIT  ALU=B ;;
3: SOURCE=RD ALU=A+B   DECODE ;;
```


ABS

Absolute Value

(n1 → n2)

Encoding:

0x7D 2 cycles

Operation:

Take the absolute value of n1, returning n2. n2 equals n1 if n1 is greater than or equal to 0. n2 is the two's complement of n1 if n1 is less than zero.

Implementation:

```
opcode: ABS
0: ALU=notA   JMP=01S ;;

( Non-negative, restore to original )
2: ALU=notA   DECODE ;;
( Two's complement )
3: ALU=A+1    DECODE ;;
```

ADC

Add With Carry

(n1 n2 cflagin → n3 cflagout)

Encoding:

0x9E 4 cycles

Operation:

Perform an add with carry. cflagin is a truth flag that represents the carry into the addition, and cflagout is a truth flag that represents the carry out of the addition, with a true value indicating carry set. n3 is the sum of n1 and n2 plus a 0 or 1 (depending on cflagin). cflagout is the carry out of the addition. This is a useful primitive for synthesizing extended precision arithmetic.

Implementation:

```
opcode: ADC
0: SOURCE=DS INC[DP]  ALU=B   JMP=01Z ;;
( cin <>0 )
2: SOURCE=DS  ALU=A+B+1 JMP=100 ;;
( cin = 0 )
3: SOURCE=DS  ALU=A+B   JMP=100 ;;
4: SOURCE=DHI  DEST=DS  JMP=11C ;;
( cout = 0 )
6: ALU=0   DECODE ;;
( cout <>0 )
7: ALU=-1  DECODE ;;
```


AND

Bitwise AND

(n1 n2 → n3)

(n1 → n3) Immediate

Encoding:

0x22	2 cycles
0x23	1 cycles (2OPS format)
0xAD / lit	2 cycles
0xAE / lit	1 cycle (2OPS format)

Operation:

Perform a bitwise logical AND of n1 and n2, returning n3. In the immediate variant, the value n2 is supplied by the literal field. Note that since the literal field is signed, the highest bits of the literal value may be either 0 (to mask n1 bits), or 1 (to allow n1 bits to propagate to n2).

Implementation:

```
opcode: AND
0: SOURCE=DS  INC[DP]   ALU=AandB  ;;
1: DECODE ;;

opcode: AND_QUICK
0: SOURCE=DS  INC[DP]   ALU=AandB
     DECODE ;;

opcode: LIT_AND
0: SOURCE=LIT  ALU=AandB  ;;
1: DECODE ;;
```

```
opcode: LIT_AND_QUICK
0: SOURCE=LIT  ALU=AandB  DECODE ;;
```


AND!

And To Memory

(n1 addr2 →)

Encoding:

0x9F 7 cycles

Operation:

Perform an atomic bitwise logical AND of n1 with the word at memory addr2, returning the result to addr2 (similar to +!, but with an AND operation instead of a + operation).

Implementation:

```
opcode: AND!
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW ;;
2: SOURCE=DS INC[DP]   ALU=B  ;;
3: SOURCE=RD   ALU=AandB    CYCLE-RAM ;;
4: SOURCE=DHI  DEST=RAM-! ;;
5: ;;
6: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```

ASIC!

I/O Output

(n1 portaddr2 →)

(n1 →) Immediate

Encoding:

0x58	4 cycles
0x59 / lit	4 cycles

Operation:

Write value n1 to output device using address portaddr2. In the immediate variant, the port address portaddr2 is provided by the literal field.

Implementation:

```
opcode: ASIC!
0: SOURCE=DHI     ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS  INC[DP]    DEST=ASIC-! ;;
2: ;;
3: SOURCE=DS  INC[DP]    ALU=B    DECODE ;;

opcode: LIT_ASIC!
0: SOURCE=LIT     ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI             DEST=ASIC-! ;;
2: ;;
3: SOURCE=DS  INC[DP]    ALU=B    DECODE ;;
```


ASIC@

I/O Input

(portaddr1 → n2)

(→ n2) Immediate

Encoding:

0x5A	4 cycles
0x5B / lit	4 cycles

Operation:

Read value n2 from input device using address portaddr1. In the immediate variant, protaddr1 is supplied by the literal field.

Implementation:

```
opcode: ASIC@
0: SOURCE=DHI     ADDR=BUS+0(CYCLE) ;;
1: ASIC-@ ;;
2: ;;
3: SOURCE=RD  ALU=B          DECODE ;;

opcode: LIT_ASIC@
0: SOURCE=LIT     ADDR=BUS+0(CYCLE) ;;
1: ASIC-@  ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD  ALU=B          DECODE ;;
```

ASR

Arithmetic Shift Right

(n1 → n2)

Encoding:

0x24 2 cycles

Operation:

Perform an arithmetic shift right of n1 by one bit, giving n2.

Notes:

This is equivalent to division by two for non-negative integers, and equivalent to floored division by two for all integers.

Implementation:

```
opcode: ASR
0: JMP=01S ;;

( non-negative: shift in a 0 )
2: ALU=A  CIN=0  SR[ALU]    DECODE ;;
( negative: shift in a 1 )
3: ALU=A  CIN=1  SR[ALU]    DECODE ;;
```


B@

Load Signed Byte

(addr1 → b2)

Encoding:

0x28 4 cycles

Operation:

Fetch byte value b1 from address addr2. The value b2 is sign extended to form a 32-bit signed integer.

Implementation:

```
opcode: B@
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: RAM-C@ ;;
2: ;;
3: SOURCE=RD-SIGNED   ALU=B    DECODE ;;
```

BIT-CLEAR

Bit Clear

(n1 n2 → n3)

Encoding:

0xC3 2 cycles

Operation:

Every bit set in n2 is cleared in value n1, giving result n3. This is a bit clear operation that uses n2 as an enabling mask for clearing bits in n1.

Implementation:

```
opcode: BIT-CLEAR
0: ALU=notA ;;
1: SOURCE=DS INC[DP]
       ALU=AandB  DECODE ;;
```


branch

Unconditional Branch

(→)

Encoding:

0x25 / call 2 cycles

Operation:

Perform an unconditional branch. The BRANCH opcode is compiled with a CALL instruction having an address field pointing to the branch target.

Notes:

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip).

Implementation:

```
opcode: branch
0: ;;
1: INC[RP]  DECODE ;;
```

C!

Store Character

(c1 addr2 →)

Encoding:

0x26 4 cycles

Operation:

Store character value c1 at address addr2. The lowest 8 bits of value c1 are stored.

Implementation:

```
opcode: C!
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS  INC[DP]  DEST=RAM-C! ;;
2: ;;
3: SOURCE=DS  INC[DP]  ALU=B
     DECODE ;;
```


C!_INC

Incrementing Store Character

(c1 addr2 → addr3) Immediate

Encoding:

0xF1 / lit 4 cycles

Operation:

Store value c1 at location addr2, then add the compiled instruction literal field to addr2, leaving addr3.

Implementation:

```
opcode: C!_INC
0: SOURCE=DHI
    ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS  INC[DP]    DEST=RAM-C! ;;
2: ;;
3: SOURCE=LIT  ALU=A+B  DECODE ;;
```

C+!

Indirect Increment Byte

(b1 addr2 →)

Encoding:

0xA1 7 cycles

Operation:

Perform an 8-bit addition of b1 to the byte at memory addr2, returning the result to addr2 (similar to +!, but with an 8-bit operation).

Implementation:

```
opcode: C+!
0: SOURCE=DHI   ADDR=BUS+0(CYCLE) ;;
1: RAM-C@ ;;
2: SOURCE=DS  INC[DP]    ALU=B  ;;
3: SOURCE=RD    ALU=A+B  CYCLE-RAM ;;
4: SOURCE=DHI   DEST=RAM-C!  ;;
5: SOURCE=DS  INC[DP]    ALU=B  ;;
6: DECODE ;;
```


C@

Load Character

(addr1 → c2)

Encoding:

0x27 4 cycles

Operation:

Fetch character value c2 from address addr1. The value c2 is placed in the lowest 8 bits of the stack word and padded with leading zeros.

Implementation:

```
opcode: C@
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: RAM-C@ ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```

C@_INC

Incrementing Fetch Character

(addr1 → addr2 c3) Immediate

Encoding:

0xEF / lit 4 cycles

Operation:

Perform a character fetch operation from addr1, returning the value n3. Also, add the compiled literal value to addr1 after the fetch is performed to return addr2.

Implementation:

```
opcode: C@_INC
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: SOURCE=LIT   ALU=A+B   RAM-C@ ;;
2: SOURCE=DHI   DEC[DP]  DEST=DS ;;
3: SOURCE=RD    ALU=B    DECODE ;;
```


CMOVE

Move Bytes

(src1 dest2 cnt3 →)

Encoding:

0x8A 4 + 4*cnt3 cycles

Operation:

Move a block of cnt3 bytes of data starting at address src1 to a block of memory starting at address dest2.

Notes:

A cnt3 value of 0 will attempt to move 4G bytes.

CMOVE destroys the value of registers SBASE and DBASE.

This is a non-interruptible instruction. The commercial chip will have an interruptible version of this opcode.

Implementation:

```
opcode: CMOVE
0: SOURCE=DS INC[DP]  DEST=DBASE ;;
1: SOURCE=DS INC[DP]  DEST=SBASE
     DHI[1]  ALU=0    ( offset ) ;;

( Byte move loop )
2: SOURCE=DHI[1]  ADDR=BUS+SBASE(CYCLE)
     JMP=101 ;;
5: RAM-C@ ;;
```

```
( Bump & Test count)
6: SOURCE=DHI[1]  ADDR=BUS+DBASE(CYCLE)
    DHI[0]  ALU=A-1  ;;
7: SOURCE=RD   DEST=RAM-C!
    DHI[1]  ALU=A+1  JMP=01Z ;;

( End loop )
3: ;;   ( wait for RAM cycle to complete)
4: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```


CONFIG!

Set Configuration Register

(n1 →)

Encoding:

0x6A 2 cycles

Operation:

Store n1 into the on-chip configuration register, setting stack over/underflow interrupt bits and the interrupt mask.

*** picture of config reg could go here ***

Notes:

This implementation should have a nop microinstruction in it to allow the config register to properly mask/unmask interrupts before executing the DECODE.

Implementation:

```
opcode: CONFIG!
0: SOURCE=DHI  DEST=CONFIG ;;
1: SOURCE=DS INC[DP]  ALU=B    DECODE ;;
```

CONFIG@

Get Configuration Register

(→ n1)

Encoding:

0x69 2 cycles

Operation:

Fetch n from the on-chip configuration register, reading stack over/underflow interrupt bits and the interrupt mask.

*** picture of config reg could go here ***

Implementation:

```
opcode: CONFIG@
0: DEC[DP]  DS-FROM-DHI
    SOURCE=CONFIG  ALU=B  ;;
1: DECODE ;;
```


COUNT

Load Count

(addr1 → addr2 un3)

Encoding:

0x8E 4 cycles

Operation:

Fetch the unsigned byte count value at addr1, returning the start of a string address addr2 (equal to addr1 plus 1) and the count (the unsigned byte at addr1). This primitive is used in conjuntion with Forth counted strings.

Implementation:

```
opcode: COUNT
0: SOURCE=DHI
   ADDR=BUS+0(CYCLE)   ALU=A+1 ;;
1: RAM-C@ ;;
2: ;;
3: DEC[DP]  DS-FROM-DHI
       SOURCE=RD  ALU=B        DECODE ;;
```

C_OR!

Or Character To Memory

(c1 addr2 →)

Encoding:

0xA2 7 cycles

Operation:

Perform an 8-bit bitwise logical OR of c1 to the byte at memory addr2, returning the result to addr2 (similar to C+!, but with an OR operation instead of an addition).

Implementation:

```
opcode: C_OR!
0: SOURCE=DHI   ADDR=BUS+0(CYCLE) ;;
1: RAM-C@ ;;
2: SOURCE=DS  INC[DP]    ALU=B  ;;
3: SOURCE=RD    ALU=AorB  CYCLE-RAM ;;
4: SOURCE=DHI   DEST=RAM-C!  ;;
5: SOURCE=DS  INC[DP]    ALU=B  ;;
6: DECODE ;;
```


D!

Double Precision Store

(d1 addr2 →)

Encoding:

0x29 7 cycles

Operation:

Store 64-bit value d1 at address addr2, placing the lower order word at address addr2+0, and the higher order word at address addr2+4.

Implementation:

```
opcode: D!
0: SOURCE=DHI  ADDR=BUS+0(CYCLE)
          INC[DP] ;;
1: SOURCE=DS   DEST=RAM-! ;;
2: SOURCE=4   ALU=A+B ;;
3: SOURCE=DHI  ADDR=BUS+0(CYCLE)
           DEC[DP] ;;
4: SOURCE=DS   DEST=RAM-!      INC[DP] ;;
5:                             INC[DP] ;;
6: SOURCE=DS   ALU=B           INC[DP]
       DECODE ;;
```

D+

Double Precision Add

(d1 d2 → d3)

Encoding:

0x2A 5 cycles

Operation:

Add 64-bit values d1 and d2, resulting in d3.

Implementation:

```
opcode: D+
0: SOURCE=DS  INC[DP]  ALU=B  DEST=DHI[1] ;;
1: SOURCE=DS  INC[DP]         DEST=DLO ;;
2: SOURCE=DS   DHI[1]  ALU=A+B ;;
3: SOURCE=DHI[1]     DEST=DS   JMP=10C ;;

( Carry false )
4: SOURCE=DLO ALU=A+B           DECODE ;;
( Carry true )
5: SOURCE=DLO ALU=A+B+1         DECODE ;;
```


D–

Double Precision Subtract

(d1 d2 → d3)

Encoding:

0x86 5 cycles

Operation:

Subtract 64-bit value d2 from d1, giving d3.

Implementation:

```
opcode: D-
0: SOURCE=DS INC[DP]  DHI[1] ALU=notB  ;;
1: SOURCE=DS  INC[DP]
       ALU=notA   DEST=DLO ;;
2: SOURCE=DS    DHI[1]  ALU=A+B+1 ;;
3: SOURCE=DHI[1]    DEST=DS   JMP=10C ;;

( Carry false )
4: SOURCE=DLO ALU=A+B           DECODE ;;
( Carry true )
5: SOURCE=DLO ALU=A+B+1         DECODE ;;
```

D=

Test for Double Precision Equal

(d1 d2 → flag)

Encoding:

0xA5 5 cycles

Operation:

Compare d1 to d2, returning a true flag if they are equal.

Implementation:

```
opcode: D=
0: SOURCE=DS  INC[DP]  DHI[1]  ALU=B ;;
1: SOURCE=DS  INC[DP]  DHI[0]  ALU=A-B ;;
2: SOURCE=DS  INC[DP]
          DHI[1]   ALU=A-B   JMP=10Z ;;

4: ALU=0 JMP=110 ;; (low words not equal)
5: JMP=11Z ;;        ( low words equal )

6: ALU=0  DECODE ;;  ( not equal )
7: ALU=-1 DECODE ;;  ( both words equal )
```


D>R

Double Precision Transfer Dstack To Rstack

(n1 n2 →)
RS(→ n2 n1)

Encoding:

0x2B 3 cycles

Operation:

Transfer the pair n1 and n2 from the data stack to the return stack. Note that the order of n1 and n2 on the stack are reversed. D>R is identical in operation to execution of the pair ">R >R". However, as long as DR> is used to transfer the results back, D>R may be thought of as a word that transfers a double precision number to the return stack as well.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with D>R as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: D>R
0: SOURCE=DHI          DEC[RP] DEST=RS  ;;
1: SOURCE=DS INC[DP]   DEC[RP] DEST=RS  ;;
2: SOURCE=DS  INC[DP]   ALU=B    DECODE ;;
```


D@

Double Precision Load

(addr1 → d2)

Encoding:

0x2C 6 cycles

Operation:

Fetch 64-bit value d1 from address addr1, reading the low order word from address addr1+0, and the high order word from address addr1+4.

Implementation:

```
opcode: D@
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: SOURCE=4  ALU=A+B ;;
2: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
3: SOURCE=RD    DEC[DP]  DEST=DS  ;;
4: ;;
5: SOURCE=RD    ALU=B      DECODE  ;;
```

DASR

Double Precision Arithmetic Shift Right

(d1 → d2)

Encoding:

0x2D 3 cycles

Operation:

Perform a 64-bit arithmetic shift right of d1, giving d2.

Implementation:

```
opcode: DASR
0: SOURCE=DS   DEST=DLO    JMP=01S ;;

( non-negative: shift in a 0 )
2: ALU=A   CIN=0  SR[ALU]
           SR[DLO]   JMP=100 ;;
( negative: shift in a 1 )
3: ALU=A   CIN=1  SR[ALU]  SR[DLO]  ;;
4: SOURCE=DLO  DEST=DS
        DECODE ;;
```


DBASE!

Set DBASE Register

(n1 →)

Encoding:

0x5C 2 cycles

Operation:

Store value n1 in register DBASE.

Implementation:

```
opcode: DBASE!
0: SOURCE=DHI  DEST=DBASE ;;
1: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```

DBASE+!

Add To DBASE

(n1 →)

Encoding:

0xEB 3 cycles

Operation:

Add the value n1 to the contents of DBASE, and place the sum back in the DBASE register.

Implementation:

```
opcode: DBASE+!
0: SOURCE=DBASE  ALU=A+B ;;
1: SOURCE=DHI  DEST=DBASE ;;
2: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```


DBASE+_!

Store Indexed With DBASE

(n1 offset2 →)

Encoding:

0xD8 4 cycles

Operation:

Store value n1 at address computed by adding offset2 to the value of DBASE.

Implementation:

```
opcode: DBASE+_!
0: SOURCE=DHI  ADDR=BUS+DBASE(CYCLE)  ;;
1: SOURCE=DS INC[DP]  DEST=RAM-! ;;
2: ;;
3: SOURCE=DS INC[DP]  ALU=B  DECODE ;;
```

DBASE+_@

Load Indexed With DBASE

(offset1 → n2)

Encoding:

0xD7 4 cycles

Operation:

Fetch value n2 from address computed by adding offset1 to the value of DBASE.

Implementation:

```
opcode: DBASE+_@
0: SOURCE=DHI  ADDR=BUS+DBASE(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```


DBASE@

Get From DBASE

(→ n1)

Encoding:

0x5D 2 cycles

Operation:

Fetch the value n1 from register DBASE.

Implementation:

```
opcode: DBASE@
0: DEC[DP]   DS-FROM-DHI
     SOURCE=DBASE    ALU=B  ;;
1: DECODE ;;
```

DDROP

Double Precision Drop

(d1 →)

Encoding:

0x75 2 cycles

Operation:

Drop d1 from the data stack.

Implementation:

```
opcode: DDROP
0: INC[DP] ;;
1: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```


DDUP

Double Precision Duplicate

(d1 → d1 d1)

Encoding:

0x76 2 cycles

Operation:

Duplicate d1 on the data stack.

Implementation:

```
opcode: DDUP
0: SOURCE=DS  ALU=B      DEC[DP]  DS-FROM-DHI  ;;
1: SOURCE=DS   ALU=B        DEC[DP]   DS-FROM-DHI
      DECODE ;;
```

DISABLE

Disable Interrupts

(→)

Encoding:

0x6C 3 cycles

Operation:

Disable maskable interrupts by setting the mask flag in CONFIG register.

Notes:

This implementation may allow an interrupt to be accepted just after the opcode executes. Future chip versions may include a NOP in the microcode to change this behavior.

Implementation:

```
opcode: DISABLE
0: DHI[1]  SOURCE=CONFIG
             ALU=B  SR[ALU] ;;
1: DHI[1]  ALU=A+A+1  ;;
            ( Sets lowest bit )
2: SOURCE=DHI[1]   DEST=CONFIG  DECODE ;;
```


DLSL

Double Precision Logical Shift Left

(d1 → d2)

Encoding:

0x7E 3 cycles

Operation:

Shift d1 left one bit, shifting in a zero bit, resulting in d2 (same as multiplication by 2).

Implementation:

```
opcode: DLSL
0: SOURCE=DS  DEST=DLO ;;
1: SL[ALU]  SL[DLO]  CIN=0 ;;
2: SOURCE=DLO  DEST=DS  DECODE ;;
```

DLSR

Double Precision Logical Shift Right

(d1 → d2)

Encoding:

0xA6 3 cycles

Operation:

Logical shift d1 right 1 bit, shifting a 0 into the highest bit, resulting in d2.

Implementation:

```
opcode: DLSR
0: SOURCE=DS  DEST=DLO ;;
1: CIN=0   SR[ALU]  SR[DLO] ;;
2: SOURCE=DLO  DEST=DS     DECODE ;;
```


DNEGATE

Double Precision Negate

(d1 → d2)

Encoding:

0x85 4 cycles

Operation:

Take the two's complement of d1, giving d2.

Implementation:

```
opcode: DNEGATE
0: SOURCE=DS  DHI[1] ALU=notB ;;
1: DHI[1]  ALU=A+1 ;;
2: SOURCE=DHI[1]  DEST=DS
           ALU=notA   JMP=10C ;;

4: ALU=A           DECODE ;;
5: ALU=A+1         DECODE ;;
```

docon

Constant Value

(→ n1)
RS(addr2 →)

Encoding:

0x2E 4 cycles

Operation:

Fetch the in-line 32-bit value n1 and perform a subroutine return to address addr2

Notes:

MUST be compiled as an EXIT instruction for proper operation.

Implementation:

```
opcode: docon
0: SOURCE=RETURN-SAVE
                ADDR=BUS+0(CYCLE) ;;
1: ;;
2: ;;
3: DEC[DP]    DS-FROM-DHI
      SOURCE=RD   ALU=B          DECODE ;;
```


dovar

Variable Address

(→ addr1)
RS(addr2 →)

Encoding:

0x2F 2 cycles

Operation:

Return the address of an in-line 32-bit variable as addr1 and perform a subroutine return to addr2 (MUST be compiled as an EXIT instruction for proper operation).

Implementation:

```
opcode: dovar
0: DEC[DP]  DS-FROM-DHI
     SOURCE=RETURN-SAVE    ALU=B ;;
1: DECODE ;;
```

DOVER

Double Precision Get Second Element

(d1 d2 → d1 d2 d1)

Encoding:

0x7F 6 cycles

Operation:

Perform a double-precision OVER operation, copying d1 to the top of the stack.

Implementation:

```
opcode: DOVER
0: INC[DP]   ALU=A  DEST=DHI[1] ;;
1: INC[DP]  ;;
2: SOURCE=DS  DEC[DP] DEST=DLO ;;
3: SOURCE=DS  DEC[DP]  ALU=B  ;;
4: SOURCE=DHI[1]  DEC[DP]  DEST=DS ;;
5: SOURCE=DLO
         DEC[DP]  DEST=DS   DECODE ;;
```


DR>

Double Precision Rstack To Dstack

(→ n1 n2)
RS(n2 n1 →)

Encoding:

0x30 2 cycles

Operation:

Transfer the pair n1 and n2 from the return stack to the data stack. Note that the order of n1 and n2 on the stack are reversed. DR> is identical in operation to execution of the pair "R> R>". However, as long as DR> is used to transfer the results that were previously transferred by a D>R, it may be thought of as a word that transfers a double precision number to the data stack as well.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with DR> as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: DR>
0: DEC[DP]   DS-FROM-DHI
    SOURCE=RS    INC[RP]   ALU=B  ;;
1: DEC[DP]   DS-FROM-DHI
    SOURCE=RS INC[RP]  ALU=B  DECODE ;;
```


DROP

Drop

(n1 →)

Encoding:

0x31	2 cycles
0x32	1 cycle (2OPS format)

Operation:

Pop the top stack element n1, discarding it.

Implementation:

```
opcode: DROP
0: SOURCE=DS  INC[DP]   ALU=B  ;;
1: DECODE ;;

opcode: DROP_FAST
0: SOURCE=DS  INC[DP]   ALU=B   DECODE ;;
```

DROP_LIT

Drop Then Push Immediate

(n1 → n2)

Encoding:

0xA7 / lit 2 cycles

Operation:

Drop n1 from the top stack element, then replace it by pushing on n2 from the compiled instruction literal field.

Implementation:

```
opcode: DROP_LIT
0: SOURCE=LIT  ALU=B ;;
1: DECODE ;;
```


DROT

Double Precision Rotate

(d1 d2 d3 → d2 d3 d1)

Encoding:

0x81 9 cycles

Operation:

Move d1 from the third element on the data stack to the top element.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with DROT as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: DROT
0: SOURCE=DS INC[DP]  DEST=DLO ;;
1: SOURCE=DS INC[DP]  DEC[RP] DEST=RS  ;;
2: SOURCE=DS INC[DP]  DHI[1]  ALU=B ;;
3: SOURCE=DS INC[DP]  DEC[RP] DEST=RS  ;;
4: DHI[1]  DS-FROM-DHI
    SOURCE=DS  ALU=B     INC[RP] ;;
5: SOURCE=RS  DEC[RP]  DEC[DP] DEST=DS ;;
```

```
6: SOURCE=DLO          DEC[DP]  DEST=DS ;;
7: DEC[DP]  DHI[0] DS-FROM-DHI
    SOURCE=RS  INC[RP]
    ALU=B    INC[MPC]  JMP=111 ;;
next opcode
7: SOURCE=DHI[1]  DEC[DP]  DEST=DS
    INC[RP]  DECODE ;;
```


DS!

Store Into Data Stack

(n1 addr2 →)

Encoding:

0xC6 7 cycles

Operation:

Using the Data Stack as a separately addressed memory, store value n1 at address addr2. The highest bits of addr2 (outside the range of the stack memory space) are ignored. No checking is performed to detect interference with stack operation, and this opcode will generate a stack overflow/underflow interrupt if addr2 is outside the permissible stack operating range.

Implementation:

```
opcode: DS!
0: SOURCE=DP  DHI[1]  ALU=B  ;;
1: SOURCE=DHI[0]    DEST=DP
        DHI[1]  ALU=A+1 ;;
2: SOURCE=DS  DEST=DLO ;;
3: SOURCE=DLO  DEST=DS ;;
4: SOURCE=DHI[1]  DEST=DP ;;
5: ;;
6: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```

DS@

Load From Data Stack

(addr1 → n2)

Encoding:

0xC5 5 cycles

Operation:

Using the Data Stack as a separately addressed memory, fetch the value n2 from address addr1. The highest bits of addr1 (outside the range of the stack memory space) are ignored. No checking is performed to detect interference with stack operation, and this opcode will generate a stack overflow/underflow interrupt if addr1 is outside the permissible stack operating range.

Implementation:

```
opcode: DS@
0: SOURCE=DP   DEST=DLO  ;;
1: SOURCE=DHI   DEST=DP ;;
2: ;;
3: SOURCE=DLO   DEST=DP ;;
( Exploits fact that DSREG maintains value
    while waiting for the new DP to access
   the DS RAM. )
4: SOURCE=DS   ALU=B   DECODE ;;
```


DSWAP

Double Precision Swap

(d1 d2 → d2 d1)

Encoding:

0x80 5 cycles

Operation:

Exchange the top two double precision stack elements, moving d1 from the second stack position to the top of the stack.

Implementation:

```
opcode: DSWAP
0: SOURCE=DS INC[DP]  DHI[1]  ALU=B ;;
1: SOURCE=DS INC[DP]  DEST=DLO  ;;
2: DHI[1] DS-FROM-DHI
       SOURCE=DS  ALU=B ;;
3: DHI[0] DEC[DP]  DS-FROM-DHI
       SOURCE=DLO  ALU=B ;;
4: SOURCE=DHI[1]
       DEC[DP]  DEST=DS  DECODE ;;
```

DUP

Duplicate

(n1 → n1 n1)

Encoding:

0x33	2 cycles
0x34	1 cycle (2OPS format)

Operation:

Duplicate the top element n1 on the stack.

Implementation:

```
opcode: DUP
0: SOURCE=DHI     DEC[DP]  DEST=DS ;;
1: DECODE ;;

opcode: DUP_FAST
0: SOURCE=DHI  DEC[DP]  DEST=DS
       DECODE ;;
```


DUP_0<

Test For Less Than Zero (Non-Destructive)

(n1 → n1 flag2)

Encoding:

0xA8	2 cycles

Operation:

Nondestructively test the top stack element n1, returning a true flag2 if n1 is less than zero, else returning a false flag2.

Implementation:

```
opcode: DUP_0<
0: SOURCE=DHI  DEC[DP]  DEST=DS
               JMP=01S ;;

2: ALU=0    DECODE ;;
3: ALU=-1   DECODE ;;
```

DUP_@

Load (Non-Destructive)

(addr1 → addr1 n2)

Encoding:

0xA9 4 cycles

Operation:

Perform a nondestructive load operation, leaving addr1 on the stack unchanged, and fetching value n2 from memory at addr1.

Implementation:

```
opcode: DUP_@
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: SOURCE=DHI  DEC[DP]  DEST=DS ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```


ENABLE

Enable Interrupts

(→)

Encoding:

0x6B 3 cycles

Operation:

Enable maskable interrupts by clearing the mask flag in CONFIG register.

Notes:

This implementation may not allow an interrupt to be accepted just after the opcode executes. Future chip versions may include a NOP in the microcode to change this behavior.

Implementation:

```
opcode: INT_ENABLE
0: DHI[1]  SOURCE=CONFIG
     ALU=B  SR[ALU] ;;
1: DHI[1]  ALU=A+A  ;; (lears lowest bit)
2: SOURCE=DHI[1]   DEST=CONFIG  DECODE ;;
```

EXECUTE

Call Data Stack Address

(addr1 →)

Encoding:

0x35 /call 4 cycles

Operation:

Perform a subroutine call or jump to addr1.

Notes:

This opcode must be compiled with a CALL instruction to *any* address (e.g. address 0) to accomplish a subroutine call. This dummy address will not be called, but rather the addr1 value from the stack will be substituted as the subroutine call target.

If a JNEXT instruction instead of a CALL instruction is used, this opcode performs an unconditional branch to the value addr1 on the data stack.

A 2OPS instruction or EXIT instruction may also be used, but the user is cautioned to beware of the effects on program control flow.

Implementation:

```
opcode: EXECUTE
0: SOURCE=DHI   ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS  INC[DP]   ALU=B  ;;
2: LATCH-INSTRUCTION ;;
3: DECODE ;;
```


FALSE

Push False Flag

(→ 0)

Encoding:

0x07 2 clocks

Operation:

Pushes constant 0. 0 is the "FALSE" flag value for the machine.

Implementation:

```
opcode: FALSE
0: DEC[DP] DS-FROM-DHI ALU=0 ;;
1: DECODE ;;
```

FETCH_AND_ADD

Add To Memory

(n1 addr2 → n3)

(addr2 → n3) Immediate

Encoding:

0xC8	7 cycles
0xC9 / lit	7 cycles

Operation:

Fetch the value from word addr2, and add it to n1 to produce n3. Store the value n3 back at the effective address, all as an atomic read/modify/write bus operation. Useful as a synchronization instruction. The immediate variant provides the value n1 from the literal field.

Implementation:

```
opcode: LIT_FETCH_AND_ADD
0: SOURCE=DHI    ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW ;;
2: SOURCE=LIT    ALU=B  ;;
3: SOURCE=RD     ALU=A+B  CYCLE-RAM ;;
4: SOURCE=DHI    DEST=RAM-!  ;;
5: ;;
6: DECODE ;;

opcode: FETCH_AND_ADD
0: SOURCE=DHI    ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW ;;
2: SOURCE=DS  INC[DP]     ALU=B  ;;
3: SOURCE=RD     ALU=A+B  CYCLE-RAM ;;
```



```
4: SOURCE=DHI    DEST=RAM-!  ;;
5: ;;
6: DECODE ;;
```

FILL

Fill Memory With Characters

(addr1 count2 c3 →)

Encoding:

0x8C 4 + 2*count cycles

Operation:

Fill count2 bytes of memory with value c3, starting at address addr1 and counting up.

Notes:

Destroys value in the DBASE register.

Implementation:

```
opcode: FILL
( Keep count/offset in DHI[1] )
0: SOURCE=DS INC[DP] DHI[1]  ALU=B
1: SOURCE=DS INC[DP]   DEST=DBASE
      DHI[1]  ALU=A-1 ;;

( Store bytes in reverse order, using count as
      offset )
2: SOURCE=DHI[1]   ADDR=BUS+DBASE(CYCLE)
             DHI[1] ALU=A-1  JMP=110 ;;
6: SOURCE=DHI[0]  DEST=RAM-C!  JMP=01S ;;

3: ;;
( Wait for RAM cycle to complete)
4: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


FP!

Put Frame Pointer Value

(n1 →)

Encoding:

0x60 2 cycles

Operation:

Store the value n1 into the FP register.

Implementation:

```
opcode: FP!
0: SOURCE=DHI  DEST=FP  ;;
1: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```

FP+!

Add To Frame Pointer Value

(n1 →)

(→) Immediate

Encoding:

0xE8	3 cycles
0xE9 / lit	3 cycles

Operation:

Add the value n1 to the contents of FP, and place the sum back in the FP register. In the immediate variant the value n1 is provided by the literal field.

Implementation:

```
opcode: FP+!
0: SOURCE=FP  ALU=A+B ;;
1: SOURCE=DHI  DEST=FP ;;
2: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;

opcode: LIT_FP+!
0: SOURCE=FP    DHI[1]  ALU=B ;;
1: SOURCE=LIT   DHI[1]  ALU=A+B ;;
2: SOURCE=DHI[1]  DEST=FP   DECODE ;;
```


FP@

Get FP Register

(→ n1)

Encoding:

0x61	2 cycles

Operation:

Fetch the value of the FP register, returning n1.

Implementation:

```
opcode: FP@
0: DEC[DP]   DS-FROM-DHI
    SOURCE=FP       ALU=B  ;;
1: DECODE ;;
```

FRAME_POP

Frame Deallocate Immediate

(→)

Encoding:

0xED 4 cycles

Operation:

Using the FP register, deallocate a memory-resident stack frame. FP is loaded with the value contained in the memory word address by adding the old FP contents and the compiled instruction literal field. The compiled instruction literal field must be -4 to work properly with FRAME_PUSH.

Implementation:

```
opcode: FRAME_POP
0: SOURCE=LIT   ADDR=BUS+FP(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD  DEST=FP    DECODE ;;
```


FRAME_PUSH

Frame Allocate Immediate

(→)

Encoding:

0xEC 6 cycles

Operation:

Using the FP register, allocate a number of bytes (determined by the compiled instruction literal field) to the memory-resident stack frame. The new value of FP is calculated by adding the old value of FP to the compiled instruction literal field. The old value of the FP register is saved at the address new FP-4.

Implementation:

```
opcode: FRAME_PUSH
0: DHI[1]  SOURCE=FP  ALU=B   DEST=DLO ;;
1: DHI[1]   SOURCE=LIT  ALU=A-B  ;;
2: SOURCE=DHI[1]  ADDR=BUS-4(CYCLE) ;;
3: SOURCE=DLO  DEST=RAM-! ;;
4: SOURCE=DHI[1]  DEST=FP  ;;
5: DECODE ;;
```

H!

Store Halfword

(h1 addr2 →)

Encoding:

0x36 4 cycles

Operation:

Store half-word value h1 at address addr2. The lowest 16 bits of value h1 are stored.

Implementation:

```
opcode: H!
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS DEST=RAM-W! INC[DP] ;;
2: ;;
3: SOURCE=DS ALU=B INC[DP]  DECODE ;;
```


H@

Load Halfword

(addr1 → h2)

Encoding:

0x37 4 cycles

Operation:

Fetch halfword value h1 from address addr. The 16-bit value is sign-extended to 32 bits.

Implementation:

```
opcode: H@
0: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
1: RAM-W@ ;;
2: ;;
3: SOURCE=RD-SIGNED   ALU=B    DECODE ;;
```

HALT

Halt Processor

(→)

Encoding:

0x05 ∞ cycles

Operation:

Enter an infinite microcode loop that is uninterruptible.

Implementation:

```
opcode: HALT
0: ;;
1: ;;
2: ;;
3: ;;
4: ;;
5: ;;
6: ;;
7: JMP=000 ;;
```


I'

Copy Second On Rstack To Dstack

(→ n1)

RS(n1 n2 → n1 n2)

Encoding:

0x3B 2 cycles

Operation:

Copy the second return stack element onto the data stack. In Looping constructs, this returns the loop limit value.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with I' as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: I'
0: SOURCE=DHI INC[RP]     DEC[DP]  DEST=DS ;;
1: SOURCE=RS  DEC[RP]     ALU=B     DECODE ;;
```

I'_!

Store With Rstack2

(n1 →)
RS(addr2 n3 → addr2 n3)

Encoding:

0xE2 5 cycles

Operation:

Store n1 at location addr2.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: I'_!
0: INC[RP] ;;
1: SOURCE=RS  DEC[RP]
     ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DHI            DEST=RAM-! ;;
3: ;;
4: SOURCE=DS  INC[DP]   ALU=B
        DECODE ;;
```


I'_4-_!

Store With Rstack2 Minus 4

(n1 →)
RS(addr2 n3 → addr2 n3)

Encoding:

0xE5 5 cycles

Operation:

Store n1 at location addr2.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: I'_4-_!
0: INC[RP] ;;
1: SOURCE=RS  DEC[RP]
          ADDR=BUS-4(CYCLE) ;;
2: SOURCE=DHI            DEST=RAM-! ;;
3: ;;
4: SOURCE=DS  INC[DP]   ALU=B
       DECODE ;;
```

I'_@

Load With Rstack2

(→ n1)
RS(addr2 n3 → addr2 n3)

Encoding:

0xE3 5 cycles

Operation:

Fetch n1 from address addr2.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: I'_@
0: INC[RP] ;;
1: SOURCE=RS  DEC[RP]
     ADDR=BUS+0(CYCLE)  ;;
2: ;;
3: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
4: SOURCE=RD   ALU=B    DECODE ;;
```


I_+

Add Return Stack (Non-Destructive)

(n1 → n2)
RS(n3 → n3)

Encoding:

0x3A 2 cycles

Operation:

Add the top return stack element n3 to the top data stack element n1, giving n2=n1+n3. Note that the return stack element is not popped.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: I_+
0: ;;
1: SOURCE=RS     ALU=A+B     DECODE ;;
```

J

Copy Third On Return Stack

(→ n1)
RS(n1 n2 n3 → n1 n2 n3)

Encoding:

0x3C 4 cycles

Operation:

Copy the third return stack element n1 onto the data stack. In a doubly-nested DO loop, this returns the value of the index of the outer loop.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with J as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: J
0: SOURCE=DHI INC[RP]
      DEC[DP]  DEST=DS ;;
1:         INC[RP] ;;
2: SOURCE=RS  DEC[RP]      ALU=B ;;
3:         DEC[RP]              DECODE ;;
```

LEAVE

Terminate Loop

(→)
RS(n1 n2 → n2 n2)

Encoding:

0xF2 3 cycles

Operation:

Set the loop limit n1 equal to the current index n2, terminating a DO..LOOP at the end of the current iteration. Works with <loop> and <+loop>.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: LEAVE
0: SOURCE=RS    DEST=DLO ;;
1: SOURCE=DLO   INC[RP]  DEST=RS  ;;
2: DEC[RP]    DECODE ;;
```


LOAD_DS

Streamed Read Data Stack

(addr1 n2 → n.1 n.2 n.3 ... n.n2)

Encoding:

0xDB 3 + 3*count cycles

Operation:

Read n2 consecutive memory words, starting at memory location addr1, onto the data stack. This is a streamed data stack load with auto-increment. The first word read will be the deepest on the stack.

Implementation:

```
opcode: LOAD_DS
0: SOURCE=DS  INC[DP]  DEST=SBASE
    DHI[1]  ALU=0 ;;
1: SOURCE=DHI[1]   ADDR=BUS+SBASE(CYCLE)
    DHI[0]  ALU=A-1  JMP=011 ;;

( Loop for fetching )
2: SOURCE=RD  DEC[DP] DEST=DS
      DHI[0]  ALU=A-1 ;;
3: SOURCE=4   DHI[1]  ALU=A+B  JMP=10Z ;;
4: SOURCE=DHI[1]   ADDR=BUS+SBASE(CYCLE)
     JMP=010 ;;

( Finish up )
5: ;;
6: SOURCE=RD    ALU=B     DECODE ;;
```

LOAD_RS

Streamed Read Return Stack

(addr1 n2 →)
RS(→ n.1 n.2 n.3 ... n.n2)

Encoding:

0xDC 4 + 3*count cycles

Operation:

Read n2 consecutive words, starting at memory location addr1, onto the return stack. This is a streamed data stack load with auto-increment. The first word read will be deepest on the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: LOAD_RS

0: SOURCE=DS  INC[DP]  DEST=SBASE
     DHI[1]  ALU=0 ;;
1: SOURCE=DHI[1]   ADDR=BUS+SBASE(CYCLE)
     DHI[0]  ALU=A-1 JMP=011 ;;

( Loop for fetching )
2: SOURCE=RD  DEC[RP] DEST=RS
     DHI[0]  ALU=A-1 ;;
3: SOURCE=4   DHI[1]  ALU=A+B  JMP=10Z ;;
4: SOURCE=DHI[1]   ADDR=BUS+SBASE(CYCLE)
     JMP=010 ;;

( Finish up )
5: ;;
6: SOURCE=RD    DEC[RP]  DEST=RS  ;;
7: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```

LOC_!

Local Store

(n1 →) Immediate

Encoding:

0xD0 / lit 4 cycles

Operation:

Store the value n1 at the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: LOC_!
0: SOURCE=LIT
                ADDR=BUS+FP(CYCLE) ;;
1: SOURCE=DHI       DEST=RAM-! ;;
2: ;;
3: SOURCE=DS  INC[DP]   ALU=B
       DECODE ;;
```


LOC_+!

Local Add To Memory

(n1 →) Immediate

Encoding:

0xD4 / lit 7 cycles

Operation:

Performs an atomic addition to the word at the address (FP+lit) with value n1, where FP is the FP register value and lit is the compiled instruction literal value.

Implementation:

```
opcode: LOC_+!
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD  ALU=A+B  CYCLE-RAM ;;
4: SOURCE=DHI            DEST=RAM-! ;;
5: ;;
6: SOURCE=DS  INC[DP]   ALU=B       DECODE ;;
```

LOC_@

Local Load

(→ n1) Immediate

Encoding:

0xCC / lit 4 cycles

Operation:

Fetch the value n1 from the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: LOC_@
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```


LOC_@_!

Local Indirect Store

(n1 →) Immediate

Encoding:

0xD3 7 cycles

Operation:

Perform an indirect store of the value n1 at the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: LOC_@_!
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD   ADDR=BUS+0(CYCLE) ;;
4: SOURCE=DHI            DEST=RAM-! ;;
5: ;;
6: SOURCE=DS  INC[DP]   ALU=B
    DECODE ;;
```

LOC_@_+

Local Load and Add

(n1 → n2) Immediate

Encoding:

0xCF / lit 4 cycles

Operation:

Fetch the value from the effective address computed by adding the FP value to the compiled instruction literal, then add it to n1, producing n2.

Implementation:

```
opcode: FP+LIT_@_+
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD   ALU=A+B    DECODE ;;
```


LOC_@_@

Local Load Indirect

(→ n1) Immediate

Encoding:

0xD2 / lit 7 cycles

Operation:

Perform an indirect fetch of the value n1 from the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: FP+LIT_@_@
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD   ADDR=BUS+0(CYCLE) ;;
4: ;;
5: ;;
6: SOURCE=RD  ALU=B    DECODE ;;
```

LOC_B@

Local Load Byte

(→ b1) Immediate

Encoding:

0xCE 4 cycles

Operation:

Fetch the sign-extended value b1 from the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: LOC_B@
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: RAM-C@ ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD-SIGNED   ALU=B    DECODE ;;
```


LOC_C!

Local Store Character

(c1 →) Immediate

Encoding:

0xD1 / lit 4 cycles

Operation:

Store the value c1 at the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:

```
opcode: LOC_C!
0: SOURCE=LIT
       ADDR=BUS+FP(CYCLE) ;;
1: SOURCE=DHI            DEST=RAM-C! ;;
2: ;;
3: SOURCE=DS  INC[DP]   ALU=B
      DECODE ;;
```

LOC_C@

Local Load Character

(→ c1) Immediate

Encoding:

0xCD / lit 4 cycles

Operation:

Fetch the value c1 from the effective address computed by adding the FP value to the compiled instruction literal. c1 is placed in the lowest 8 bits of the stack word and padded with 0s in the high bits.

Implementation:

```
opcode: LOC_C@
0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: RAM-C@ ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```


LSLN

Logical Shift Left By N

(n1 count2 → n3)

(n1 → n3) Immediate

Encoding:

0xB3 2 + count cycles
0xB4 / lit 3 + count cycles

Operation:

Shift n1 left by count2 bits, shifting a 0 into the lowest bit, and discarding the highest bit on each shift. In the immediate variant, the count count2 is supplied by the literal field.

Implementation:

```
opcode: LSLN
0: SOURCE=DS INC[DP]   DEST=DLO
    ALU=A-1  JMP=01Z ;;

2: SL[DLO]    ALU=A-1  JMP=01Z ;;

3: SOURCE=DLO  ALU=B   DECODE ;;

opcode: LIT_LSLN
0: SOURCE=LIT  DHI[1]  ALU=B ;;
1: SOURCE=DHI[0]   DEST=DLO
     DHI[1] ALU=A-1  JMP=01Z ;;

2: SL[DLO]  DHI[1]  ALU=A-1  JMP=01Z ;;

3: SOURCE=DLO  ALU=B   DECODE ;;
```

LSRN

Logical Shift Right By N

(n1 count2 → n3)

(n1 → n3) Immediate

Encoding:

0xB6	2 + 2*count cycles
0xB7 / lit	2 + 2*count cycles

Operation:

Shift n1 right by count2 bits, shifting a 0 into the highest bit, and discarding the lowest bit on each shift (logical shift right), producing the result n3. The immediate variant uses the literal field to supply the value count2.

Implementation:

```
opcode: LSRN
0: SOURCE=DS INC[DP]
     DHI[1]  ALU=B  JMP=01Z ;;

2: DHI[0]  ALU=A-1 JMP=100 ;;
4: DHI[1]  CIN=0 ALU=A SR[ALU]
      JMP=01Z ;;

3: SOURCE=DHI[1]  ALU=B   DECODE ;;

opcode: LIT_LSRN
0: SOURCE=LIT  ALU=B DHI[1] ;;
1: DHI[1]  ALU=A-1 JMP=10Z ;;
  ( count of 0 is 3 cycles)
5:                  DECODE ;;
```



```
2: DHI[1]  ALU=A-1 JMP=100 ;;
4: DHI[0]  CIN=0 ALU=A SR[ALU]
    JMP=01Z ;;

3:                  DECODE ;;
```

LSR

Logical Shift Right

(n1 → n2)

Encoding:

0xB5 2 cycles

Operation:

Shift n1 right one bit, shifting a 0 bit into the highest order bit position (logical shift right).

Implementation:

```
opcode: LSR
0: CIN=0  SR[ALU]  ;;
1: DECODE ;;
```


M+

Add Double To Single

(d1 n2 → d3)

Encoding:

0x8F 4 cycles

Operation:

Mixed precision addition, where n2 is sign-extended to 64 bits, then added to d1, giving d3.

Implementation:

```
opcode: M+
0: SOURCE=DS  INC[DP]  DHI[1]  ALU=B ;;
1: SOURCE=DS  ALU=A+B ;;
2: SOURCE=DHI  DEST=DS   JMP=10C ;;

4: DHI[1]  ALU=A   DEST=DHI[0]  DECODE ;;
5: DHI[1]  ALU=A+1 DEST=DHI[0]  DECODE ;;
```

MOVE

Move Words

(addr1 addr2 n3 →)

Encoding:

0x8B 3 + 5*cnt cycles

Operation:

Move n3 words of memory from address addr1 to address addr2.

Notes:

An n3 value of 0 will attempt to move 4G words.

MOVE destroys the value of registers SBASE and DBASE.

This is a non-interruptible instruction. The commercial chip will have an interruptible version of this opcode.

Implementation:

```
opcode: MOVE
0: SOURCE=DS INC[DP]  DEST=DBASE ;;
1: SOURCE=DS INC[DP]  DEST=SBASE
    DHI[1]  ALU=0 ;;

( Word move loop )
2: SOURCE=DHI[1]  ADDR=BUS+SBASE(CYCLE)
     JMP=100 ;;
4:  ;;
5: SOURCE=DHI[1]  ADDR=BUS+DBASE(CYCLE)
                  ;;
6: SOURCE=RD   DEST=RAM-1
       DHI[0]  ALU=A-1 ;;
7: DHI[1]  SOURCE=4  ALU=A+B   JMP=01Z ;;

3: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```

MRAM!

Microcode RAM Store

(n1 addr2 →)

Encoding:

0x62 3 cycles

Operation:

Store value n1 at address addr2 within the MRAM memory. The highest order bits of addr2 are ignored when addressing the memory.

Implementation:

```
opcode: MRAM!
0: SOURCE=DHI  DEST=MICRO-ADR ;;
1: SOURCE=DS  INC[DP]  DEST=MRAM ;;
2: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```


MRAM@

Microcode RAM Load

(addr1 → n2)

Encoding:

0x63 3 cycles

Operation:

Fetch value n2 from address addr1 within the MRAM memory. The highest order bits of addr1 are ignored when addressing the memory.

Implementation:

```
opcode: MRAM@
0: SOURCE=DHI  DEST=MICRO-ADR ;;
( Can't put DECODE in next microinstruction in
     case of MRAM opcode )
1: SOURCE=MRAM  ALU=B  ;;
2: SOURCE=MRAM  ALU=B  DECODE ;;
```

MROM@

Microcode ROM Load

(addr1 → n2)

Encoding:

n/a 3 cycles

Operation:

Fetch value n2 from address addr1 within the MROM memory. The highest order bits of addr1 are ignored when addressing the memory.

Notes:

This opcode must be executed from MRAM to work properly, since a program many not both read MROM and execute from it at the same time. The implementation given below may be used by application programs if desired.

Implementation:

```
opcode: MROM@
0: SOURCE=DHI  DEST=MICRO-ADR ;;
( Can't put DECODE in next microinstruction in
      case of MROM opcode )
1: SOURCE=MROM  ALU=B  ;;
2: SOURCE=MROM  ALU=B  DECODE ;;
```


NEGATE

Two's Complement Negation

(n1 → n2)

Encoding:

0x84 2 cycles

Operation:

Take the two's complement of n1, returning it as n2.

Implementation:

```
opcode: NEGATE
0: ALU=notA ;;
1: ALU=A+1  DECODE ;;
```

NIP

Drop Second On Stack

(n1 n2 → n2)

Encoding:

0x77	2 cycles
0x78	1 cycle (2OPS format)

Operation:

Drop the second element on the data stack n1, putting n2 in its place and decreasing the stack size by 1 element.

Implementation:

```
opcode: NIP
0: INC[DP] ;;
1: DECODE ;;

opcode: NIP_FAST
0: INC[DP] DECODE ;;
```


NOP

No Operation

(→)

Encoding:

0x00	2 cycles
0x06	1 cycle (2OPS format)

Operation:

Do nothing, except wasting 1 or 2 clock cycles. NOP should be used to fill the opcode field for CALL, EXIT, and JNEXT instructions for which there is no desired opcode. NOP_FAST may be used as the first or second opcode of a 2OPS instruction which, for some reason, does not have two useful opcodes.

Implementation:

```
opcode: NOP
0: ;;
1: DECODE ;;
2: JMP=000 ;;
3: JMP=000 ;;
4: JMP=000 ;;
5: JMP=000 ;;
6: JMP=000 ;;
7: JMP=000 ;;

opcode: NOP_FAST
0: DECODE ;;
```

NOT

One's Complement Negation

(n1 → n2)

Encoding:

0x09 2 cycles

Operation:

Take the one's complement of n1, returning n2. This is a bitwise logical complement.

Implementation:

```
opcode: NOT
0: ALU=notA ;;
1: DECODE ;;
```


not?branch

Jump If Not Zero

(flag1 →)

Encoding:

0x98 / call 4 cycles

Operation:

Perform branch if flag1 is non-zero, otherwise continue executing in-line. ?BRANCH must be compiled as the opcode in a CALL instruction that has the branch target as its next address field.

Implementation:

```
opcode: NOT?BRANCH
0: SOURCE=RETURN-SAVE     ADDR=BUS+0(CYCLE) ;;
1: INC[RP]   JMP=01Z ;;

( Zero flag, fall through )
3: LATCH-INSTRUCTION  ;;
4: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;

( Non-zero flag, take the branch )
2: JMP=111 ;;
7: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```

ONE

One

(→ 1)

Encoding:

0x6D 2 cycles

Operation:

Pushes constant 1.

Implementation:

```
opcode: ONE
0:  DEC[DP]  DS-FROM-DHI   ALU=0 ;;
1:  ALU=A+1          DECODE ;;
```


OR

Logical Or

(n1 n2 → n3)

(n1 → n3) Immediate

Encoding:

0x3D	2 cycles
0x3E	1 cycle (2OPS format)
0xAF / lit	2 cycles
0xB0 / lit	1 cycle (2OPS format)

Operation:

Perform a bitwise logical OR operation of n1 and n2, giving n3. For the immediate variant, n2 is supplied by the literal field.

Implementation:

```
opcode: OR
0: SOURCE=DS  INC[DP]    ALU=AorB ;;
1: DECODE ;;

opcode: OR_FAST
0: SOURCE=DS INC[DP]  ALU=AorB  DECODE ;;

opcode: LIT_OR
0: SOURCE=LIT  ALU=AorB  ;;
1: DECODE ;;

opcode: LIT_OR_FAST
0: SOURCE=LIT  ALU=AorB  DECODE ;;
```

OR!

Logical Or To Memory

(n1 addr2 →)

Encoding:

0xA0	7 cycles

Operation:

Perform an atomic bitwise logical OR of n1 to the word at memory addr2, returning the result to addr2 (similar to +!, but with an OR operation instead of a + operation).

Implementation:

```
opcode: OR!
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW ;;
2: SOURCE=DS INC[DP]   ALU=B  ;;
3: SOURCE=RD   ALU=AorB    CYCLE-RAM ;;
4: SOURCE=DHI  DEST=RAM-! ;;
5: ;;
6: SOURCE=DS  INC[DP]  ALU=B    DECODE ;;
```


OVER

Copy Second On Stack

(n1 n2 → n1 n2 n1)

Encoding:

0x3F	2 cycles
0x40	1 cycle (2OPS format)

Operation:

Copy the second element on the data stack n1 to the top.

Implementation:

```
opcode: OVER
0: SOURCE=DS  ALU=B
       DEC[DP] DS-FROM-DHI ;;
1: DECODE ;;

opcode: OVER_FAST
0: SOURCE=DS  ALU=B
    DEC[DP]  DS-FROM-DHI   DECODE ;;
```

OVER_!

Reversed Store (Non-Destructive)

(addr1 n2 → addr1)

Encoding:

0xB8 4 cycles

Operation:

Store the value n2 at address addr1, without popping addr1.

Implementation:

```
opcode: OVER_!
0: SOURCE=DS   ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI  DEST=RAM-! ;;
2: ;;
3: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


OVER_+

Add (Non-Destructive to Second Element)

(n1 n2 → n1 n3)

Encoding:

0xBA 2 cycles

Operation:

Add n1 and n2, giving n3, but without destroying n1.

Implementation:

```
opcode: OVER_+
0: SOURCE=DS  ALU=A+B ;;
1: DECODE ;;
```

OVER_@

Fetch Using Second On Stack (Non-Destructive)

(addr1 n2 → addr1 n2 n3)

Encoding:

0xB9 4 cycles

Operation:

Fetch value n3 from addr1, pushing it onto the stack.

Implementation:

```
opcode: OVER_@
0: SOURCE=DS  ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS ;;
3: SOURCE=RD  ALU=B       DECODE ;;
```


R+!

Add To Rstack Immediate

(→)
RS(n1 → n2)

Encoding:

0xC1 4 cycles

Operation:

Pop n1 from the return stack, add to it the compiled instruction literal field, and place the result back onto the return stack as n2.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R+!
0: SOURCE=LIT  DHI[1]  ALU=B ;;
1: SOURCE=RS   DHI[1]  ALU=A+B ;;
2: SOURCE=DHI[1]  DEST=RS ;;
3: DECODE ;;
```

PICK

Push Nth On Stack

(n.m ... n.1 m → n.m ... n.1 n.m)

Encoding:

0x1C 5 cycles

Operation:

Copies the mth value from the data stack to the top of stack. <PICK> is a primitive that does not check for stack underflow, nor for negative input numbers.

Notes:

"0 <PICK>" is a no-op. "1 <PICK>" is equivalent to "DUP". "2 <PICK>" is equivalent to "OVER".

This primitive is dangerous in an environment where stack memory is being spilled into program memory. It will not check to see if the input m is too deep in the stack, nor will it generate a stack underflow interrupt if m is big enough to cause a wrap-around back into the active stack space.

Implementation:

```
opcode: PICK
0: SOURCE=DP   DEST=DLO   ALU=A+B  ;;
1: SOURCE=DHI   DEST=DP ;;
2: ;;
3: SOURCE=DLO   DEST=DP ;;
```



```
( Exploits fact that DSREG maintains value while
     waiting for the new DP to access the DS RAM.
     )
4: SOURCE=DS   ALU=B   DECODE ;;
```

R>

Transfer From Rstack to Dstack

(→ n1)
RS(n1 →)

Encoding:

0x41	2 cycles
0x42	1 cycle (2OPS format)

Operation:

Pop n1 from the return stack, placing it on the data stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with R> as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

```
opcode: R>
0:  SOURCE=DHI   DEC[DP] DEST=DS ;;
1:  SOURCE=RS  INC[RP] ALU=B DECODE ;;

opcode: R>_FAST
0:  DEC[DP]  DS-FROM-DHI
```



```
SOURCE=RS  INC[RP]    ALU=B  DECODE ;;
```

R>_!

Store With Return Stack

(n1 →)
RS(addr2 →)

Encoding:

0xBF 4 cycles

Operation:

Store value n1 at location addr2, where addr2 resides on the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R>_!
0: SOURCE=RS
            ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI              DEST=RAM-! ;;
2: INC[RP] ;;
3: SOURCE=DS  INC[DP]   ALU=B
        DECODE ;;
```


R>_@

Load With Return Stack

(→ n1)
RS(addr2 →)

Encoding:

0xC0 4 cycles

Operation:

Fetch value n1 from location addr2, where addr2 resides on the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R>_@
0: SOURCE=RS    ADDR=BUS+0(CYCLE)  ;;
1: INC[RP] ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```

R@

Copy Top Of Rstack To Dstack

(→ n1)
RS(n1 → n1)

Encoding:

0x38	2 cycles
0x39	1 cycle (2OPS format)

Operation:

Copy the top return stack element onto the data stack.

Notes:

This is the same opcode used to access the inner loop index (the I word in Forth).

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: I
0: SOURCE=DHI     DEC[DP]  DEST=DS ;;
1: SOURCE=RS      ALU=B           DECODE ;;

opcode: I_FAST
0: DEC[DP]  DS-FROM-DHI
   SOURCE=RS      ALU=B           DECODE ;;
```


R@_!

Store With Return Stack (Non-Destructive)

(n1 →)
RS(addr2 → addr2)

Encoding:

0xBD	4 cycles

Operation:

Store n1 at location addr2, where addr2 is copied from the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R@_!
0: SOURCE=RS
     ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI      DEST=RAM-! ;;
2: ;;
3: SOURCE=DS  INC[DP]   ALU=B
      DECODE ;;
```

R@_4-_!

Store With Return Stack Minus 4 (Non-Destructive)

(n1 →)
RS(addr2 → addr2)

Encoding:

0xE4 4 cycles

Operation:

Store n1 at location addr2-4, where addr2 is copied from the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R@_4-_!
0: SOURCE=RS        ADDR=BUS-4(CYCLE) ;;
1: SOURCE=DHI             DEST=RAM-! ;;
2: ;;
3: SOURCE=DS  INC[DP]   ALU=B
     DECODE ;;
```


R@_@

Load With Return Stack (Non-Destructive)

(→ n1)
RS(addr2 → addr2)

Encoding:

0xBE 4 cycles

Operation:

Fetch n1 from location addr2, where addr2 is copied from the return stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: R@_@
0: SOURCE=RS     ADDR=BUS+0(CYCLE)  ;;
1: ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```

RLC

Rotate Left Through Carry

(n1 flag2 → n3 flag4)

Encoding:

0xBB 4 cycles

Operation:

Perform a one-bit rotate left, using flag2 as the carry-in value (true or false), and n1 as the input number. If flag2 is true, a 1 is rotated into n1, otherwise a 0 is rotated into n1. flag4 is set to true if the bit rotated out of n1 (i.e. bit 31 of n1) is 1, otherwise flag4 is set to zero. The carry flag participates as a 33rd bit in the rotation.

Implementation:

```
opcode: RLC
0: SOURCE=DS  ALU=B        JMP=01Z ;;

2: ( cin <>0 )  ALU=A+A+1 JMP=001 ;;
3: ( cin = 0 )  ALU=A+A    JMP=001 ;;

1: SOURCE=DHI  DEST=DS       JMP=10C ;;

4: ( cout = 0 ) ALU=0      DECODE ;;
5: ( cout <>0 ) ALU=-1     DECODE ;;
```


ROLL

Get Nth On Stack

(n.m n.x ... n.1 m → n.x ... n.1 n.m)

Operation:

0x1D 5+2*m cycles

Operation:

Moves the mth value from the data stack to the top of stack. <PICK> is a primitive that does not check for stack underflow, nor for negative input numbers.

Notes:

"0 <ROLL>" and "1 <ROLL>" are no-ops. "1 <ROLL>" is equivalent to "SWAP". "2 <ROLL>" is equivalent to "ROT".

This primitive is dangerous in an environment where stack memory is being spilled into program memory. It will not check to see if the input m is too deep in the stack, nor will it generate a stack underflow interrupt if m is big enough to cause a wrap-around back into the active stack space.

Implementation:

```
opcode: <ROLL>
0: SOURCE=DP  DEST=DLO  ;;
( Roll loop )
1: DS-FROM-DHI
   SOURCE=DS  DHI[1]  ALU=B   JMP=01Z ;;
2: INC[DP]  DHI[0]  ALU=A-1  JMP=001 ;;
```

```
( Done with loop )
3: SOURCE=DLO  DEST=DP ;;
4: ;;
5: SOURCE=DHI[1]  DHI[0]  ALU=B
          INC[DP]  DECODE ;;
```


ROT

Get Third Stack Element

(n1 n2 n3 → n2 n3 n1)

Encoding:

0x43 3 cycles

Operation:

Rotate the third stack element n1 to the top.

Implementation:

```
opcode: ROT
0: SOURCE=DS  INC[DP]    DHI[1]  ALU=B ;;
1: DHI[1] DS-FROM-DHI
    SOURCE=DS  ALU=B ;;
2: DEC[DP]  DHI[0] DS-FROM-DHI
    SOURCE=DHI[1]  ALU=B    DECODE ;;
```

RP!

Put Return Stack Pointer

(n1 →)

Encoding:

0x0F 3 cycles

Operation:

Store the lowest six bits of n in the hardware return stack pointer.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: RP!
0: SOURCE=DHI DEST=RP ;;
1: ;;
2: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


RP@

Get Return Stack Pointer

(→ n1)

Encoding:

0x10 2 cycles

Operation:

Get n, the six-bit return stack pointer value.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: RP@
0: SOURCE=RP DEST=DLO ;;
1: DEC[DP]  DS-FROM-DHI
    SOURCE=DLO ALU=B  DECODE ;;
```

RPLIM!

Put RP Limit

(n1 →)

Encoding:

0x64 2 cycles

Operation:

Store n1 in the return stack pointer limit register. Bits 0-5 of n1 are stored in the lower limit register, while bits 16-20 are stored in the upper limit register. If the value of RP ever exceeds the upper limit register value, or is less than the lower limit register value, a stack overflow/underflow interrupt is generated.

*** Picture here ***

Implementation:

```
opcode: RPLIM!
0: SOURCE=DHI   DEST=RP-LIMIT  ;;
1: SOURCE=DS  INC[DP]   ALU=B   DECODE ;;
```


RPLIM@

Get RP Limit Register

(→ n1)

Encoding:

0x65 2 cycles

Operation:

Fetch n1 from the return stack pointer limit register. Bits 0-5 of n1 are from the lower limit register, while bits 16-20 are from the upper limit register.

*** Picture here ***

Implementation:

```
opcode: RPLIM@
0: DEC[DP]  DS-FROM-DHI
    SOURCE=RP-LIMIT   ALU=B  ;;
1: DECODE ;;
```

RRC

Rotate Right Carry

(n1 flag2 → n3 flag4)

Encoding:

0xBC 4 cycles

Operation:

Perform a one-bit rotate right, using flag2 as the carry-in value (true or false), n1 as the input number, and n3 as the output number. If flag2 is true, a 1 is rotated into n1, otherwise a 0 is rotated into n1. flag4 is set to true if the bit rotated out of n1 (i.e. bit 0 of n1) is 1, otherwise flag4 is set to zero. In this opcode the carry flag acts as a 33rd bit for the rotation.

Implementation:

```
opcode: RRC
0: SOURCE=DS  ALU=B  DEST=DLO  JMP=01Z ;;

2: ( cin <>0 ) CIN=1  SR[ALU]  JMP=001 ;;
3: ( cin = 0 ) CIN=0  SR[ALU]  JMP=001 ;;

1: SOURCE=DHI  DEST=DS       JMP=10L ;;

4: ( cout = 0 )  ALU=0     DECODE ;;
5: ( cout <>0 )  ALU=-1    DECODE ;;
```


RTI

Return From Interrupt

(n1 →)
RS(addr2 →)

Encoding:

0x44 4 cycles

Operation:

Return from interrupt. n1 is written to the config register (it is the user's responsibility to ensure that this clears the desired stack underflow/overflow interrupt bits and clears the interrupt mask if desired). The return stack has a value addr2 that points to the restart address plus 4.

Implementation:

```
opcode: RTI
0: SOURCE=RS  INC[RP]
       ADDR=BUS-4(CYCLE) ;;
1: SOURCE=DHI  DEST=CONFIG  ;;
2: LATCH-INSTRUCTION ;;
3: SOURCE=DS  INC[DP]  ALU=B
           DECODE ;;
```

S>D

Extend From Single To Double

(n1 → d2)

Encoding:

0x83 2 cycles

Operation:

Sign extend n1, creating d2.

Implementation:

```
opcode: S>D
0: JMP=01S ;;

2: DEC[DP]  DS-FROM-DHI  ALU=0  DECODE ;;
3: DEC[DP]  DS-FROM-DHI  ALU=-1 DECODE ;;
```


SBASE!

Set SBASE Register

(n1 →)

Encoding:

0x66 2 cycles

Operation:

Store value n1 in register SBASE.

Implementation:

```
opcode: SBASE!
0: SOURCE=DHI  DEST=SBASE ;;
1: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```

SBASE+!

Add To SBASE

(n1 →)

Encoding:

0xEA 3 cycles

Operation:

Add the value n1 to the contents of SBASE, and place the sum back in the SBASE register.

Implementation:

```
opcode: SBASE+!
0: SOURCE=SBASE  ALU=A+B ;;
1: SOURCE=DHI  DEST=SBASE ;;
2: SOURCE=DS  INC[DP]  ALU=B  DECODE ;;
```


SBASE+_!

Store Indexed With SBASE

(n1 offset2 →)

Encoding:

0xD6 4 cycles

Operation:

Store value n1 at address computed by adding offset2 to the value of SBASE.

Implementation:

```
opcode: SBASE+_!
0: SOURCE=DHI  ADDR=BUS+SBASE(CYCLE)  ;;
1: SOURCE=DS INC[DP]  DEST=RAM-! ;;
2: ;;
3: SOURCE=DS INC[DP]  ALU=B  DECODE ;;
```

SBASE+_@

Load Indexed With SBASE

(offset1 → n2)

Encoding:

0xD5 4 cycles

Operation:

Fetch value n2 from address computed by adding offset1 to the value of SBASE.

Implementation:

```
opcode: SBASE+_@
0: SOURCE=DHI  ADDR=BUS+SBASE(CYCLE)  ;;
1: ;;
2: ;;
3: SOURCE=RD   ALU=B    DECODE ;;
```


SBASE@

Get From SBASE

(→ n1)

Encoding:

0x67 2 cycles

Operation:

Fetch the value n1 from register SBASE.

Implementation:

```
opcode: SBASE@
0: DEC[DP]   DS-FROM-DHI
    SOURCE=SBASE  ALU=B ;;
1: DECODE ;;
```

SP!

Put Data Stack Pointer

(n1 →)

Encoding:

0x0D 3 cycles

Operation:

Store the lowest six bits of n1 in the hardware data stack pointer.

Implementation:

```
opcode: SP!
0: SOURCE=DHI DEST=DP ;;
1: ;;
2: SOURCE=DS  INC[DP]   ALU=B
          DECODE ;;
```


SP@

Get Data Stack Pointer

(→ n1)

Encoding:

0x0E 2 cycles

Operation:

Get n1, the six-bit data stack pointer value.

Implementation:

```
opcode: SP@
0: SOURCE=DP   DEST=DLO ;;
1: DEC[DP]  DS-FROM-DHI
   SOURCE=DLO ALU=B  DECODE ;;
```

SPLIM!

Get Data Stack Pointer Limits

(n1 →)

Encoding:

0x5E 2 cycles

Operation:

Store n1 in the data stack pointer limit register. Bits 0-5 of n1 are stored in the lower limit register, while bits 16-20 are stored in the upper limit register. If the value of DP ever exceeds the upper limit register value, or is less than the lower limit register value, a stack overflow/underflow interrupt is generated.

** PICTURE **

Implementation:

```
opcode: SPLIM!
0: SOURCE=DHI   DEST=DP-LIMIT  ;;
1: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


SPLIM@

Put Data Stack Pointer Limit

(→ n1)

Encoding:

0x5F 2 cycles

Operation:

Fetch n1 from the data stack pointer limit register. Bits 0-5 of n1 are from the lower limit register, while bits 16-20 are from the upper limit register.

** PICTURE **

Implementation:

```
opcode: SPLIM@
0: DEC[DP]  DS-FROM-DHI
      SOURCE=DP-LIMIT   ALU=B  ;;
1: DECODE ;;
```

STORE_DS

Streamed Write Data Stack

(n.1 n.2 ... n.count addr1 ncount2 →)

Encoding:

0xDD 3 + 3*count cycles

Operation:

Write ncount2 consecutive words, starting *backwards* at memory location addr1, from the data stack. This is a streamed data stack store with auto-decrement.

Implementation:

```
opcode: STORE_DS
0: SOURCE=DS  INC[DP]  DEST=DBASE
    DHI[1]  ALU=0 ;;
( Count in DHI[0], offset in DHI[1] )
1: SOURCE=DHI[1]
       ADDR=BUS+DBASE(CYCLE)  ;;

( Loop for fetching )
2: SOURCE=DS  INC[DP] DEST=RAM-!
       DHI[0]  ALU=A-1 ;;
3: SOURCE=4  DHI[1]  ALU=A-B  JMP=10Z ;;
4: SOURCE=DHI[1]   ADDR=BUS+DBASE(CYCLE)
     JMP=010 ;;

( Finish up )
5: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


STORE_RS

Streamed Write Return Stack

(addr1 ncount2 →)
(RS: n.1 n.2 ... n.count →)

Encoding:

0xDE 3 + 3*count cycles

Operation:

Write ncount2 consecutive words, starting *backwards* at memory location addr1, from the return stack. This is a streamed return stack store with auto-decrement.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

```
opcode: STORE_RS
0: SOURCE=DS  INC[DP]  DEST=DBASE
    DHI[1]  ALU=0 ;;
( Count in DHI[0], offset in DHI[1] )
1: SOURCE=DHI[1]
       ADDR=BUS+DBASE(CYCLE)  ;;

( Loop for fetching )
2: SOURCE=RS  INC[RP] DEST=RAM-!
      DHI[0]  ALU=A-1 ;;
3: SOURCE=4   DHI[1]  ALU=A-B  JMP=10Z ;;
4: SOURCE=DHI[1]   ADDR=BUS+DBASE(CYCLE)
      JMP=010 ;;
```

```
( Finish up )
5: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


SWAP

Get Second On Stack

(n1 n2 → n2 n1)

Encoding:

0x45	2 cycles
0x46	1 cycles (2OPS format)

Operation:

Swap the top two stack elements.

Implementation:

```
opcode: SWAP
0: SOURCE=DS ALU=B    DS-FROM-DHI ;;
1: DECODE ;;

opcode: SWAP_FAST
0: SOURCE=DS ALU=B
    DS-FROM-DHI  DECODE ;;
```

SWAP_!

Reversed Store

(addr1 n2 →)

Encoding:

0xE6 4 cycles

Operation:

Store n2 at address addr1.

Implementation:

```
opcode: SWAP_!
0: SOURCE=DS INC[DP]
       ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI  DEST=RAM-! ;;
2: ;;
3: SOURCE=DS INC[DP]   ALU=B   DECODE ;;
```


SWAP_-

Reverse Subtract

(n1 n2 → n3)

Encoding:

0xE7 2 cycles

Operation:

Subtract n1 from n2, giving n3.

Implementation:

```
opcode: SWAP_-
0: SOURCE=DS INC[DP]  ALU=A-B  ;;
1: DECODE ;;
```

SWAP_-_!

Store With Subtracted Index

(n1 n2 addr3 →)

Encoding:

0xFD 5 cycles

Operation:

Subtract offset n2 from addr3, then store n1 at the resulting address.

Implementation:

```
opcode: SWAP_-_!
0: SOURCE=DS  INC[DP]   ALU=A-B ;;
1: SOURCE=DHI
     ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS  INC[DP]    DEST=RAM-! ;;
3: ;;
4: SOURCE=DS  INC[DP]   ALU=B
     DECODE ;;
```


SWAP_-_@

Load With Subtracted Index

(n1 addr2 → n3)

Encoding:

0xF7 5 cycles

Operation:

Subtract offset n1 from addr2, then fetch n3 from the resulting address.

Implementation:

```
opcode: SWAP_-_@
0: SOURCE=DS  INC[DP]   ALU=A-B ;;
1: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
2: ;;
3: ;;
4: SOURCE=RD   ALU=B    DECODE ;;
```

SWAP_-_C!

Store Character With Subtracted Index

(c1 n2 addr3 →)

Encoding:

0xFE 5 cycles

Operation:

Subtract offset n2 from addr3, then store c1 at the resulting address.

Implementation:

```
opcode: SWAP_-_C!
0: SOURCE=DS  INC[DP]   ALU=A-B ;;
1: SOURCE=DHI
       ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS  INC[DP]    DEST=RAM-C! ;;
3: ;;
4: SOURCE=DS  INC[DP]   ALU=B
        DECODE ;;
```


SWAP_-_C@

Load Character With Subtracted Index

(n1 addr2 → c3)

Encoding:

0xF8 5 cycles

Operation:

Subtract offset n1 from addr2, then fetch c3 from the resulting address.

Implementation:

```
opcode: SWAP_-_C@
0: SOURCE=DS  INC[DP]   ALU=A-B ;;
1: SOURCE=DHI   ADDR=BUS+0(CYCLE)  ;;
2: RAM-C@ ;;
3: ;;
4: SOURCE=RD   ALU=B    DECODE ;;
```

ROT_+

Get Second On Stack, Add Immediate

(n1 n2 → n2 n3) Immediate

Encoding:

0xC2 / lit 2 cycles

Operation:

Add the compiled instruction literal field to n1, leaving n3. Note that n2 moves from the top of the stack to the second element on the stack.

Implementation:

```
opcode: LIT_ROT_+
0: DS-FROM-DHI
    SOURCE=DS  ALU=B  ;;
1: SOURCE=LIT  ALU=A+B  DECODE ;;
```


SWAP_OVER_!

Store (Non-Destructive)

(n1 addr2 → addr2)

Encoding:

0xAA 4 cycles

Operation:

Store the value n1 at location addr2, without destroying addr2.

Implementation:

```
opcode: SWAP_OVER_!
0: SOURCE=DHI
       ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS  INC[DP]    DEST=RAM-! ;;
2: ;;
3: DECODE ;;
```

TEST_AND_SET

Test And Set

(n1 addr2 → n3 tflag)

... → fflag)

Encoding:

0xCA 7 cycles

Operation:

Fetch the value n3 from word addr2, then perform a logical OR with n1, and write the result back to addr2. If any of the bits set in n1 were already set at location addr2, return n3 and a true flag. If none of the bits set in n1 were set at location addr2 before the OR operation, return a false flag. The operation is atomic, using the read/modify/write bus protocol. Useful as a synchronization instruction.

Implementation:

```
opcode: TEST_AND_SET
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW    SOURCE=DS  DHI[1]  ALU=B ;;
2: SOURCE=DS INC[DP]  ALU=B  ;;
3: SOURCE=RD   DEST=DLO   ALU=AorB
           CYCLE-RAM ;;
4: SOURCE=DHI  DEST=RAM-!  ;;
5: SOURCE=DLO  DHI[1] ALU=AandB  ;;
6: SOURCE=DLO  ALU=B  INC[MPC] JMP=11Z ;;
next opcode
6: SOURCE=DHI  DEC[DP] DEST=DS
     ALU=-1  DECODE ;;
7: ALU=0   DECODE ;;
```

TEST_UNDER_MASK

Test Under Mask

(n1 n2 → flag3)

Encoding:

0xC7 3 cycles

Operation:

Test value n1 under mask n2. Returns a true flag if n1 AND n2 is non-zero, otherwise returns a false flag. In other words, a true flag is returned if any of the bits set in n1 correspond to set bits of n2.

Implementation:

```
opcode: TEST_UNDER_MASK
0: SOURCE=DS INC[DP]  ALU=AandB ;;
1: JMP=01Z ;;

2: ALU=-1 DECODE ;;
3: ALU=0  DECODE ;;
```


TRUE

Push True Flag

(→ -1)

Encoding:

0x13 2 cycles

Operation:

Push the value -1 on the stack. -1 is the "true" flag value for the machine.

Implementation:

```
opcode: TRUE
0: DEC[DP] DS-FROM-DHI   ALU=-1 ;;
1: DECODE ;;
```

TUCK

Put Top As Second (Non-Destructive)

(n1 n2 → n2 n1 n2)

Encoding:

0x79 2 cycles

Operation:

Copy the top stack element n2 under the second stack element n1.

Implementation:

```
opcode: TUCK
0: SOURCE=DS  ALU=B    DS-FROM-DHI  ;;
1: SOURCE=DS  ALU=B
     DEC[DP]  DS-FROM-DHI   DECODE ;;
```


U>

Test For Unsigned Greater Than

(u1 u2 → flag3)

Encoding:

0x47 3 cycles

Operation:

Perform an unsigned comparison of u1 to u2, returning a true flag3 only if u1 > u2 is satisfied.

Implementation:

```
opcode: U>
0: SOURCE=DS  INC[DP]   ALU=A-B ;;
( if A-B results in borrow, then true )
1: JMP=01C ;;

2: ALU=0  DECODE ;;
3: ALU=-1 DECODE ;;
```

UDNORMALIZE

Unsigned Double Precision Normalize

(dmant1 exp1 → dmant2 exp2)

Encoding:

0xDA 5 + 2*x cycles

Operation:

Normalize dmant1 so that its most significant 1 bit is in bit position 63, giving dmant2. Add one to exp1 for every right shift required for normalization, subtract one from exp1 for every left shift required for normalization, giving exp2.

Notes:

Does *not* check for zero input, and will infinitely loop if given one.

Implementation:

```
opcode: UDNORMALIZE
( shifts left until high bit set, then shifts right
      one bit, )
(   to normalize top bit in bit 30 position. )
( DOES *NOT* check for zero input )
( Exponent in DHI0, Mantissa in DHI1 )
0: SOURCE=DS  INC[DP]    DHI[1]  ALU=B ;;
( Perform zero check & pre-adjust
      exponent )
1:  SOURCE=DS    DEST=DLO      DHI[0]    ALU=A+1
      JMP=10S ;;

( Normalization loop )
2: DHI[0] ALU=A-1  JMP=10S ;;
4: DHI[1] ALU=A SL[ALU]  SL[DLO]
       JMP=010 ;;

( Done, unshift )
5: DHI[1]  CIN=0  SR[ALU] SR[DLO] ;;
6: SOURCE=DLO     DEST=DS ;;
7: SOURCE=DHI[1]  DEC[DP] DEST=DS
       DECODE ;;
```

UM*

Unsigned Multiply

(u1 u2 → ud3)

Encoding:

0x48 36 cycles

Operation:

Perform a 32x32 bit unsigned multiply, giving a 64 bit unsigned product.

Implementation:

```
opcode: UM*
( Multiplier in DS,  Multiplicand in DLO )
( DHI initialized to 0 )
0: SOURCE=DHI   DEST=DLO ALU=0  ;;
( Initial bit shifts )
1: SR[DLO] ;;
2: SR[DLO]      JMP=10L ;;

( Every pair of microinstructions does one bit)
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 2-5 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 6-9 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 10-13 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
```

```
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 14-17 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 18-21 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
```



```
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 22-25 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 26-29 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=10L ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=10L ;;
4: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=11L ;;
5: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=11L ;;
6: MULTIPLY-STEP  ALU=A+0
```

```
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
7: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=00L  INC[MPC] ;;
next opcode: ( bits 30-31 )
0: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=01L ;;
1: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=01L ;;
2: MULTIPLY-STEP  ALU=A+0
    SR[ALU] SR[DLO]  JMP=100 ;;
3: MULTIPLY-STEP  ALU=A+B
    SR[ALU] SR[DLO]  JMP=100 ;;
4:          SOURCE=DLO                    DEST=DS
      DECODE ;;
```


UM/MOD

Unsigned Division

(uddivdnd undivsr → unrem unquot)

Encoding:

0x51 37 cycles

Operation:

Perform a 64/32 bit unsigned division, giving a 32 bit unsigned remainder and a 32 bit unsigned quotient.

Implementation:

```
opcode: UM/MOD
( DHI:DLO  is dividend,  DS is divisor )
0: SOURCE=DS  INC[DP]      DHI[1] ALU=B ;;
1: DS-FROM-DHI             DHI[0]
    SOURCE=DS  DEST=DLO  ;;
( Initial subtraction )
2: DIVIDE  DHI[1]  DEST=DHI[0]   SL[ALU]  SL[DLO]
     ;;

( Iterated step-divisions bits 0-4 )
3: DIVIDE  SL[DLO]  SL[ALU]  ;;
4: DIVIDE  SL[DLO]  SL[ALU]  ;;
5: DIVIDE  SL[DLO]  SL[ALU]  ;;
6: DIVIDE  SL[DLO]  SL[ALU]  ;;
7: DIVIDE  SL[DLO]  SL[ALU]
     INC[MPC] JMP=000 ;;
next opcode ( bits 5-12 )
0: DIVIDE  SL[DLO]  SL[ALU]  ;;
1: DIVIDE  SL[DLO]  SL[ALU]  ;;
2: DIVIDE  SL[DLO]  SL[ALU]  ;;
3: DIVIDE  SL[DLO]  SL[ALU]  ;;
```

```
4: DIVIDE  SL[DLO]  SL[ALU]  ;;
5: DIVIDE  SL[DLO]  SL[ALU]  ;;
6: DIVIDE  SL[DLO]  SL[ALU]  ;;
7: DIVIDE  SL[DLO]  SL[ALU]
      INC[MPC] JMP=000 ;;
next opcode ( bits 13-20 )
0: DIVIDE  SL[DLO]  SL[ALU]  ;;
1: DIVIDE  SL[DLO]  SL[ALU]  ;;
2: DIVIDE  SL[DLO]  SL[ALU]  ;;
3: DIVIDE  SL[DLO]  SL[ALU]  ;;
4: DIVIDE  SL[DLO]  SL[ALU]  ;;
5: DIVIDE  SL[DLO]  SL[ALU]  ;;
6: DIVIDE  SL[DLO]  SL[ALU]  ;;
7: DIVIDE  SL[DLO]  SL[ALU]
      INC[MPC] JMP=000 ;;
next opcode ( bits 21-28 )
0: DIVIDE  SL[DLO]  SL[ALU]  ;;
1: DIVIDE  SL[DLO]  SL[ALU]  ;;
2: DIVIDE  SL[DLO]  SL[ALU]  ;;
3: DIVIDE  SL[DLO]  SL[ALU]  ;;
4: DIVIDE  SL[DLO]  SL[ALU]  ;;
5: DIVIDE  SL[DLO]  SL[ALU]  ;;
6: DIVIDE  SL[DLO]  SL[ALU]  ;;
7: DIVIDE  SL[DLO]  SL[ALU]
      INC[MPC] JMP=000 ;;
next opcode ( bits 29-31 )
0: DIVIDE  SL[DLO]  SL[ALU]  ;;
1: DIVIDE  SL[DLO]  SL[ALU]  ;;
2: DIVIDE  SL[DLO]  ;;

 ( Final divide step is implicit ) ;;
3: SOURCE=DS ;;
4: DS-FROM-DHI   SOURCE=DLO
       ALU=B      DECODE ;;
```


UNORMALIZE

Unsigned Normalize

(mant1 exp1 → mant2 exp2)

Encoding:

0xD9 3/5+2*N cycles

Operation:

Normalize mant1 so that its most significant 1 bit is in bit position 30, giving mant2. Add one to exp1 for every right shift required for normalization, subtract one from exp1 for every left shift required for normalization, giving exp2. Takes 3 clock cycles if the input is zero.

Implementation:

```
opcode: UNORMALIZE
( shifts left until high bit set, then shifts right
      one bit to normalize top bit in bit 30
      position. )
( Exponent in DHI0, Mantissa in DHI1 )
0: SOURCE=DS  DHI[1]  ALU=B ;;
( Perform zero check &
    pre-adjust exponent )
1: DHI[1]          JMP=01Z ;;

2: DHI[0] ALU=A+1  JMP=10S ;;

( Normalization loop )
7: DHI[0] ALU=A-1  JMP=10S ;;
4: DHI[1] ALU=A+A  JMP=111 ;;

( Done, unshift )
```

```
5: DHI[1]  CIN=0  SR[ALU]  ;;
6: SOURCE=DHI[1]  DEST=DS  DECODE ;;

( Zero input – force clean zero )
3: ALU=0   DECODE ;;
```


WAIT

Wait For Interrupt

(→)

Encoding:

0x68 2 or more cycles

Operation:

Halt until an unmasked interrupt is recognized, then continue.

Implementation:

```
opcode: WAIT
0: JMP=00P
1: DECODE ;;
```

WFILL

Fill Memory

(addr1 count2 n3 →)

Encoding:

0x8D 4 + 3*count cycles

Operation:

Fill count2 words of memory with value n3, starting at address addr1 and counting up.

Notes:

Destroys value in the DBASE register.

Implementation:

```
opcode: WFILL
( DHI[1] = OFFSET & COUNT*4 )
0: SOURCE=0  DEST=DLO  DHI[1]  ALU=-1  ;;
1: SOURCE=DS INC[DP]
     DHI[1]  ALU=A+B  SL[ALU]  JMP=111 ;;
7: SOURCE=DS INC[DP]   DEST=DBASE
      DHI[1]  ALU=A+A   JMP=010 ;;

( Store words )
2: SOURCE=DHI[1]   ADDR=BUS+DBASE(CYCLE)
      JMP=100 ;;
4: SOURCE=DHI[0]  DEST=RAM-!    DHI[1] ;;
5: SOURCE=4    DHI[1] ALU=A-B
                   JMP=01Z ;;

3: SOURCE=DS  INC[DP]  ALU=B   DECODE ;;
```


XOR

Exclusive Or

(n1 n2 → n3)

Encoding:

0x56 2 cycles
0x57 1 cycle (2OPS format)

Operation:

Perform a bitwise logical eXclusive OR function on n1 and n2, returning n3.

Implementation:

```
opcode: XOR
0: SOURCE=DS  INC[DP]   ALU=AxorB  ;;
1: DECODE ;;

opcode: XOR_FAST
0: SOURCE=DS  INC[DP]
       ALU=AxorB   DECODE ;;
```