

Parallax Propeller 2 Assembly Instruction Set

Here you find the instruction set for the new P2 (2015) chip. Please feel free to edit this document and if something requires more explanation or examples then just link that to another section of the document. The emphasis is mainly on the instruction set and memory map while [Chip's document](#) provides the overview and many other details. Please refer to his document for more information about the Propeller 2 chip itself.

[<click here for published version>](#)

CONTENTS

[LINKS](#)

[LABELS](#)

[EXPRESSIONS](#)

[ADDRESSING](#)

[P2 MEMORY MAP](#)

[EXEC MAP](#)

[COG REGISTERS](#)

[LUT](#)

[HUB](#)

[HUB ROM](#)

[P2 INTERNAL STACK](#)

[Conditional execution codes table](#)

[INSTRUCTION BIT-FIELD SYMBOLS](#)

[P2 INSTRUCTIONS LIST](#)

[SHIFTS ROTATES](#)

[ARITHMETIC](#)

[LOGICAL](#)

[INSTRUCTION MODIFIERS](#)

[COG NIBBLE/BYTE/WORD Operations](#)

[BRANCHING](#)

[CALL REGISTER*i*199*i*](#)

[CALL LONG](#)

[LUT MEMORY](#)

[Example: Create stacks in LUT memory.](#)

[HUB MEMORY](#)

[SMART PINS](#)

[COG and HUB CONTROL](#)

[CORDICbb](#)

[EVENTS, WAITS and INTERRUPTS](#)

[NOTES](#)

[SETTING EDGE EVENTS](#)

[ALIASES](#)

[POINTER ADDRESSING MODES](#)

[Examples:](#)

[HUB MEMORY READING AND WRITING](#)

[STREAMER](#)

[ALTDS](#)

[ALTDS Examples](#)

[copy 16 cog regs from .src to .dest](#)

[PUSHZC ??? - Old P2 hot instruction ???](#)

[RDFAST](#)

[P2 INTERNAL STACK](#)

[MAILBOXES AND DEBUG INTERRUPT VECTORS](#)

[Migrating From Propeller 1](#)

[Instruction Changes](#)

[Removed Instructions/Registers/Effects](#)

[Experimenting with different document layouts](#)

LINKS

[Link to Chip's P2 document](#)

[Link to PBJ's opcode testing \(pubdocs version\)](#)

Mindrobot's [P2 memory-map architecture spreadsheet](#)

Discussion about LUT to HUB flow is [here](#)

LABELS

- Labels are either globally-scoped or locally-scoped.
- A globally-scoped label must begin with an underscore or letter (a-z, A-Z). All other characters must be an underscore, letter (a-z, A-Z) or number (0-9).
- A locally-scoped label must begin with a period, followed by an underscore or letter (a-z, A-Z). All other characters must be an underscore, letter (a-z, A-Z) or number (0-9).
- Each local scope begins immediately after each global label and ends immediately before the next global label.
- All labels must be unique within the scope they belong to.

Label values are determined as follows:

- Labels defined in an ORGH section resolve to a hub address or offset (in bytes), regardless of whether the label is referenced in an ORGH or ORG section.
- Labels defined in an ORG section resolve to a cog address or offset (in longs), regardless of whether the label is referenced in an ORGH or ORG section.
- When the effective hub address or offset is needed for a label that is defined in an ORG section, the label may be preceded by a "@" to force resolution to a hub address or offset.
- Though it is possible to apply the "@" to labels defined in ORGH sections, it has no effect.

EXPRESSIONS

- Expressions can contain numbers, labels, and nested expressions. The simplest expression is either a single number or label.
- An expression that begins with # or ## is known as an "immediate" value.
- For branching instructions, immediate values can be either "absolute" or "relative", depending on context.
- For non-branching instructions, immediate values are always "absolute".
- "Absolute immediate" interpretation can be forced by using "#\" or "##\".
- There is no operator for forcing a "relative immediate" interpretation.
- # indicates a 9-bit (short-form) or 20-bit (long-form) immediate value:
 - For short-form branch instructions, this is a 9-bit relative immediate.
 - For long-form branch instructions that change execution mode (cog <-> hub), this is a 20-bit absolute immediate.
 - For long-form branch instructions that do not change execution mode, this is a 20-bit relative immediate.
 - For all other instructions, this is a 9-bit absolute immediate.

- In circumstances where an absolute immediate must be forced, the expression is prefaced with "#\".
- ## indicates a 32-bit immediate value
 - An implicit AUGx will precede the instruction containing the expression.
 - The lower 9 bits will be encoded in the instruction and the upper 23 bits will be encoded in the AUGx.
 - For short-form branch instructions, this is a 20-bit relative immediate. The upper 12 bits are ignored.
 - For non-branch instructions, this is a 32-bit absolute immediate.
 - This is meaningless for long-form branch instructions. PNUT throws an error.
- For BYTE/WORD/LONG, the expression is encoded as raw data. If the expression begins with # or ##, PNUT throws an error.
- For all other expressions that do not begin with # or ##, the expression is encoded as a register address and must be between \$000 and \$1FF.

ADDRESSING P2 MEMORY MAP

<[mindrobots cheat sheet](#)>

Reading memory from \$0000 to \$03FF with RDxxxx will read from hub memory whereas a jump/call to these locations will execute from cog or lut.

EXEC MAP

ADDR	NAME	DESCRIPTION
\$00_0000..\$00_01EF	COG EXEC	Code executes from cog register space (self-modifying code permitted)
\$00_0200..\$00_03FF	LUT EXEC	Code executes from lut register space
\$00_0400..\$0F_FFFF	HUB EXEC	Code executes from hub space (hub uses byte addressing) Code is not required to be long aligned Uses instruction streamer

COG REGISTERS

(9-bit addressable)

01F0: 0000.0000 0000.0000 0000.0000 0000.0000 0000.0000 0000.0000 0000.0980 0000.0000

01F8: 0000.131C 0000.0010 0000.0000 4000.0000 0000.0000 4000.0000 FFFF.FFFE FC00.0000

ADDR	READ	WRITE	NAME/USE	DESCRIPTION
------	------	-------	----------	-------------

<i>000-1EF</i>	<i>RAM</i>	<i>RAM</i>	<i>user</i>	<i>general-purpose 32-bit registers (and cog exec code space)</i>
<i>1F0</i>	<i>RAM</i>	<i>RAM</i>	<i>IJMP3</i>	<i>interrupt call address for INT3</i>
<i>1F1</i>	<i>RAM</i>	<i>RAM</i>	<i>IRET3</i>	<i>interrupt return address for INT3</i>
<i>1F2</i>	<i>RAM</i>	<i>RAM</i>	<i>IJMP2</i>	<i>interrupt call address for INT2</i>
<i>1F3</i>	<i>RAM</i>	<i>RAM</i>	<i>IRET2</i>	<i>interrupt return address for INT2</i>
<i>1F4</i>	<i>RAM</i>	<i>RAM</i>	<i>IJMP1</i>	<i>interrupt call address for INT1</i>
<i>1F5</i>	<i>RAM</i>	<i>RAM</i>	<i>IRET1</i>	<i>interrupt return address for INT1</i>
<i>1F6</i>	<i>RAM</i>	<i>RAM</i>	<i>ADRA</i>	<i>receives CALLD-immediate return or LOC address</i>
<i>1F7</i>	<i>RAM</i>	<i>RAM</i>	<i>ADRB</i>	<i>receives CALLD-immediate return or LOC address</i>
<i>1F8</i>	<i>PTRA</i>	<i>PTRA</i>	<i>PTRA</i>	<i>dedicated register for hub access pointer with auto inc/dec, cog ram is not accessible</i>
<i>1F9</i>	<i>PTRB</i>	<i>PTRB</i>	<i>PTRB</i>	<i>dedicated register for hub access pointer with auto inc/dec, cog ram is not accessible</i>
<i>1FA</i>	<i>RAM</i>	<i>DIRA (+RAM)</i>	<i>DIRA</i>	<i>output enables for P0..P31</i>
<i>1FB</i>	<i>RAM</i>	<i>DIRB (+RAM)</i>	<i>DIRB</i>	<i>output enables for P32..P63</i>
<i>1FC</i>	<i>RAM</i>	<i>OUTA (+RAM)</i>	<i>OUTA</i>	<i>output states for P0..P31</i>
<i>1FD</i>	<i>RAM</i>	<i>OUTB (+RAM)</i>	<i>OUTB</i>	<i>output states for P32..P63</i>
<i>1FE</i>	<i>INA</i>	<i>RAM</i>	<i>INA</i>	<i>input states for P0..P31 (also debug shadow int call address)</i>
<i>1FF</i>	<i>INB</i>	<i>RAM</i>	<i>INB</i>	<i>input states for P32..P63 (also debug shadow int ret address)</i>

LUT

<i>ADDR</i>	<i>R/W</i>	<i>NAME</i>	<i>DESCRIPTION</i>
-------------	------------	-------------	--------------------

200-3FF	RAM	user/cog-exec	
---------	-----	---------------	--

HUB

Updated 151010

ADDR	R/W	NAME	DESCRIPTION
\$00_0000..\$07_FFFF	RAM	user/hub-exec	(hubexec does not function for hub \$00000..\$00FFF as it is mapped to COG & LUT)
\$0F_FF80..\$0F_FFBF		mailboxes	16 special longs that create r/w events
\$0F_FFC0..\$0F_FFFF			Cog 0..15 (initial) debug interrupt vectors (PNut does not download to this)

HUB ROM

ADDR	R/W	NAME	DESCRIPTION
\$00_0000..\$00_3FFF	n/a	ROM	boot only - not accessible

INTERNAL STACK

P2 STACK_{ssw}

There is an eight level 22-bit Internal Stack in all COGs. This is accessible using the following instructions:

PUSH	D/#	push D/# on internal stack
POP	D {WC,WZ}	pop D from internal stack
CALL	D {WC,WZ}	save return address on internal stack
CALL	#abs/@rel	save return address on internal stack

RET	{WC,WZ}	jump via internal stack
-----	---------	-------------------------

=====

Conditional execution codes table

CODE	PASM directive	ALT	Description	Logic
1111	always		default	
1100	if_c	if_b	if below	C
0011	if_nc	if_ae	if above or equal	NC
1010	if_z	if_e	if equal	Z
0101	if_nz	if_ne	if not equal	NZ
1000	if_c_and_z			C&Z
0100	if_c_and_nz			C&NZ
0010	if_nc_and_z			NC&Z
0001	if_nc_and_nz	if_a	if above	NC&NZ
1110	if_c_or_z	if_be	if below or equal	C Z
1101	if_c_or_nz			C NZ
1011	if_nc_or_z			NC Z
0111	if_nc_or_nz			NC NZ
1001	if_c_eq_z			C=Z
0110	if_c_ne_z			C<>Z
0000	never		forces NOP	

INSTRUCTION BIT-FIELD SYMBOLS

Field	Description
S	Source address
D	Destination address
I	Immediate source
L	Immediate destination
R	Relative address
C	Effects Carry status
Z	Effects Zero status
	Fixed instruction field
CCCC	Conditional execution code - default is "always"

X

SAR D, S/# {wc,wz} Shift Arithmetic Right

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	0	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Shift Arithmetic right and preserve sign

SAL D, S/# {wc,wz} Shift Arithmetic Left

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	0	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Shift Arithmetic left and preserves lsb

ARITHMETIC

ADD D, S/# {wc,wz} Add S to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Add S to D unsigned. If the wc is specified then the carry flag is set if there is an overflow

ADDX D, S/# {wc,wz} Add S and carry to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Add S with carry to D unsigned. If the wc is specified then the carry flag is set if there is an overflow

ADD S D, S/# {wc,wz} Add signed S to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Add S to D signed. If the wc is specified then the carry flag is set if there is an overflow

ADD SX D, S/# {wc,wz} Add signed S with carry to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Add S with carry to D signed. If the wc is specified then the carry flag is set if there is an overflow

SUB D, S/# {wc,wz} Subtract S from D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	1	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Subtract S from D unsigned. If the wc is specified then the carry flag is set if there is an overflow

SUBX D, S/# {wc,wz} Subtract S with carry from D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Subtract S with carry from D unsigned. If the wc is specified then the carry flag is set if there is an overflow

SUBS D, S/# {wc,wz} Subtract signed S from D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Subtract S from D signed. If the wc is specified then the carry flag is set if there is an overflow

SUBSX D, S/# {wc,wz} Subtract signed S with carry from D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	0	1	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Subtract S with carry from D signed. If the wc is specified then the carry flag is set if there is an overflow

CMP D, S/# {wc,wz} Compare S to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare S to D unsigned. If the wc is specified then the carry flag is set if there is an overflow

CMPX D, S/# {wc,wz} Compare S with carry to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare S with carry to D unsigned. If the wc is specified then the carry flag is set if there is an overflow

CMPSX D , S/# {wc,wz} Compare signed S with carry to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare S with carry to D signed. If the wc is specified then the carry flag is set if there is an overflow

CMPR D , S/# {wc,wz} Compare Reverse (D to S)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	1	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare D to S (reversed) unsigned. If the wc is specified then the carry flag is set if there is an overflow

CMPM D , S/# {wc,wz} Compare (MSB) S to D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare S to D unsigned. If the wc is specified then the carry flag is set with the MSB of the (unwritten) result

SUBR D , S/# {wc,wz} Subtract Reverse (D from S)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Subtract D from S unsigned with result in D. If the wc is specified then the carry flag is set if these is an overflow

CMPSUB D , S/# {wc,wz} Compare S to D and Subtract if S<=D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	0	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Compare S to D unsigned and subtract S from D if it is lesser or equal. If the wc is specified then the carry flag is set if these is an overflow?

MIN D , S/# {wc,wz} Minimum limit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Limit value of D to a minimum of S

MAX D , S/# {wc,wz} Maximum limit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Limit value of D to a maximum of S

MINS D , S/# {wc,wz} Minimum Signed limit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Limit signed value of D to a minimum of S

MAXS D , S/# {wc,wz} Maximum Signed limit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Limit signed value of D to a maximum of S

0

MñjSUMC D , S/# {wc,wz} Sum Carry signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	1	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Sum signed value of D with the signed value of S which is negated if C=1

SUMNC D , S/# {wc,wz} Sum Not Carry signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Sum signed value of D with the signed value of S which is negated if C=0

SUMZ D , S/# {wc,wz} Sum Zero signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Sum signed value of D with the signed value of S which is negated if Z=1

SUMNZ D , S/# {wc,wz} Sum Not Zero signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	0	1	1	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Sum signed value of D with the signed value of S which is negated if Z=0

MUL D , S/# {wc,wz} Multiply 16x16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

C	C	C	C	0	1	1	1	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S	S
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Multiply 16-bit S with 16-bit D with a 32-bit result in D

MULS **D ,** **S/#** **{wc,wz}** **Multiply Signed 16x16**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	1	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Multiply signed 16-bit S with 16-bit D with a 32-bit signed result in D

ABS **D ,** **S/#** **{wc,wz}** **Absolute value**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Absolute value of S into D, that is negate the value of S if it is negative to make it positive.

NEG **D ,** **S/#** **{wc,wz}** **Negate value**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Negate value of S into D.

NEGC **D ,** **S/#** **{wc,wz}** **Negate value if Carry set**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	1	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get value S into D and negate if C=1.

NEGNC **D ,** **S/#** **{wc,wz}** **Negate value if Not Carry**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get value S into D and negate if C=0.

NEGZ **D ,** **S/#** **{wc,wz}** **Negate value if Zero set**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get value S into D and negate if Z=1.

NEGNZ **D ,** **S/#** **{wc,wz}** **Negate value if Not Zero**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get value S into D and negate if Z=0.

LOGICAL

ISOB **D ,** **S/#** **{wc,wz}** **Isolate Bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Isolate bit D[S/#] into C.

NOTB **D ,** **S/#** **{wc,wz}** **Not Bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Invert bit D[S/#] and set C to bit before it was inverted

CLRB **D ,** **S/#** **{wc,wz}** **Clear bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Clear bit D[S/#] and set C to bit before it was clear

SETB **D ,** **S/#** **{wc,wz}** **Set bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Clear bit D[S/#] and set C to bit before it was set

SETBC **D ,** **S/#** **{wc,wz}** **Set bit to C**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	1	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set bit D[S/#] to C and set C to bit before it was modified

SETBNC D, S/# {wc,wz} Set bit to Not C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set bit D[S/#] to Not C and set C to bit before it was modified

SETBZ D, S/# {wc,wz} Set bit to Z

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set bit D[S/#] to Z and set C to bit before it was modified

SETBNZ D, S/# {wc,wz} Set bit to Not Z

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	0	1	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set bit D[S/#] to Not Z and set C to bit before it was modified

ANDN D, S/# {wc,wz} AND Not

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	1	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

AND the Not of S to D

AND D, S/# {wc,wz} AND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	1	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

AND S to D

OR D, S/# {wc,wz} OR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	1	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

OR S to D

XOR D, S/# {wc,wz} XOR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	0	1	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

XOR S to D

Test the bits in D using the mask in S and set Z and C accordingly

ANYB **D ,** **S/#** **{wc,wz}** **Any Bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	1	0	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

OR S and D without modification and set C=any bit set, Z=result = 0

TESTB **D ,** **S/#** **{wc,wz}** **Test Bit**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	1	0	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Test bit D[S/#] and set C/Z to state ???

INSTRUCTION MODIFIERS

ALTI **D ,** **S/#** **Alter D/S in the next instruction**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	1	0	0	0	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Uses a D register for D/S field substitutions in the next instruction, while S/# modifies the D register's D and S fields and controls D/S substitution.

This is the old ALTDS without the wc,wz options.

ALTR **D ,** **S/#** **Alter R in the next instruction**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	1	0	0	0	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Use the sum of D and S/# for the result register in the next instruction

ALTD **D ,** **S/#** **Alter D in the next instruction**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	1	0	0	0	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Use the sum of D and S/# for the D register in the next instruction

ALTS **D ,** **S/#** **Alter S in the next instruction**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

Mix pixels

REV **D , S/#** **Reverse bits**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	1	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Reverse the bits in S and write to D (changed from P1)

SETI **D , S/#** **Set Instruction field**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	1	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set Instruction field (b27..b19)? of **Destination** with **Source**

SETD **D , S/#** **Set Destination field**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	1	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set destination field of D with S

SETS **D , S/#** **Set Source field**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	1	1	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set source field of D with S

REP **D/# , S/#** **Repeat instruction block**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	1	1	0	1	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Repeat following dest instructions by source count where 0 = infinite

AUGS **#S(23)** **Augment source of next instruction**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	1	1	0	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Augment the next instruction by extending its source field to a full 32-bits (9+23) [<test>](#)

Augment the next instruction's S or D field with additional 23-bits taken from b31..b9 of the assembler supplied parameter (b8..b0 are disregarded in PNut)

[illegible]

SETCZ	D/#	{wc,wz}	Set C and Z flags
-------	-----	---------	-------------------

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	0	0	0	1	1	0	0	0	0	

TOPONE	D ,	S/#	{wc,wz}	Top one
--------	-----	-----	---------	---------

[illegible]

BOTONE	D.	S/#	{wc,wz}	Bottom one
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
6	6	6	6	6
7	7	7	7	7
8	8	8	8	8
9	9	9	9	9
10	10	10	10	10
11	11	11	11	11
12	12	12	12	12
13	13	13	13	13
14	14	14	14	14
15	15	15	15	15
16	16	16	16	16
17	17	17	17	17
18	18	18	18	18
19	19	19	19	19
20	20	20	20	20
21	21	21	21	21
22	22	22	22	22
23	23	23	23	23
24	24	24	24	24
25	25	25	25	25
26	26	26	26	26
27	27	27	27	27
28	28	28	28	28
29	29	29	29	29
30	30	30	30	30
31	31	31	31	31
32	32	32	32	32
33	33	33	33	33
34	34	34	34	34
35	35	35	35	35
36	36	36	36	36
37	37	37	37	37
38	38	38	38	38
39	39	39	39	39
40	40	40	40	40
41	41	41	41	41
42	42	42	42	42
43	43	43	43	43
44	44	44	44	44
45	45	45	45	45
46	46	46	46	46
47	47	47	47	47
48	48	48	48	48
49	49	49	49	49
50	50	50	50	50
51	51	51	51	51
52	52	52	52	52
53	53	53	53	53
54	54	54	54	54
55	55	55	55	55
56	56	56	56	56
57	57	57	57	57
58	58	58	58	58
59	59	59	59	59
60	60	60	60	60
61	61	61	61	61
62	62	62	62	62
63	63	63	63	63
64	64	64	64	64
65	65	65	65	65
66	66	66	66	66
67	67	67	67	67
68	68	68	68	68
69	69	69	69	69
70	70	70	70	70
71	71	71	71	71
72	72	72	72	72
73	73	73	73	73
74	74	74	74	74
75	75	75	75	75
76	76	76	76	76
77	77	77	77	77
78	78	78	78	78
79	79	79	79	79
80	80	80	80	80
81	81	81	81	81
82	82	82	82	82
83	83	83	83	83
84	84	84	84	84
85	85	85	85	85
86	86	86	86	86
87	87	87	87	8

[illegible]

INCMOD	D.	S/#	{wc,wz}	Increment Modulo
---------------	-----------	------------	----------------	-------------------------

[illegible]

DECMOD	D ,	S/#	{wc,wz}	Decrement Modulo
---------------	------------	------------	----------------	-------------------------

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	1	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

DECODE	D ,	S/#	{wc,wz}	Decode alias MASK
--------	-----	-----	---------	-------------------

[illegible]

Decode index in S[4..0] into a **mask** in D. All other bits are set to 0.

MOV **D , S/# {wc,wz} Move**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Move from cog source to cog destination

NOT **D , S/# {wc,wz} Not**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	0	1	1	0	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Bitwise negation, Z = (result = 0), C = result[31]

LOC **reg , #abs/@rel Locate**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	1	0	1	W	W	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Locate the 20-bit address in hub and load into a pointer register. W: ? R: ? ADRA / ADRB

COG NIBBLE/BYTE/WORD Operations

SETNIB **D , S/#, #n Set Nibble**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	0	0	0	n	n	n	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set the nth nibble in the cog register D to S[3..0]

GETNIB **D , S/#, #n Get Nibble**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	0	0	1	n	n	n	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get the nth nibble in the cog register S to D[3..0] ???

ROLNIB **D , S/#, #n Rotate Left Nibble**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	0	1	0	n	n	n	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

SETBYTE	D ,	S/#,	#n	Set Byte
---------	-----	------	----	----------

GETBYTE	D ,	S/#,	#n	Get Byte
----------------	------------	-------------	-----------	-----------------

ROLBYTE	D,	S/#,	#n	Rotate Left Byte
---------	----	------	----	------------------

SETWORD	D,	S/#,	#n	Set Word
---------	----	------	----	----------

GETWORD	D,	S/#,	#n	Get Word
---------	----	------	----	----------

ROLWORD	D ,	S/#,	#n	Rotate Left Word
----------------	------------	-------------	-----------	-------------------------

Rotate the cog register D left by 16 bits, then add nth word in S

SETBYTES D, S/# Set Bytes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	0	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set ?

MOVBYTES D, S/# Move Bytes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	0	1	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Move ?

SPLITB D, S/# Split Bytes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	1	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Split ?

MERGE B D, S/# Merge Bytes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	1	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Merge ?

SPLITW D, S/# Split Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	1	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Split ?

MERGEW D, S/# Merge Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	0	1	1	1	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Merge ?

SEUSSF D, S/# SEUSS Forward

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	0	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Overwrite register "D (0-511)" with a pseudo random bit pattern seeded from the value in source.
After 32 forward iterations, the original bit pattern is returned.

SEUSSR D, S/# SEUSS Reverse

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	0	0	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Set ?

BRANCHING

Relative jumps are 9-bit signed so instructions such as DJNZ may jump forward as well as backward.

DJZ D, S/@ Decrement and Jump if Zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	0	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Decrement dest and if zero jump to source (9-bit signed relative)

DJNZ D, S/@ Decrement and Jump if Not Zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	0	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Decrement dest and if NOT zero jump to source (9-bit signed relative)

DJS D, S/@ Decrement and Jump if Signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	0	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Decrement dest and if signed positive jump to source (9-bit signed relative)

DJNS D, S/@ Decrement and Jump if Not Signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	0	1	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Decrement dest and if NOT signed positive jump to source (9-bit signed relative)

TJZ D , S/@ Test and Jump if Zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	1	0	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Test dest and if zero jump to source (9-bit signed relative)

TJNZ D , S/@ Test and Jump if Not Zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	1	0	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Test dest and if NOT zero jump to source (9-bit signed relative)

TJS D , S/@ Test and Jump if Signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	1	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Test dest and if signed positive jump to source (9-bit signed relative)

TJNS D , S/@ Test and Jump if Not Signed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	0	1	1	1	1	1	1	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Test dest and if NOT signed positive jump to source (9-bit signed relative)

JMPREL D/# Jump relative indexed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	0	0	0	0

Jump relative to the instruction using the index which automatically adjusts for hub (x4) or cog memory

Example

```

jmprel index      'works in both cog and hub
jmp   #pgm0
jmp   #pgm1
jmp   #pgm2
jmp   #pgm3
    
```

CALL REGISTER

CALL **D** **{wc,wz}** **Call**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	1	0	1

Call indirectly via dest register and use the internal hardware stack (8 levels)

CALLA **D** **{wc,wz}** **Call using PTR A**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	1	1	0

Call indirectly via dest register and use PTR A for the stack pointer

CALLB **D** **{wc,wz}** **Call using PTR B**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	1	1	1

Call indirectly via dest register and use PTR B for the stack pointer

POP **D** **{wc,wz}** **Pop hardware stack**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	1	1

Pop the return address from the hardware stack into register (23 bits?)

PUSH **D/#** **Push hardware stack**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	1	0

Push the register or immediate value onto the hardware stack

RET **D** **{wc,wz}** **Return**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1

Return via the hardware stack

RETA **D** **{wc,wz}** **Return using PTR A**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0

Return via the PTRB stack

RETB **D** **{wc,wz}** **Return using PTRB**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1

Return via the PTRB stack

CALLD **D ,** **S/@** **{wc,wz}** **Call and link D**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	1	0	1	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Call dest and save return in register S ?

CALL LONG

JMP **#abs20/@rel20** **Jump to 20-bit absolute or relative address**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	1	0	0	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Call the 20-bit absolute or relative address and use the internal hardware stack (8 levels)

CALL **#abs20/@rel20** **Call 20-bit absolute or relative address**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	1	0	1	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Call the 20-bit absolute or relative address and use the internal hardware stack (8 levels)

CALLA **#abs20/@rel20** **Call subroutine at 20-bit absolute or relative address**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	1	1	0	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Call the 20-bit absolute or relative address and use PTRB for the stack pointer

CALLB **#abs20/@rel20** **Call subroutine at 20-bit absolute or relative address**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	1	1	1	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Call the 20-bit absolute or relative address and use PTRB for the stack pointer

CALLD reg,#abs20/@rel20 Call subroutine at 20-bit absolute or relative address

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	1	0	0	w	w	R	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Call the 20-bit absolute or relative address and store the return address in index register "ww" (PTR A,PTR B,ADR A,ADR B)

LUT MEMORY

These instructions are mainly used to construct stacks as they work in a similar way to WRLONG and RDLONG.

WRLUT D/#, S/# Write to LUT memory

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	0	0	1	0	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Write to LUT RAM where S is the pointer to write D to

RDLUT D, S/# {wc,wz} Read from LUT memory

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	1	0	1	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Read from LUT memory

Example: Create stacks in LUT memory.

Pushing data to an incrementing stack

```
wrlut      mydata,stkptr      ' Save mydata
```

```
add        stkptr,#1
```

Popping data from an incrementing stack

```
sub        stkptr,#1
```

```
rdlut      mydata,stkptr      ' restore mydata from top of stack
```

HUB MEMORY

Write long to hub using S for pointer

SETQ D/# Set HUB repeat

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	1	1	0

Repeat HUB memory op (RDxxxx/WRxxxx) with auto increment. D = count-1

SETQ2 D/# Set LUT repeat

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	1	1	1

Repeat LUT memory op (RDxxxx/WRxxxx) with auto increment. D = count-1

RDFAST D/# , S/# Read Fast setup

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	0	1	1	1	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Setup a RDFAST block with D 64-byte blocks starting from address S

WRFAST D/# , S/# Write Fast setup

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	1	0	0	0	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Setup a WRFAST block with D times 64-byte blocks starting from address S before wrapping.

To make wrapping work S needs to be long aligned. If D = 0 = infinite then there is no wrapping.

FBLOCK D/# , S/# Fast Block

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	1	0	0	1	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Fast Block

RFBYTE D {wc,wz} Read Fast Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	0	0	0

Read fast byte

RFWORD D {wc,wz} Read Fast Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	0	0	1

Read fast word

RFLONG D {wc,wz} Read Fast Long

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	0	1	0

Read fast long

WFBYTE D/# {wc,wz} Write Fast Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	0	1	1

Write fast byte

WFWORD D/# {wc,wz} Write Fast Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	1	0	0

Write fast word

WFLONG D/# {wc,wz} Write Fast Long

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	0	1	0	1

Write fast long

SMART PINS

COND	INSTR	ZCR	DEST	SOURCE	NAME	OPER	EFFECTS	DESCRIPTION
CCCC	1011110	0LI	DDDDDDDD	SSSSSSSS	SETPAE	D/#,S/#		Set Port A Edges ?
CCCC	1011110	1LI	DDDDDDDD	SSSSSSSS	SETPAN	D/#,S/#		Set Port A edge polarity?
CCCC	1011111	0LI	DDDDDDDD	SSSSSSSS	SETPBE	D/#,S/#		Set Port B Edges ?
CCCC	1011111	1LI	DDDDDDDD	SSSSSSSS	SETPBN	D/#,S/#		Set Port B Edges ?

CCCC	1100001	1LI	DDDDDDDD	SSSSSSSS	MSGOUT	D/#,S/#		
CCCC	1010111	CZI	DDDDDDDD	SSSSSSSS	MSGIN	D,S/#	{WC,WZ}	
CCCC	1100000	0LI	DDDDDDDD	SSSSSSSS	JP	D/#,S/@		Jump to source if dest pin is high (dest spans ports)
CCCC	1100000	1LI	DDDDDDDD	SSSSSSSS	JNP	D/#,S/@		Jump to source if dest pin is low (dest spans ports)
CCCC	1100101	0LI	DDDDDDDD	SSSSSSSS	XINIT	D/#,S/#		transfer init, reset phase
CCCC	1100101	1LI	DDDDDDDD	SSSSSSSS	XZERO	D/#,S/#		transfer init, reset phase
CCCC	1100110	0LI	DDDDDDDD	SSSSSSSS	XCONT	D/#,S/#		transfer update, wait for rollover, continue

SMART PINS - INSTRUCTIONS

WSBYTE D/#,S/# 'write D[07:0] to pin S[5:0] data, mode dependent
 WSWORD D/#,S/# 'write D[15:0] to pin S[5:0] data, mode dependent
 WSLONG D/#,S/# 'write D[31:0] to pin S[5:0] data, mode dependent
 WSMODE D/#,S/# 'write D[31:0] to pin S[5:0] mode %MMMM_FFFFCIOHHLLL

RSBYTE D,S/# 'read byte from pin S[5:0] into D, mode dependent
 RSLONG D,S/# 'read long from pin S[5:0] into D, mode dependent

A = IN from this pad, B = IN from other pad, B OUT = OUT to other pad

MMMM	Description	pad DIR	pad OUT	Pattern	Setup	Update
00000	OUT (default)	DIR	OUT			
00001	B OUT	DIR	B OUT			
00010	CLK	DIR	CLK			
00011	* transitions		DIR	mode update-period-repeat	WSBYTE=prescaler	WSLONG=transitions
00100	* duty		DIR	mode update-period-repeat	WSBYTE=prescaler	WSLONG=adder ~
00101	* nco		DIR	mode update-period-repeat	WSBYTE=prescaler	WSLONG=adder ~
00110	* pwm sawtooth 16:16	DIR	mode update-period-repeat	WSBYTE=prescaler	WSLONG=F:T, WSWORD=T ~	
00111	* pwm triangle 16:16	DIR	mode update-period-repeat	WSBYTE=prescaler	WSLONG=F:T, WSWORD=T ~	
01000	* count highs		DIR ** OUT	period-update-repeat	WSLONG=period (0=cont)	RSLONG=count ~
01001	* count lows		DIR ** OUT	period-update-repeat	WSLONG=period (0=cont)	RSLONG=count ~
01010	* count all edges		DIR ** OUT	period-update-repeat	WSLONG=period (0=cont)	RSLONG=count ~
01011	* count positive edges		DIR ** OUT	period-update-repeat	WSLONG=period (0=cont)	RSLONG=count ~
01100	* time highs		DIR ** OUT	event-update-repeat		RSLONG=count ~
01101	* time lows		DIR ** OUT	event-update-repeat		RSLONG=count ~

Adds S/# to D, and sets internal timer counter 1 to the same value

ADDCT3 D , S/# Add and set Clock Tick 3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	0	1	0	1	0	0	1	0	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Adds S/# to D, and sets internal timer counter 1 to the same value

COGINIT D/# , S/# {wc} Cog Init

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	0	1	1	1	C	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Initialize Cog

D[8:6] *Reserved*

D[5] = %0 Copy \$1F8 longs from Hub @PTRB into cog, then JMP to \$000

D[5] = %1 JMP to PTRB

D[4:0] = %1---- Target cog is lowest-numbered inactive cog

D[4:0] = %0nnnn Target cog is indicated by %nnnn

S Address of first instruction to execute, copied to PTRB of the target cog

Q 20-bit value provided by SETQ, copied to PTRB of the target cog

{wc} If D[4] = %1 and D is not an immediate value:

- If no cog is available, C is set to %1
- Otherwise, C is set to %0 and D is set to new cog's ID

COGID D/# {wc,wz} Cog ID

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	0	0	1

Read the Cog's ID

COGSTOP D/# Cog Stop

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	0	1	

Stop the Cog

CLKSET **D/#** **{wc,wz}** **Clock Set**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	0	0	0

Set the system clock modes (PLL etc)

LOCKNEW **D** **{wc,wz}** **Lock New**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	0	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	1	0	0

Lock new

LOCKRET **D/#** **Lock ret**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	1	0	1

Lock ret

LOCKCLR **D/#** **{wc}** **Lock Clear**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	1	1	0

Clear the lock

LOCKSET **D/#** **{wc}** **Lock Set**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	0	0	1	1	1

Set the lock

GETCNT **D** **Get Count**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	1	0	1	0

Get system clock count into D

CORDIC

QMUL D/# , S/# Cordic Multiply

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	0	0	0	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Multiply S to D using the cordic engine for a 64-bit result in X (low) and Y (high). Use GETQX and GETQY to retrieve result.

QDIV D/# , S/# Cordic Divide

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	0	0	1	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Divide D by S using the cordic engine for a 32-bit quotient in X and 32-bit remainder in Y. Use GETQX and GETQY to retrieve result.
(64 / 32 unsigned divide)

QSQR D/# , S/# Cordic Square Root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	0	1	0	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Find the square root of S and place the integer result in the high word of X and the fractional result in the low word of X.

D is not used ? (supposed to be a 64-bit to 32-bit op)

Use GETQX to retrieve result. Example: TF2# 2 SQRT .LONG 0001.6A09 ok

SETDACS D/# Set DACs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	1	1	0	0

Set DACs

SETXFRQ D/# Set XFRQ

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	0	1	1	1	0	1

Set XFRQ

GETXCOS**D****Get X Cosine**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	1	1	1	0

Get X Cosine

GETXSIN**D****Get X Sine**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	D	D	D	D	D	D	D	D	D	0	0	0	0	1	1	1	1	1

Set DACs

CCCC 1101011 00L DDDDDDDDD 000010110	SETQ D/#	
CCCC 1101011 00L DDDDDDDDD 000010111	SETQ2 D/#	
CCCC 1101011 CZ0 DDDDDDDDD 000011000	GETQX D	{WC,WZ}
CCCC 1101011 CZ0 DDDDDDDDD 000011001	GETQY D	{WC,WZ}
CCCC 1101011 000 DDDDDDDDD 000011010	GETCT D	
CCCC 1101011 CZ0 DDDDDDDDD 000011011	GETRND {D}	{WC,WZ}

4c

- * **WFBYTE** and **WFWORD** write hub at first opportunity, bypassing the FIFO, meaning data no longer lingers until whole longs are formed
- * **Color space converter** added after Transfer to do RGB->YIQ/YPbPr/YUV/etc conversions
- * **ALTR/ALTD/ALTS** instructions added for doing indirect or base+offset accesses in next instruction
- * **ALTDS** renamed to **ALTI**
- * **SETXDAC** renamed to **SETDACS**
- * **GETPTR** instruction added to read back WFxxxx/RFxxxx address - doesn't wrap, though
- * **GETINT** instruction added to read INT1/INT2/INT3 states and event flags (non-destructive)
- * **SETBRK** modified to read back STALLI status and INT1/INT2/INT3 selector settings
- * **SETCY/SETCI/SETCQ/SETCFRQ/SETCMOD** instructions added to support colorspace converter

Older news:

- * Hub exec FIFO-level bug fixed

- * GETCNT renamed to GETCT
- * The Prop123-A7 board now has 10 cogs, not 11.
- * ADDCNT expanded to ADDCT1/ADDCT2/ADDCT3 - three timer events usable as interrupts
- * WMLONG added - like WRLONG, but doesn't write \$FF bytes, works with SETQ/SETQ2
- * 'JMP D' added - CALLD still required for interrupt returns
- * SETRD/SETWRL - related bugs fixed
- * C/Z properly restored on RETurns now
- * New SETHLK used to set hub LOCK bit event
- * GETQX/GETQY waiting improved to allow overlapped CORDIC operations without WAITX
- * PNut SUBX bug fixed
- * PNut now allows unary NOT/ABS/NEG... instructions (if D-only, D gets used for S)
- * PNut fixed for properly-oriented if_00/if_01/if_x0...

Initial debug ISR's have been moved up to \$FFFC0..\$FFFFF.

Event-triggering LONGs have been moved up to \$FFF80..\$FFFBF.

(No more complications at the bottom of hub RAM - everything starts at \$00000)

EVENTS, WAITS and INTERRUPTS

SETHLK **D/#** **Set Hub Lock**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	0	0	1	1

Set hub LOCK bit event

POLLINT **{wc}** **Poll interrupt**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLCT1 **{wc}** **Poll counter 1**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLCT2 {wc} Poll counter 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLCT3 {wc} Poll counter 3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLPAT {wc} Poll Pattern

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLEDG {wc} Poll Edge

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLRDL {wc} Poll RDLONG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLWRL {wc} Poll WRLONG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLHLK {wc} Poll Hub Lock

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0

Poll hub lock

POLLXMT

{wc}

Poll XMT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLXFI

{wc}

Poll XFI

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLXRO

{wc}

Poll XRO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	0	0

Poll the Transfer-NCO-rolled-over event flag

POLLFBW

{wc}

Poll FBW

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	1	0	0

Poll ?

POLLRLE

{wc}

Poll RLE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	1	0	0

Poll ?

WAITINT

{wc}

Wait for Interrupt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0

wait for interrupt-event, WC=1 for timeout using Q

WAITCT1

{wc}

Wait for Counter 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Wait for timer-event , WC=1 for timeout using Q

WAITCT2 {wc} Wait for Counter 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0

Wait for timer-event , WC=1 for timeout using Q

WAITCT3 {wc} Wait for Counter 3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	0

Wait for timer-event , WC=1 for timeout using Q

WAITPAT {wc} Wait for Pattern

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0

Wait for

WAITEDG {wc} Wait for Edge

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0

Poll the pin-edge-detected event flag

WAITRDL {wc} Wait for RDLONG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	1	0	0

Wait for the special-long-read event flag

WAITWRL {wc} Wait for WRLONG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	1	0	0

Wait for the special-long-written event flag

WAITHLK {wc} Wait for Hub Lock

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	1	0	0

Wait for the hub-LOCK-edge-detected event flag

WAITXRO {wc} Wait for Transfer Rolled-Over

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	1	0	0

Wait for Transfer-NCO-rolled-over event flag

WAITFBW {wc} Wait for FIFO Block Wrap

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0	0	1	0	0

Poll the hub-FIFO-interface-block-wrap event flag

WAITRLE {wc} Wait for RAM Block Event

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	0	0	0	0	0	0	1	1	0	1	1	0	0	0	1	0	0	1	0	0

Wait for the hub-RAM-FIFO-interface-block-wrap event flag

ALLOWI Allow Interrupts

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0

Allow Interrupts

STALLI Stall Interrupts

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0

Stall Interrupts

SETINT1 D/# Set Interrupt 1 mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	0	1	0	1

Set INT1 event to 0..15

SETINT2 D/# Set Interrupt 2 mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	0	1	1	0

Set INT2 event to 0..15

SETINT3 D/# Set Interrupt 3 mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	0	1	1	1

Set INT3 event to 0..15

WAITX D/# Wait for X cycles

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	0	0

Wait for X cycles

SETCZ D/# {wc,wz} Set C and Z

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	0	1

Set C flag to state of b1 in D if wc is specified, Set Z flag to state of b0 in D if wz is specified

PUSH D/# Push

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	1	0

Push D onto the internal 8-level stack

POP D {wc,wz} Pop D

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	0	1	1

Pop from the internal stack to D (typically return address from CALL) - Note, this stack is only 23-bits wide?

JMP D {wc,wz} Jump

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	C	Z	L	D	D	D	D	D	D	D	D	D	0	0	0	1	0	1	1	0	0

Jump to the 9-bit cog location (or via register?)

GETPTR D Get fast Pointer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	D	D	D	D	D	D	D	D	D	0	0	0	1	1	0	1	0	0

Get the current RD/WR FAST pointer

GETINT D Get Interrupt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	0	D	D	D	D	D	D	D	D	D	0	0	0	1	1	0	1	0	1

Get interrupt ?

SETBRK D/# Set break

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	0	1	1	0

Set break

SETCY D/# Set CY

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	0	0	0

Set CY

SETCI D/# Set CI

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	0	0	1

Set CI

SETCQ D/# Set CQ

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	0	1	0

Set CQ

SETCFRQ D/# Set CFRQ

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	0	1	1

Set CFRQ

SETCMOD										D/#																		Set CMOD									
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	1	0	0						

Set CMOD - color space converter

SETPIX										D/#										Set Pixels											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	1	0	1

Set pixels

SETPIV										D/#										Set PIV											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	1	1	0	1	0	1	1	0	0	L	D	D	D	D	D	D	D	D	D	0	0	0	1	1	1	1	1	0

Set PIV?

NOTES

SETTING EDGE EVENTS

SETEDG %L_EE_PPPPP

%L: 0 = pin
 1 = lock
%EE: 00 = any edge
 01 = pos edge
 10 = neg edge
 11 = any edge
%PPPPPP: pin number
%xxPPPP: lock number

SETRWL %RRRR_WWW

%RRRR: RDLONG-event address %0000_0000_0000_00RR_RR00
%WWW: WRLONG-event address %0000_0000_0000_00WW_WW00

SETINT1/SETINT2/SETINT3 %MMM

%MMM: 000 = disable interrupt - default
 001 = enable timer-event interrupt
 010 = enable pat-event interrupt
 011 = enable edge-event interrupt
 100 = enable RDLONG-event interrupt
 101 = enable WRLONG-event interrupt
 110 = enable transfer-rollover-event interrupt
 111 = enable fast-block-wrap-event interrupt

WAITINT is waiting for an interrupt. The next instruction is already in the pipeline. WAITINT stops waiting when an interrupt occurs. The

next instruction executes, while the interrupt CALLD is being injected into the pipeline. The next instruction that executes is CALLD. So the instruction following the WAITINT executes **before** the interrupt code.

ALIASES

JMP	reg	{WC,WZ}	=	CALLD	INB,reg	{WC,WZ}
-----	-----	---------	---	-------	---------	---------

PUSHA	reg/#	=	WRLONG	reg/#,PTRA++
-------	-------	---	--------	--------------

PUSHB	reg/#	=	WRLONG	reg/#,PTRB++
-------	-------	---	--------	--------------

POPA	reg	=	RDLONG	reg,--PTRA
------	-----	---	--------	------------

POPB	reg	=	RDLONG	reg,--PTRB
------	-----	---	--------	------------

RETI0	=	CALLD	INB,INB	WC,WZ
-------	---	-------	---------	-------

RETI1	=	CALLD	INB,\$1F5	WC,WZ
-------	---	-------	-----------	-------

RETI2	=	CALLD	INB,\$1F3	WC,WZ
-------	---	-------	-----------	-------

RETI3	=	CALLD	INB,\$1F1	WC,WZ
-------	---	-------	-----------	-------

NOP	=	\$00000000
-----	---	------------

POINTER ADDRESSING MODES

INDEX = -16..+15 for simple offsets, 0..15 for ++'s, or 0..16 for --'s

SCALE = 1 for byte, 2 for word, 4 for long

S = 0 for PTR A, 1 for PTR B

U = 0 to keep PTRx same, 1 to update PTRx

P = 0 to use PTRx + INDEX*SCALE, 1 to use PTRx (post-modify)

NNNNN = INDEX

nnnnn = -INDEX

1SUPNNNNN	PTR expression		

100000000	PTRA	'use PTRA	
110000000	PTRB	'use PTRB	
101100001	PTRA++	'use PTRA,	PTRA += SCALE
111100001	PTRB++	'use PTRB,	PTRB += SCALE
101111111	PTRA--	'use PTRA,	PTRA -= SCALE
111111111	PTRB--	'use PTRB,	PTRB -= SCALE
101000001	++PTRA	'use PTRA + SCALE,	PTRA += SCALE
111000001	++PTRB	'use PTRB + SCALE,	PTRB += SCALE
101011111	--PTRA	'use PTRA - SCALE,	PTRA -= SCALE
111011111	--PTRB	'use PTRB - SCALE,	PTRB -= SCALE
1000NNNNN	PTRA[INDEX]	'use PTRA + INDEX*SCALE	
1100NNNNN	PTRB[INDEX]	'use PTRB + INDEX*SCALE	
1011NNNNN	PTRA++[INDEX]	'use PTRA,	PTRA += INDEX*SCALE
1111NNNNN	PTRB++[INDEX]	'use PTRB,	PTRB += INDEX*SCALE
1011nnnnn	PTRA--[INDEX]	'use PTRA,	PTRA -= INDEX*SCALE
1111nnnnn	PTRB--[INDEX]	'use PTRB,	PTRB -= INDEX*SCALE
1010NNNNN	++PTRA[INDEX]	'use PTRA + INDEX*SCALE,	PTRA += INDEX*SCALE
1110NNNNN	++PTRB[INDEX]	'use PTRB + INDEX*SCALE,	PTRB += INDEX*SCALE
1010nnnnn	--PTRA[INDEX]	'use PTRA - INDEX*SCALE,	PTRA -= INDEX*SCALE
1110nnnnn	--PTRB[INDEX]	'use PTRB - INDEX*SCALE,	PTRB -= INDEX*SCALE

Examples:

Read byte at PTRA into D

```
1111 1011000 001 DDDDDDDDD 100000000 RDBYTE D,PTRA
```

Write lower word in D to PTRB+7*2

```
1111 1100010 001 DDDDDDDDD 110000111      WRWORD  D,PTRB[7]
```

Write long value 10 at PTRB, PTRB += 1*4

```
1111 1100011 011 000001010 111100001      WRLONG  #10,PTRB++
```

Read word at PTRB into D, PTRB -= 1*2

```
1111 1011001 001 DDDDDDDDD 101111111      RDWORD  D,PTRB--
```

Write lower byte in D at PTRB-1*1, PTRB -= 1*1

```
1111 1100010 001 DDDDDDDDD 101011111      WRBYTE  D,--PTRB
```

Read long at PTRB+10*4 into D, PTRB += 10*4

```
1111 1011010 001 DDDDDDDDD 111001010      RDLONG  D,++PTRB[10]
```

Write lower byte in D to PTRB, PTRB += 15*1

```
1111 1100010 001 DDDDDDDDD 101101111      WRBYTE  D,PTRB++[15]
```

HUB MEMORY READING AND WRITING

Here are the basic instructions for reading and writing hub RAM:

```
CCCC 1011000 CZI DDDDDDDDD SSSSSSSSS      RDBYTE  D,S/#/PTRx {WC,WZ}
CCCC 1011001 CZI DDDDDDDDD SSSSSSSSS      RDWORD  D,S/#/PTRx {WC,WZ}
CCCC 1011010 CZI DDDDDDDDD SSSSSSSSS      RDLONG  D,S/#/PTRx {WC,WZ}
CCCC 1100010 0LI DDDDDDDDD SSSSSSSSS      WRBYTE  D/#,S/#/PTRx
CCCC 1100010 1LI DDDDDDDDD SSSSSSSSS      WRWORD  D/#,S/#/PTRx
CCCC 1100011 0LI DDDDDDDDD SSSSSSSSS      WRLONG  D/#,S/#/PTRx
```

In the case of the 'S/#/PTRx' operand, three possibilities exist:

- S is a register
- #00..\$FF indicates hub address \$00..\$FF
- PTRx expression with optional pre-/post-modifier and scaled index

STREAMER

Ability to stream hub RAM and/or lookup RAM to DACs and pins, also pins to hub RAM.

By preceding RDLONG with either SETQ or SETQ2, multiple hub longs can be read into either register RAM or lookup RAM. This transfer happens at the rate of one long per clock, assuming there is no hub streaming going on. If hub streaming is active, the hub reads will have to wait for cycles when the next-needed window occurs and the streamer is not requiring the window for itself.

```
CZL  000001101    <empty>                                00L  000011101    SETXFRQ D/#
00L  000001110    QLOG    D/#                            000  000011110    GETXCOS D
```

The [streamer](#) can write data directly to the i/o pins, not just to the DACs, up to 32 bits per clock, **from** HUB or LUT and **to** HUB.

Here is how you read multiple hub longs into register RAM:

```
SETQ    #x                                `x = number of longs, minus 1, to read
RDLONG  first_reg,S/#/PTRx                `read x+1 longs starting at first_reg
```

Here is how you read hub longs into lookup RAM:

```
SETQ2   #x                                `x = number of longs, minus 1, to read
RDLONG  first_lut,S/#/PTRx                `read x+1 longs starting at first_lut
```

WRLONG can be preceded by SETQ or SETQ2 to write multiple hub longs from register RAM. If SETQ2 is used, only non-\$FF bytes will be written. This masking feature enables byte-level overlay data to be imposed onto existing hub data.

A simple way to do a long fill with a const, here 0, is just:

```
SETQ longcount
```

WRLONG #0,startaddress

The I/O Transfer Unit (Streamer) accesses HubRAM via the FIFO Unit

The FIFO Unit of each Cog performs all HubRAM burst accesses for that Cog; including for HubExec, for RD/WRFast instructions and for the I/O Transfer Unit. Only one of these three can use the FIFO Unit at a time.

ALTDS

(Seairth) On a slightly related note, I just noticed that there weren't any INDx registers in the 8/13 document. Did we lose indirect registers in the new design?

(Chip)

Yes, they are gone. We have an ALTDS instruction now that substitutes D and S fields in the next instruction. ALTDS also increments/decrements those fields in its D register, with S supplying the inc/dec controls. It was a really cheap way around what could be a huge hardware situation, like in Prop2-Hot.

You might want to review the conversation on ALTDS here - they describe single and double indirection code examples.

<http://forums.parallax.com/discussion/156242/question-about-altDS-implementation-in-new-chip/p1>

(Chip)

The other day I revisited ALTDS because we had moved the CCCC bits to the front of the opcode. The old SETI instruction now writes S[8:0] into D[27:19] (the 0000000CZ bits), instead of into the top bits

opcode: CCCC 0000000 CZI DDDDDDDDD SSSSSSSSS

The 0000000CZ bits in a variable (not an instruction) can be used to redirect result writing, while the DDDDDDDDD and SSSSSSSSS fields can redirect D and S. It works like this:

ALTDS D,S/# 'modify D according to bits in S and possibly replace next instruction's CCCC0000000CZI / DDDDDDDDD / SSSSSSSSS fields.

In ALTDS, S provides the following pattern: %RRR_DDD_SSS

%RRR: (101 allows instruction substitution)

000 = don't affect D's CCCC00000000CZI field
001 = don't affect D's CCCC00000000CZI field, cancel write for next instruction
010 = decrement D's 0000000CZ field
011 = increment D's 0000000CZ field
100 = use D's 0000000CZ field as the result register for the next instruction (separate from D)
101 = use D's CCCC0000000CZI field as next instruction's CCCC0000000CZI field
110 = use D's 0000000CZ field as the result register for the next instruction, decrement D's 0000000CZ field
111 = use D's 0000000CZ field as the result register for the next instruction, increment D's 0000000CZ field

%DDD

000 = don't affect D's DDDDDDDDD field
001 = copy D's SSSSSSSSS field into its DDDDDDDDD field
010 = decrement D's DDDDDDDDD field
011 = increment D's DDDDDDDDD field
100 = use D's DDDDDDDDD field as the DDDDDDDDD field for the next instruction
101 = use D's DDDDDDDDD field as the DDDDDDDDD field for the next instruction, copy D's SSSSSSSSS field into its DDDDDDDDD field
110 = use D's DDDDDDDDD field as the DDDDDDDDD field for the next instruction, decrement D's DDDDDDDDD field
111 = use D's DDDDDDDDD field as the DDDDDDDDD field for the next instruction, increment D's DDDDDDDDD field

%SSS

000 = don't affect D's SSSSSSSSS field
001 = copy D's DDDDDDDDD field into its SSSSSSSSS field
010 = decrement D's SSSSSSSSS field
011 = increment D's SSSSSSSSS field
100 = use D's SSSSSSSSS field as the SSSSSSSSS field for the next instruction
101 = use D's SSSSSSSSS field as the SSSSSSSSS field for the next instruction, copy D's DDDDDDDDD field into its SSSSSSSSS field
110 = use D's SSSSSSSSS field as the SSSSSSSSS field for the next instruction, decrement D's SSSSSSSSS field
111 = use D's SSSSSSSSS field as the SSSSSSSSS field for the next instruction, increment D's SSSSSSSSS field

You can see that when those three-bit RRR/DDD/SSS fields have their MSB's clear, they are only affecting D. When their MSB's are set, though, they additionally affect the next instruction in some way.

When RRR is 101, it actually uses D's upper bits to replace the functionality of the next instruction, which might as well be a NOP, unless its DDDDDDDDD and SSSSSSSSS fields are meaningful.

It hurts to think about, but I think, as someone proposed above, compounded indirection can be achieved. Also, some crazy instruction substitution possibilities exist. And, not being self-modifying code, this can all work from hub-exec.

ALTDS uses a D register for D/S field substitutions in the next instruction, while S/# modifies the D register's D and S fields and controls D/S substitution.

ALTDS D,S/#

D - a register whose D/S fields may be substituted for the next instructions' D/S fields

S/# - an 8-bit code: %ABBBCDDD

%A:

0 = don't substitute next instructions' D field with current D register's D field

1 = substitute next instructions' D field with current D register's D field

%BBB:

000 = leave the current D register's D field the same

0xx = add 1/2/3 to D field,

1xx = subtract 1/2/3/4 from D field

%C:

0 = don't substitute next instructions' S field with current D register's S field

1 = substitute next instructions' S field with current D register's S field

%DDD:

000 = leave the current D register's S field the same

0xx = add 1/2/3 to S field

1xx = subtract 1/2/3/4 from S field

([Cluso](#))

This permits the additional possibilities of:

- * redirecting the result

- * redirecting the result to an unused register (maybe INx) to perform a pseudo NR

Therefore, might it be beneficial, and would it be easy to do the following ???

S/# = %RRRDDDSSS

where RRR, DDD and SSS mean:

000 = don't substitute next instructions S/D/R field, leave the current D registers S/D/I value the same

001 = substitute next instructions S/D/R field with the current D registers S/D/I field, then add 1 to the current D registers S/D/I value

010 = substitute next instructions S/D/R field with the current D registers S/D/I field, then add 2 to the current D registers S/D/I value

011 = substitute next instructions S/D/R field with the current D registers S/D/I field, then add 4 to the current D registers S/D/I value

100 = substitute next instructions S/D/R field with the current D registers S/D/I field, leave the current D registers S/D/I value the same

101 = substitute next instructions S/D/R field with the current D registers S/D/I field, then subtract 1 from the current D registers S/D/I value

110 = substitute next instructions S/D/R field with the current D registers S/D/I field, then subtract 2 from the current D registers S/D/I value

111 = substitute next instructions S/D/R field with the current D registers S/D/I field, then subtract 4 from the current D registers S/D/I value

1/2/4 covers byte/word/long in hub, and 1/2/4 longs in cog.

ALTDS Examples

(Ozpropdev)

While I agree that ALTDS is a little awkward it more than compensates I think in its efficiency.
For example

copy 16 cog regs from .src to .dest

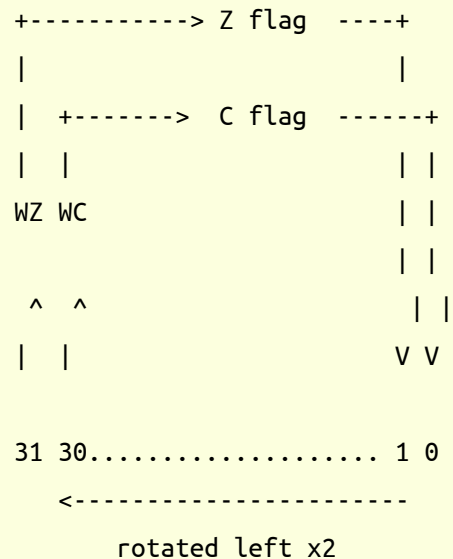
```
copy_cogram  mov    .myreg,##.dest << 9 | .src
              rep    @.copy_end,#16
              altds  .myreg,##%000_111_111
              mov    0-0,0-0
.copy_end
```

Nice and compact and efficient.

PUSHZC ??? - Old P2_hot instruction ???

PUSHZC rotates ZC flags into D register bits 1:0.

Bits 31:30 of D are rotated into ZC flags if WZ WC effect is included.



POPZC rotates bits 0:1 of D register into ZC flags.

ZC flags are rotated into Bits 31:30 of D if WZ WC effect is included.

In this example the lower 4 bits of zc_reg contain the before and after ZC flags status.

```
PUSHZC zc_reg
INCMOD myreg,#13 wz,wc
PUSHZC zc_reg
```

RDFAST

(Chip)

RDFAST #0,startbyteaddress

Once you do that, 'RFBYTE D (WC,WZ)' can be used to read contiguous bytes, starting from startbyteaddress. RFBYTE means 'read fast byte' and it always takes 2 clocks - meaning RDFAST blocks waiting for Hub. RDFAST initiates the read-fast mode. This doesn't work with hub exec, because hub exec uses the RDFAST mode, itself. That first D/# term in RDFAST tells how many 64-byte blocks to read before wrapping back to startbyteaddress (0= infinite). To make wrapping work, startbyteaddress must be long-aligned.

WRFast works the same way, and uses WFBYTE, WFWORD, WFLONG.

NOTE: Makes use of the Cog's FIFO Unit. This is also used by HubExec and the Streamer - Mutually exclusive.

P2 INTERNAL STACK

(ALL COGs) 24SEP2015

=====

There is an eight level 22-bit Internal Stack in all COGs. It is intended use is to store C & Z flags plus a 20-bit return address for stack CALL instructions.

NOTE: isn't this actually an 8-level 23-bit wide Stack just for calls? (edited above - Cluso 8 OCT)

This is accessible using the following instructions...

```
-----
CCCC 1101011 00L DDDDDDDDD 000101000  PUSH   D/#           'push D/# on internal stack
CCCC 1101011 CZ0 DDDDDDDDD 000101100  POP     D      {WC,WZ} 'pop  D from internal stack
CCCC 1101011 CZ0 DDDDDDDDD 000101001  CALL    D      {WC,WZ} 'save return address on internal stack
CCCC 1101101 Rnn nnnnnnnnn nnnnnnnnn  CALL    #abs/@rel    'save return address on internal stack
CCCC 1101011 CZ0 000000000 000101101  RET      {WC,WZ} 'jump via internal stack
=====
```

MAILBOXES AND DEBUG INTERRUPT VECTORS

```
$0F.FF80 $80 DUMPL
FF80: 0000.0000 0000.0000 0000.0000 0000.0000 .....
FF90: 0000.0000 0000.0000 0000.0000 0000.0000 .....
FFA0: 0000.0000 0000.0000 0000.0000 0000.0000 .....
FFB0: 0000.0000 0000.0000 0000.0000 0000.0000 .....
FFC0: FABB.FFFF FABB.FFFF FABB.FFFF FABB.FFFF .....
FFD0: FABB.FFFF FABB.FFFF FABB.FFFF FABB.FFFF .....
FFE0: FABB.FFFF FABB.FFFF FABB.FFFF FABB.FFFF .....
FFF0: FABB.FFFF FABB.FFFF FABB.FFFF FABB.FFFF ..... ok
```

Migrating From Propeller 1

Instruction Changes

Instruction	Propeller 1	Propeller 2
CALL	Alias for JMPRET, assembler trickery	Push PC+1/C/Z on 8-deep stack, then jump to D
DJNZ	Can set C/Z with WC/WZ.	C/Z stays unchanged.
JMP	Alias for JMPRET NR	Jump to D
MAX	Z is set to (S = 0), C is set to unsigned(D<S)	Z is set to (result = 0), C is set to (result <> D)
MAXS	Z is set to (S = 0), C is set to signed(D<S)	Z is set to (result = 0), C is set to (result <> D)
MIN	Z is set to (S = 0), C is set to unsigned(D<S)	Z is set to (result = 0), C is set to (result <> D)
MINS	Z is set to (S = 0), C is set to signed(D<S)	Z is set to (result = 0), C is set to (result <> D)
NEG	C is set to S[31]	C is set to result[31]
NEGC	C is set to S[31]	C is set to result[31]
NEGNC	C is set to S[31]	C is set to result[31]
NEGNZ	C is set to S[31]	C is set to result[31]
NEGZ	C is set to S[31]	C is set to result[31]
RET	Alias for JMPRET, relies on “_ret” label	Returns to top address on 8-deep stack. Use with CALL.
REV	D[31..0] is set to D[0..31], then shifted right by S	D[31..0] is set to S[0..31]
RCL	C is set to D[31]	C is set to last bit shifted out
RCR	C is set to D[0]	C is set to last bit shifted out
ROL	C is set to D[31]	C is set to last bit shifted out
ROR	C is set to D[0]	C is set to last bit shifted out

SHL	C is set to D[31]	C is set to last bit shifted out
SHR	C is set to D[0]	C is set to last bit shifted out
TJNZ	Can set C/Z with WC/WZ	C/Z stays unchanged.
TJZ	Can set C/Z with WC/WZ	C/Z stays unchanged.
WAITCNT	Wait until target CNT is reached, then add delta to D	Wait until target CNT is reached. Use ADDCT1, ADDCT2, or ADDCT3 to set target and add delta.

Removed Instructions/Registers/Effects

Name	Type	Comment
ABSNEG	instruction	Can be achieved with combination of ABS and NEG
ADDABS	instruction	Can be achieved with a combination of ABS and ADD
CNT	register	Use GETCNT instruction
CTRA	register	Replaced by smart pins.
CTRB	register	Replaced by smart pins.
FRQA	register	Replaced by smart pins.
FRQB	register	Replaced by smart pins.
JMPRET	instruction	Closest match is CALLD
MOVD	instruction	Renamed to SETD
MOVI	instruction	Renamed to SETI
MOVS	instruction	Renamed to SETS
NR	effect	Where the NR/WR feature is needed, two instructions exist (TEST and AND, CMP and SUB, etc.)

PAR	register	
PHSA	register	Replaced by smart pins.
PHSB	register	Replaced by smart pins.
SUBABS	instruction	Can be achieved with a combination of ABS and SUB
VCFG	register	
VSCL	register	
WAITPEQ	instruction	Set with SETPAE/SETPBE. Use WAITPAT to block. Can also use POLLPAT or interrupt.
WAITPNE	instruction	Set with SETPAN/SETPBN. Use WAITPAT to block. Can also use POLLPAT or interrupt.
WAITVID	instruction	
WR	effect	Not available on P2. Where the NR/WR feature is needed, two instructions exist.

151027 P2 UPDATES

- * ADDCNT expanded to ADDCT1/ADDCT2/ADDCT3 - three timer events usable as interrupts
- * WMLONG added - like WRLONG, but doesn't write \$FF bytes, works with SETQ/SETQ2
- * 'JMP D' added - CALLD still required for interrupt returns
- * SETRDL/SETWRL - related bugs fixed
- * C/Z properly restored on RETurns now
- * New SETHLK used to set hub LOCK bit event
- * GETQX/GETQY waiting improved to allow overlapped CORDIC operations without WAITX
- * PNut SUBX bug fixed
- * PNut now allows unary NOT/ABS/NEG... instructions (if D-only, D gets used for S)
- * PNut fixed for properly-oriented if_00/if_01/if_x0...

151109

- * **WFBYTE and WFWORD write hub at first opportunity, bypassing the FIFO, meaning data no longer lingers until whole longs are formed**
- * **Color space converter added after Transfer to do RGB->YIQ/YPbPr/YUV/etc conversions**
- * **ALTR/ALTD/ALTS instructions added for doing indirect or base+offset accesses in next instruction**
- * **ALTDS renamed to ALTI**
- * **SETXDAC renamed to SETDACS**
- * **GETPTR instruction added to read back WFxxxx/RFxxxx address - doesn't wrap, though**
- * **GETINT instruction added to read INT1/INT2/INT3 states and event flags (non-destructive)**
- * **SETBRK modified to read back STALLI status and INT1/INT2/INT3 selector settings**
- * **SETCY/SETCI/SETCQ/SETCFRQ/SETCMOD instructions added to support colorspace converter**

CCCC 1101011 00L DDDDDDDDDD 000100011 SETHLK D/#

CCCC 1101011 C00 000000000 000100100	POLLINT	{WC}
CCCC 1101011 C00 000000001 000100100	POLLCT1	{WC}
CCCC 1101011 C00 000000010 000100100	POLLCT2	{WC}
CCCC 1101011 C00 000000011 000100100	POLLCT3	{WC}
CCCC 1101011 C00 000000100 000100100	POLLPAT	{WC}
CCCC 1101011 C00 000000101 000100100	POLLEDG	{WC}
CCCC 1101011 C00 000000110 000100100	POLLRDL	{WC}
CCCC 1101011 C00 000000111 000100100	POLLWRL	{WC}
CCCC 1101011 C00 000001000 000100100	POLLHLK	{WC}

CCCC 1101011 C00 000001001 000100100	POLLXRO	{WC}
CCCC 1101011 C00 000001010 000100100	POLLFBW	{WC}
CCCC 1101011 C00 000001011 000100100	POLLRLE	{WC}
CCCC 1101011 C00 000001100 000100100	POLLWLE	{WC}
CCCC 1101011 C00 000010000 000100100	WAITINT	{WC}
CCCC 1101011 C00 000010001 000100100	WAITCT1	{WC}
CCCC 1101011 C00 000010010 000100100	WAITCT2	{WC}
CCCC 1101011 C00 000010011 000100100	WAITCT3	{WC}
CCCC 1101011 C00 000010100 000100100	WAITPAT	{WC}
CCCC 1101011 C00 000010101 000100100	WAITEDG	{WC}
CCCC 1101011 C00 000010110 000100100	WAITRDL	{WC}
CCCC 1101011 C00 000010111 000100100	WAITWRL	{WC}
CCCC 1101011 C00 000011000 000100100	WAITHLK	{WC}
CCCC 1101011 C00 000011001 000100100	WAITXRO	{WC}
CCCC 1101011 C00 000011010 000100100	WAITFBW	{WC}
CCCC 1101011 C00 000011011 000100100	WAITRLE	{WC}
CCCC 1101011 000 000100000 000100100	ALLOWI	
CCCC 1101011 000 000100001 000100100	STALLI	