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

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 branche 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

- All cog register accesses are direct via instructions (MOV rega,regb).
- All lut access is via RDLUT/WRLUT (cog registers <--> lut registers) or SETQ2+RDLONG (hub --> lut).
- All hub accesses are via RDxxxx/WRxxxx/RFxxxx/WFxxxx, only.
- "@", "#hublabel", "#\hublabel" refers to hub RAM, only.
- "#@hublabel" is the same as "#hublabel".
- "@", "#\", and "#@" cannot be used to point at anything in the cog or lut. They always denote hub memory.

All symbols defined under ORGH are hub addresses. Any reference to one of them returns a hub address.

All symbols defined under ORG are both cog and hub addresses, with a direct reference returning a cog address. Using @ before one of those symbols returns the hub address, instead.

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

LUT

ADDR	R/W	NAME	DESCRIPTION
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

P2 INTERNAL STACK

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"

P2 INSTRUCTIONS LIST

SHIFTS ROTATES

ROR D , S/# {wc,wz} Rotate 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	0	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Rotate D right by S linking from bit 0 to bit 31. If wc is specified the C will be set if the lsb of the result = 1 ?

ROL D , S/# {wc,wz} Rotate 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	0	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Rotate D left by S linking from bit 31 to bit 0. If wc is specified the C will be set if the msb of the result = 1

SHR D , S/# {wc,wz} Shift 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	0	1	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Shift D right by S with zero written to bit 31. If wc is specified the C will be set if the lsb of the result = 1

SHL D , S/# {wc,wz} Shift 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	0	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Shift D left by S with zero written to bit 0. If wc is specified the C will be set if the msb of the result = 1

RCR D , S/# {wc,wz} Rotate Carry 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	0	0	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

x

RCL D , S/# {wc,wz} Rotate Carry 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	0	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

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

ADDS 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

ADDSX 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

Comapre 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

SUMC 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	D	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

COND	INSTR	CZI	DEST	SOURCE	NAME	OPER	EFFECTS	DESCRIPTION
CCCC	0100000	CZI	DDDDDDDDD	SSSSSSSSS	**ISOB**	D,S/#	{WC,WZ}	Isolate bit D[S/#], C = isolated bit
CCCC	0100001	CZI	DDDDDDDDD	SSSSSSSSS	**NOTB**	D,S/#	{WC,WZ}	Invert bit D[S/#], C = bit before invert
CCCC	0100010	CZI	DDDDDDDDD	SSSSSSSSS	**CLRB**	D,S/#	{WC,WZ}	Clear bit D[S/#], C = bit before clear
CCCC	0100011	CZI	DDDDDDDDD	SSSSSSSSS	**SETB**	D,S/#	{WC,WZ}	Set bit D[S/#], C = bit before set
CCCC	0100100	CZI	DDDDDDDDD	SSSSSSSSS	**SETBC**	D,S/#	{WC,WZ}	Set/Clear destination bit to C status
CCCC	0100101	CZI	DDDDDDDDD	SSSSSSSSS	**SETBNC**	D,S/#	{WC,WZ}	Set/Clear destination bit to NOT C status
CCCC	0100110	CZI	DDDDDDDDD	SSSSSSSSS	**SETBZ**	D,S/#	{WC,WZ}	Set/Clear destination bit to Z status
CCCC	0100111	CZI	DDDDDDDDD	SSSSSSSSS	**SETBNZ**	D,S/#	{WC,WZ}	Set/Clear destination bit to NOT Z status

CCCC	0101000	CZI	DDDDDDDDD	SSSSSSSSS	**ANDN**	D,S/#	{WC,WZ}	
CCCC	0101001	CZI	DDDDDDDDD	SSSSSSSSS	**AND**	D,S/#	{WC,WZ}	
CCCC	0101010	CZI	DDDDDDDDD	SSSSSSSSS	**OR**	D,S/#	{WC,WZ}	
CCCC	0101011	CZI	DDDDDDDDD	SSSSSSSSS	**XOR**	D,S/#	{WC,WZ}	
CCCC	0101100	CZI	DDDDDDDDD	SSSSSSSSS	**MUXC**	D,S/#	{WC,WZ}	Set bits in dest specified by source to the state of C
CCCC	0101101	CZI	DDDDDDDDD	SSSSSSSSS	**MUXNC**	D,S/#	{WC,WZ}	Set bits in dest specified by source to the state of NOT C
CCCC	0101110	CZI	DDDDDDDDD	SSSSSSSSS	**MUXZ**	D,S/#	{WC,WZ}	Set bits in dest specified by source to the state of Z
CCCC	0101111	CZI	DDDDDDDDD	SSSSSSSSS	**MUXNZ**	D,S/#	{WC,WZ}	Set bits in dest specified by source to the state of NOT Z
CCCC	0110001	CZI	DDDDDDDDD	SSSSSSSSS	**NOT**	D,S/#	{WC,WZ}	
CCCC	1010000	CZI	DDDDDDDDD	SSSSSSSSS	**TESTN**	D,S/#	{WC,WZ}	
CCCC	1010001	CZI	DDDDDDDDD	SSSSSSSSS	**TEST**	D,S/#	{WC,WZ}	
CCCC	1010010	CZI	DDDDDDDDD	SSSSSSSSS	**ANYB**	D,S/#	{WC,WZ}	Or’s D and S without saving result in D. C=Any bit set. Z = Or’ed result = 0
CCCC	1010011	CZI	DDDDDDDDD	SSSSSSSSS	**TESTB**	D,S/#	{WC,WZ}	

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	0	1	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)

INSTRUCTION MODIFIERS

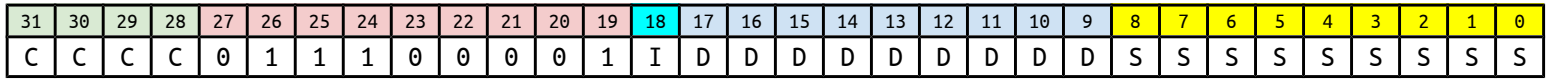
ALTI D , S/# Alter D/S in the next instruction

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.

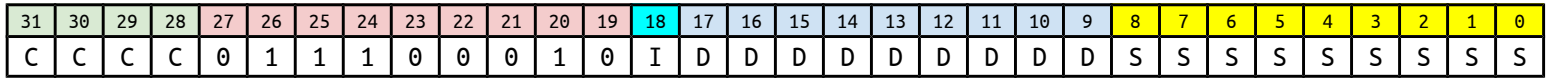
This is the old ALTDS without the wc,wz options.

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



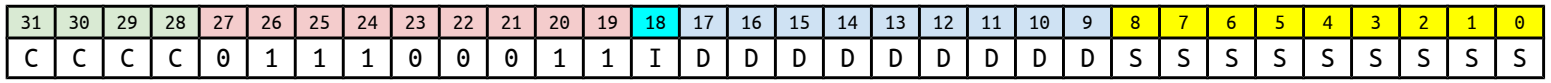
Use the sum of D and S/# for the result register in the next instruction

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



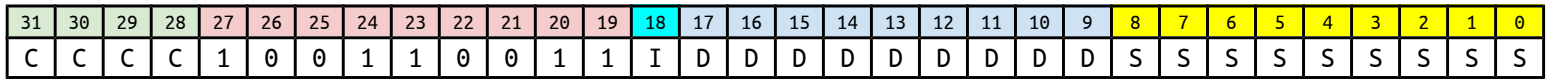
Use the sum of D and S/# for the D register in the next instruction

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



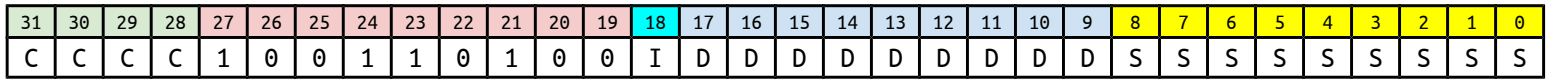
Use the sum of D and S/# for the S register in the next instruction

SETI D , S/# Set Instruction field



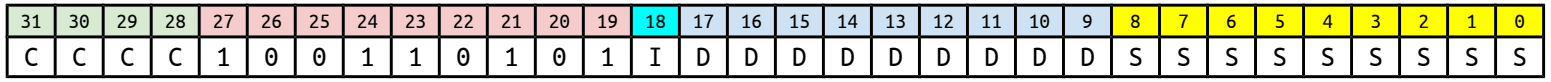
Set **I**nstruction field (b27..b19)? of **D**estination with **S**ource

SETD D , S/# Set Destination field



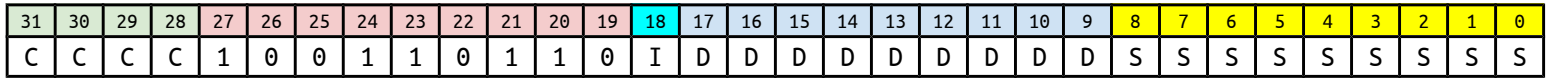
Set destination field (b17..b9) of **D**estination (instruction) with **S**ource

GETD D , S/# Get Destination field



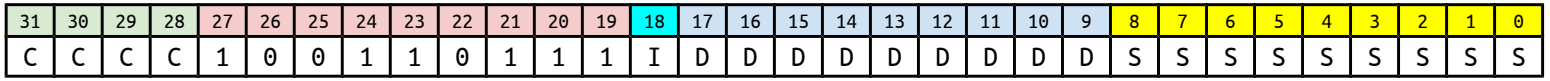
Get destination field ???

SETS D , S/# Set Source field



Set source field

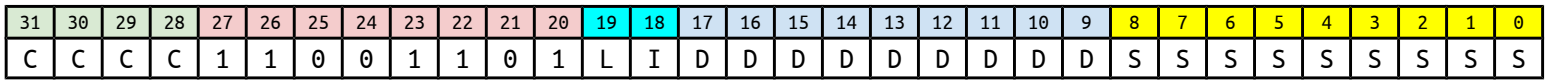
GETS D , S/# Get Source field



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

Get source field

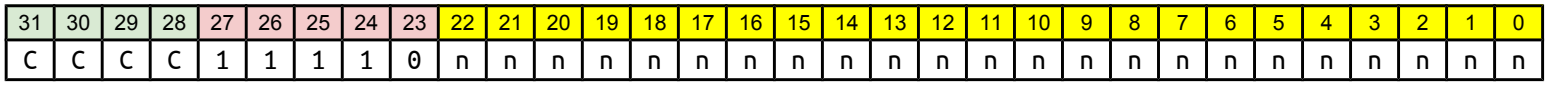
REP D/# , S/# Repeat instruction block



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

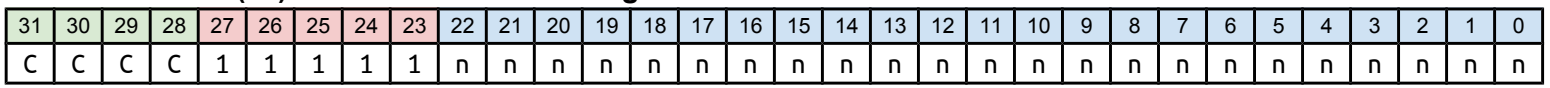
AUGS #S(23) Augment source of next instruction



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)

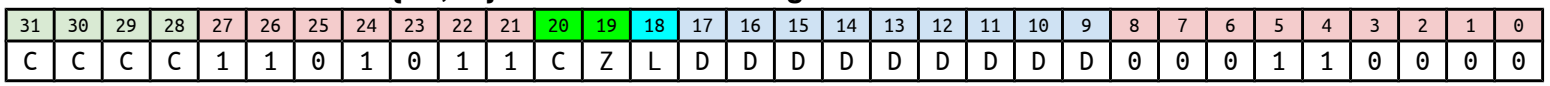
AUGD #D(23) Augment destination of next instruction



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	1	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 destination field to a full 32-bits (9+23) <test>

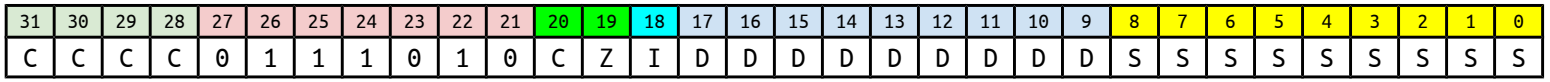
SETCZ D/# {wc,wz} Set C and Z flags



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	D	0	0	0	1	1	0	0	0	0

Set the carry and zero flags to b1 and b0 of D. If WC is applied then C = b1 of D and Z = b0 of D

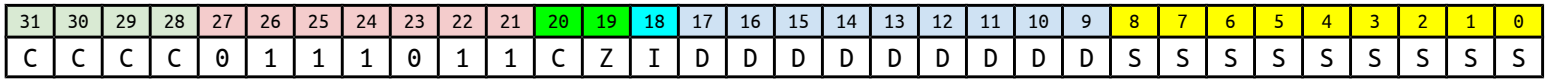
TOPONE D , S/# {wc,wz} Top one



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

Index of most significant bit that is set to 1, C is set to (S/# = 0)

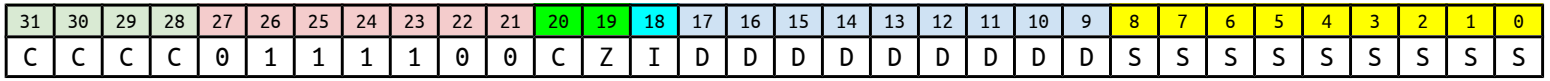
BOTONE D , S/# {wc,wz} Bottom one



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	1	1	C	Z	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

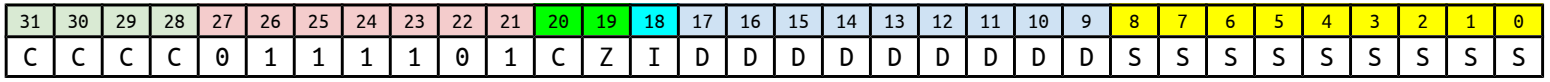
Index of least significant bit that is set to 1, C is set to (S/# = 0)

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



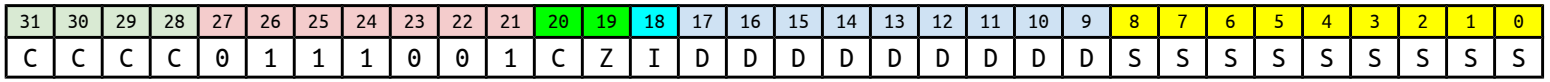
D = (D==S ? 0 : D+1) Increment to S/# then wrap to zero. (inclusive)

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



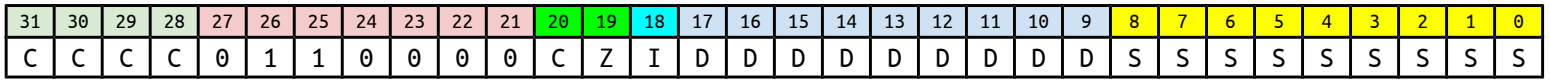
D = (D==S ? 0 : D-1) Decrement to S/# then wrap to S/#. (inclusive)

DECOD D , S/# {wc,wz} Decode



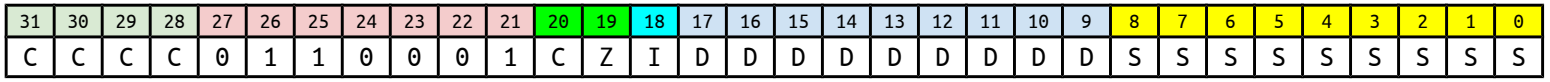
Decode index in S[4..0] into a mask in D. All other bits are set to 0.

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



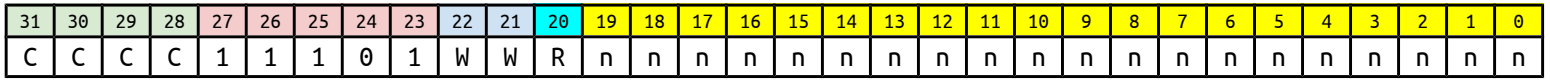
Move from cog source to cog destination

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



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

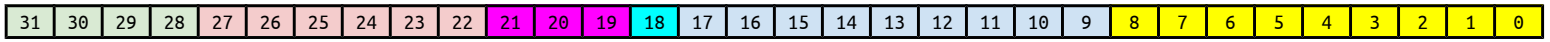
LOC reg , #abs/@rel Locate



Locate the 20-bit address in hub and load into a pointer register

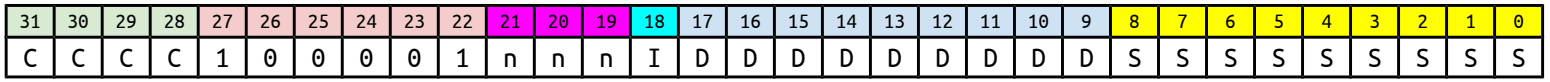
COG NIBBLE/BYTE/WORD Operations

SETNIB D , S/#, #n Set Nibble



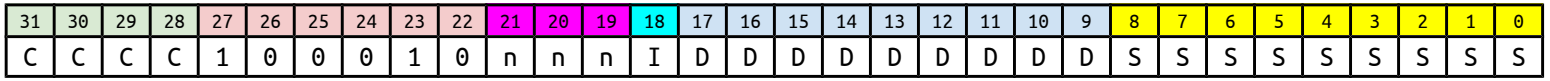
Set the nth nibble in the cog register D to S[3..0]

GETNIB D , S/#, #n Get Nibble



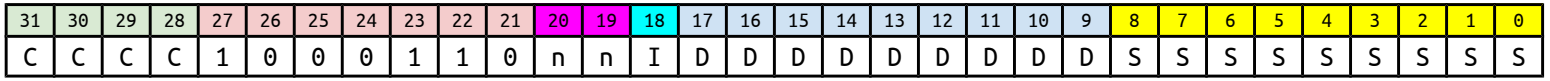
Get the nth nibble in the cog register S to D ???

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



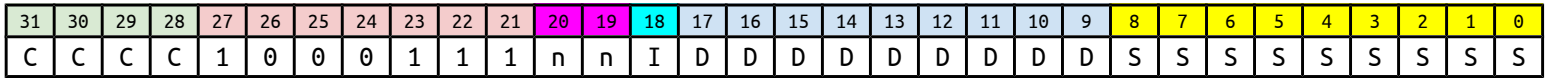
Rotate the cog register D left by 4 bits, then add nth nibble in S

SETBYTE D , S/#, #n Set Byte



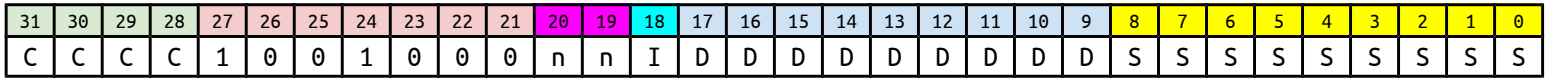
Set the nth byte in the cog register D to S[7..0]

GETBYTE D , S/#, #n Get Byte



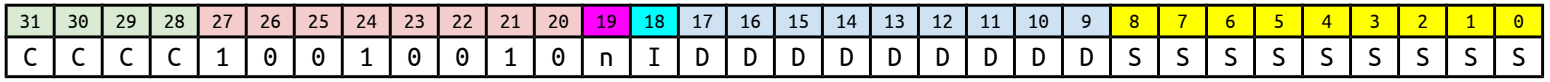
Get the nth byte in the cog register S to D[7..0]

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



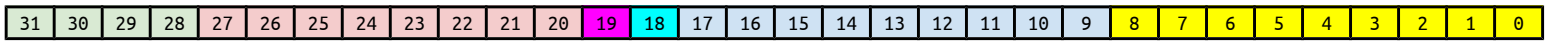
Rotate the cog register D left by 8 bits, then add nth byte in S

SETWORD D , S/#, #n Set Word



Set the nth word in the cog register D to S[15..0]

GETWORD D , S/#, #n Get Word



C	C	C	C	1	0	0	1	0	0	1	1	n	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Get the nth word in the cog register S to D[15..0]

ROLWORD D , S/#, #n Rotate Left 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	0	0	1	0	1	0	0	n	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

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

SETBYTS 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 ?

MOVBYTS 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 ?

MERGEB 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 PTRA

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 PTRA for the stack pointer

CALLB D {wc,wz} Call 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	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 PTRB 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 PTRA

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 PTRA 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 PTRA 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" (PTRA,PTRB,ADRA,ADRB)

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

WMLONG D , S/#/PTRx Write masked 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	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

Works like WRLONG but doesn't write $FF bytes, works with SETQ/SETQ2

RDBYTE D , S/#/PTRx {wc,wz} Read Byte from hub

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	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

Read byte from hub using S for pointer

RDWORD D , S/#/PTRx {wc,wz} Read Word from hub

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	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

Read unaligned word from hub using S for pointer

RDLONG D , S/#/PTRx {wc,wz} Read Long from hub

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	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

Read unaligned long from hub using S for pointer

WRBYTE D/# , S/#/PTRx Write Byte to hub

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	0	0	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Write byte to hub using S for pointer

WRWORD D/# , S/#/PTRx Write Word to hub

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	0	1	L	I	D	D	D	D	D	D	D	D	D	S	S	S	S	S	S	S	S	S

Write word to hub using S for pointer

WRLONG D/# , S/#/PTRx Write Long to hub

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

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	DDDDDDDDD	SSSSSSSSS	**SETPAE**	D/#,S/#		Set Port A Edges ?
CCCC	1011110	1LI	DDDDDDDDD	SSSSSSSSS	**SETPAN**	D/#,S/#		Set Port A edge polarity?
CCCC	1011111	0LI	DDDDDDDDD	SSSSSSSSS	**SETPBE**	D/#,S/#		Set Port B Edges ?
CCCC	1011111	1LI	DDDDDDDDD	SSSSSSSSS	**SETPBN**	D/#,S/#		Set Port B Edges ?
CCCC	1100001	1LI	DDDDDDDDD	SSSSSSSSS	**MSGOUT**	D/#,S/#		
CCCC	1010111	CZI	DDDDDDDDD	SSSSSSSSS	**MSGIN**	D,S/#	{WC,WZ}	
CCCC	1100000	0LI	DDDDDDDDD	SSSSSSSSS	**JP**	D/#,S/@		Jump to source if dest pin is high (dest spans ports)
CCCC	1100000	1LI	DDDDDDDDD	SSSSSSSSS	**JNP**	D/#,S/@		Jump to source if dest pin is low (dest spans ports)
CCCC	1100101	0LI	DDDDDDDDD	SSSSSSSSS	**XINIT**	D/#,S/#		transfer init, reset phase
CCCC	1100101	1LI	DDDDDDDDD	SSSSSSSSS	**XZERO**	D/#,S/#		transfer init, reset phase
CCCC	1100110	0LI	DDDDDDDDD	SSSSSSSSS	**XCONT**	D/#,S/#		transfer update, wait for rollover, continue

COG and HUB CONTROL

ADDCT1 D , S/# Add and set Clock Tick 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	0	1	0	1	0	0	0	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

ADDCT2 D , S/# Add and set Clock Tick 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	0	1	0	1	0	0	0	1	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

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[3..0] with the code in hub RAM at address S

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	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**
*** 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...**

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 ?

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	0	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	0	1	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	0	1	1	0	0	0	1	0	0	1	0	0

Poll ?

POLLWLE **{wc}** **Poll WLE**

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 ?

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	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

NOTES

SETTING EDGE EVENTS

```
SETEDG %L_EE_PPPPPP

%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_WWWW

%RRRR:        RDLONG-event address %0000_0000_0000_00RR_RR00
%WWWW:        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 PTRA, 1 for PTRB
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 PTRA into D, PTRA -= 1*2

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

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

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

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 PTRA, PTRA += 15*1

```
1111 1100010 001 DDDDDDDDD 101101111     WRBYTE  D,PTRA++[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 OOOOOOOCZ bits), instead of into the top bits
opcode: CCCC OOOOOOO CZI DDDDDDDDD SSSSSSSSS

The OOOOOOOCZ 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 CCCCOOOOOOOCZI / DDDDDDDDD / SSSSSSSSS fields.
In ALTDS, S provides the following pattern: %RRR_DDD_SSS

%RRR: (101 allows instruction substitution)
000 = don't affect D's CCCCOOOOOOOOOCZI field
001 = don't affect D's CCCCOOOOOOOOOCZI field, cancel write for next instruction
010 = decrement D's OOOOOOOCZ field
011 = increment D's OOOOOOOCZ field
100 = use D's OOOOOOOCZ field as the result register for the next instruction (separate from D)
101 = use D's CCCCOOOOOOOCZI field as next instruction's CCCCOOOOOOOCZI field
110 = use D's OOOOOOOCZ field as the result register for the next instruction, decrement D's OOOOOOOCZ field
111 = use D's OOOOOOOCZ field as the result register for the next instruction, increment D's OOOOOOOCZ 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

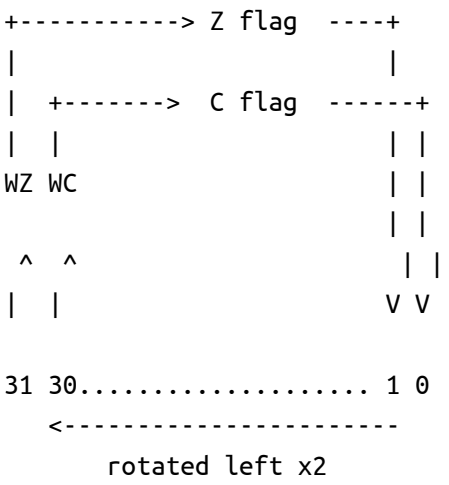
```
copy_cogram  mov    .myreg,##.dest << 9 | .src
             rep    @.copy_end,#16
```

```
            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)
MINS	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)
MIN	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.

Experimenting with different document layouts

Use consistent colors and bit-field numbers to help identify the makeup of the instruction. Some instructions use a preset D or S field to identify the instruction so these are colored the same as the instruction field etc.

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 and if the wc is specified then the carry flag is set if there is an overflow

SETCZ 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	1	0	0	0	0

Set carry and zero status flags to corresponding bits in D
If WC effect applied, C = D[1]. If WZ effect applied, Z = D[0].

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...

CCCC 1101011 00L DDDDDDDDD 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
```