```
Some info merged from:
Propeller2DetailedPreliminaryFeatureList-v2.pdf
Some info merged from:
From diverse thread's

'SPECIAL REGISRERS
'-----------------

"Nutson - Sapieha. I remember Chip saying there were more than 40 registers now.
                 ' We will get a full description in due time.

'Just read the preliminary feature list and made this list for my own reference:
' I added some more.

'There are 10 memory mapped registers that allow control over I/O pins and indirection:

INDA/INDB                0x1F6 - 0x1F7   'Indirect registers access to COG memory
PINA/PINB/PINC/PIND      0x1F8 - 0x1FB   'Read / write I/O ports
DIRA/DIRB/DIRC/DIRD      0x1FC - 0x1FF   'Enables or disables the output functionally of PORTA.
                                         'Input reading is never disabled.

'All other registers can be accessed only with specialised instructions

PTRA/PTRB                        'Pointer for hub access
SPA/SPB                          'CLUT (stack) pointer
CNT                              'System time counter
CTRA/CTRB (FRQ,PHS,SIN,COS)      'Each have FRQ, PHS, SIN and COS register
MULLL/MULLH                      'etc, registers to acces the multiply, divide, SQRT and CORDIC ooperations
DAC0/DAC1/DAC2/DAC3              'configuration and data for the DAC's
LFSR                             'Random number generator
MACA/MACB                        'Accu for 64 bit MAC operation
(ACCA 64-bit)                    'Multiply Accumulator A.
(ACCB 64-bit)                    'Multiply Accumulator B.
```

```
Instruction                                    | Encoding
--------------------------------------------------------------------------------------------------
_____ D(estination),S(ource)____________________________________ D(estination),S(ource)___________
          MNEMONIC  D,S                        |   ------ --- 0 1111 DDDDDDDDD SSSSSSSSS
          MNEMONIC  D,#n                       |   ------ --- 1 1111 DDDDDDDDD nnnnnnnnn
[COND]    MNEMONIC  D,S [WZ] [WC] [NR]         |   ------ ZCR 1 CCCC DDDDDDDDD SSSSSSSSS
          MNEMONIC  INDA,S [WZ] [WC] [NR]      |   ------ ZCR 0 AA00 111110110 SSSSSSSSS
          MNEMONIC  INDA,#n [WZ] [WC] [NR]     |   ------ ZCR 0 AA00 111110110 nnnnnnnnn
          MNEMONIC  D,INDA [WZ] [WC] [NR]      |   ------ ZCR 0 00AA DDDDDDDDD 111110110
          MNEMONIC  INDA,INDB [WZ] [WC] [NR]   |   ------ ZCR 0 AABB 111110110 111110111
          -------
          Example                              ------------------------------------
          RDBYTE  D,S                          000000 0 01 0 1111 DDDDDDDDD SSSSSSSSS
          RDBYTE  D,PTR                        000000 0 01 1 CCCC DDDDDDDDD SUPNNNNNN
[COND]    RDBYTE  D,S [WZ]                     000000 Z 01 0 CCCC DDDDDDDDD SSSSSSSSS
[COND]    RDBYTE  D,PTR [WZ]                   000000 Z 01 1 CCCC DDDDDDDDD nnnnnnnnn
          RDBYTE  INDA,S [WZ]                  000000 Z 01 0 AA00 111110110 SSSSSSSSS
          RDBYTE  INDA,PTR [WZ]                000000 Z 01 1 AA00 111110110 SUPNNNNNN
          RDBYTE  D,INDA [WZ]                  000000 Z 01 0 00AA DDDDDDDDD 111110110
          RDBYTE  INDA,INDB [WZ]               000000 Z 01 0 AABB 111110110 111110111
-------------------------------------------------------------------------------------------------
-Effect       |   Result
--------------------------------------------------------------------------------------------------
    WZ       |   [[(default)] meaning of zero flag set to 1]
    WC       |   [[(default)] meaning of carry flag set to 1]
    WR       |   [[(default)] meaning of value written to register]
--------------------------------------------------------------------------------------------------
          COGINIT D                        |    000011 000 0 1111 DDDDDDDDD SSSSSSSSS
[COND]    COGINIT D,S [WZ] [WC] [WR]       |    000011 ZCR 0 CCCC DDDDDDDDD SSSSSSSSS
          COGSTOP D                        |    000011 000 1 1111 DDDDDDDDD 000000011
[COND]    COGSTOP D [WZ] [WC] [WR]         |    000011 ZCR 0 CCCC DDDDDDDDD 000000011
--------------------------------------------------------------------------------------------------
CCCC             'Condition bit pattern (not available for instructions using indirect addressing)
 AA INDA/INDB    'destination encoding for all instructions that support indirect addressing
 BB INDA/INDB    'source encoding for all instructions that support indirect addressing
 Z  'Zero effect
 C  'Carry effect
 R  'Register effect
 I  'Immediate effect
```

```
PROPELLER 2 MEMORY
------------------

In the Propeller 2, there are two primary types of memory:

HUB MEMORY

    128K bytes of main memory shared by all cogs

        - cogs launch from this memory
        - cogs can access this memory as bytes, words, longs, and quads (4 longs)
        - $00000..$00E7F is ROM - contains Booter, SHA-256/HMAC, and Monitor
    - $00E80..$1FFFF is RAM - for application usage


COG MEMORY (8 instances)

    512 longs of register RAM for code and data usage

        - simultaneous instruction, source, and destination reading, plus writing
        - last eight registers are for I/O pin control

    256 longs of stack RAM for data and video usage

        - accessible via push and pop operations
        - video circuit can read data simultaneously and asynchronously
_____________________________________
xxxxxx xxx x xxxx DDDDDDDDD SSSSSSSSS
                  |         S   Source field in instruction
                  D   Destination field in instruction

 PTRA and PTRB are only for pointing to HUB memory .
 INDA and INDB are for pointing to COG memory .
 SPA  and SPB  are for pointing to CLUT/stack memory .

'If you want to read longs quickly into registers,
 it's simplest to just do 'RDLONGC INDA++,PTRA++'.
```

```
' Less stuff to think about that way.

PTR EXPRESSIONS:
INDEX   -32 .. +31      Simple offset
INDEX     0 ..  31      ++ Auto-increments range
INDEX     0 ..  32      -- Auto-decrement range
SCALE     1             BYTE
SCALE     2             WORD
SCALE     4             LONG
SCALE    16             QUAD
```

HUB MEMORY INSTRUCTIONS

These instructions read and write HUB memory .

All instructions use D as the data conduit, except WRQUAD/RDQUAD/RDQUADC, which uses the four QUAD registers. The QUADs can be mapped into COG register space using the SETQUAD instruction or kept hidden, in which case they are still useful as data conduit and as a read cache. If mapped, the QUADs overlay four contiguous COG registers. These overlaid registers can be read and written as any other registers, as well as executed. Any write via D to the QUAD registers, when mapped, will affect the underlying COG registers, as well. A RDQUAD/RDQUADC will affect the QUAD registers, but not the underlying COG registers.

The cached reads RDBYTEC/RDWORDC/RDLONGC/RDQUADC will do a RDQUAD if the current read address is outside of the 4-long window of the prior RDQUAD. Otherwise, they will immediately return cached data. The CACHEX instruction invalidates the cache, forcing a fresh RDQUAD next time a cached read executes.

Hub memory instructions must wait for their COG's HUB cycle, which comes once every 8 clocks. The timing relationship between a COG's instruction stream and its HUB cycle is generally indeterminant, causing these instructions to take varying numbers of clocks. Timing can be made determinant, though, by intentionally spacing these instructions apart so that after the first in a series executes, the subsequent HUB memory instructions fall on HUB cycles, making them take the minimal numbers of clocks. The trick is to write useful code to go in between them.

WRBYTE/WRWORD/WRLONG/WRQUAD/RDQUAD complete on the HUB cycle, making them take 1..8 clocks.

```
RDBYTE/RDWORD/RDLONG complete on the 2nd clock after the HUB cycle, making them take 3..10 clocks.

RDBYTEC/RDWORDC/RDLONGC take only 1 clock if data is cached, otherwise 3..10 clocks.

RDQUADC takes only 1 clock if data is cached, otherwise 1..8 clocks.


 Floating QUAD 'window does not copy its contents to the underlying registers.

After a RDQUAD, mapped QUAD registers are accessible via D and S after three clocks:


        RDQUAD  hubaddress      'read a quad into the QUAD registers mapped at quad0..quad3

    NOP         'do something for at least 3 clocks to allow QUADs to update
    NOP
    NOP

    CMP     quad0,quad1     'mapped QUADs are now accessible via D and S


After a RDQUAD, mapped QUAD registers are executable after three clocks and one instruction:


        SETQUAD #quad0          'map QUADs to quad0..quad3
                                Floating QUAD window does not copy its contents to the underlying registers.

        RDQUAD  hubaddress      'read a quad into the QUAD registers mapped at quad0..quad3

        NOP                     'do something for at least 3 clocks to allow QUADs to update
        NOP
        NOP

        NOP                     'do at least 1 instruction to get QUADs into pipeline

quad0   NOP                     'QUAD0..QUAD3 are now executable
quad1   NOP
quad2   NOP
```

```
quad3   NOP


After a SETQUAD, mapped QUAD registers are writable immediately, but original contents are
readable via D and S after 2 instructions:


        SETQUAD #quad0          'map QUADs to quad0..quad3 (new address)

        NOP         'do at least two instructions to queue up QUADs
        NOP

        CMP     quad0,quad1     'mapped QUADS are now accessible via D and S


On cog startup, the QUAD registers are cleared to 0's.

instructions                                                                                    clocks
------------------------------------------------------------------------------------------------------
000000 000 0 CCCC DDDDDDDDD SSSSSSSSS     WRBYTE  D,S       write lower byte in D at S             1..8
000000 000 1 CCCC DDDDDDDDD SUPNNNNNN     WRBYTE  D,PTR     write lower byte in D at PTR           1..8
000000 Z01 0 CCCC DDDDDDDDD SSSSSSSSS     RDBYTE  D,S       read byte at S into D                 3..10
000000 Z01 1 CCCC DDDDDDDDD SUPNNNNNN     RDBYTE  D,PTR     read byte at PTR into D               3..10
000000 Z11 0 CCCC DDDDDDDDD SSSSSSSSS     RDBYTEC D,S       read cached byte at S into D      1, 3..10
000000 Z11 1 CCCC DDDDDDDDD SUPNNNNNN     RDBYTEC D,PTR     read cached byte at PTR into D    1, 3..10

000001 000 0 CCCC DDDDDDDDD SSSSSSSSS     WRWORD  D,S       write lower word in D at S             1..8
000001 000 1 CCCC DDDDDDDDD SUPNNNNNN     WRWORD  D,PTR     write lower word in D at PTR           1..8
000001 Z01 0 CCCC DDDDDDDDD SSSSSSSSS     RDWORD  D,S       read word at S into D                 3..10
000001 Z01 1 CCCC DDDDDDDDD SUPNNNNNN     RDWORD  D,PTR     read word at PTR into D               3..10
000001 Z11 0 CCCC DDDDDDDDD SSSSSSSSS     RDWORDC D,S       read cached word at S into D      1, 3..10
000001 Z11 1 CCCC DDDDDDDDD SUPNNNNNN     RDWORDC D,PTR     read cached word at PTR into D    1, 3..10

000010 000 0 CCCC DDDDDDDDD SSSSSSSSS     WRLONG  D,S       write D at S                           1..8
000010 000 1 CCCC DDDDDDDDD SUPNNNNNN     WRLONG  D,PTR     write D at PTR                         1..8
000010 Z01 0 CCCC DDDDDDDDD SSSSSSSSS     RDLONG  D,S       read long at S into D                 3..10
000010 Z01 1 CCCC DDDDDDDDD SUPNNNNNN     RDLONG  D,PTR     read long at PTR into D               3..10
000010 Z11 0 CCCC DDDDDDDDD SSSSSSSSS     RDLONGC D,S       read cached long at S into D      1, 3..10
000010 Z11 1 CCCC DDDDDDDDD SUPNNNNNN     RDLONGC D,PTR     read cached long at PTR into D    1, 3..10
```

```
000011 000 0 CCCC DDDDDDDDD 010110000     WRQUAD   D        write QUADs at D                    1..8
000011 001 1 CCCC SUPNNNNNN 010110000     WRQUAD   PTR      write QUADs at PTR                  1..8
000011 000 0 CCCC DDDDDDDDD 010110001     RDQUAD   D        read quad at D into QUADs           1..8
000011 001 1 CCCC SUPNNNNNN 010110001     RDQUAD   PTR      read quad at PTR into QUADs         1..8
000011 010 0 CCCC DDDDDDDDD 010110001     RDQUADC  D        read cached quad at D into QUADs     1, 1..8
000011 011 1 CCCC SUPNNNNNN 010110001     RDQUADC  PTR      read cached quad at PTR into QUADs   1, 1..8
                                          RDQUADC           Conditionally read into QUADs from hub memory at D
-------------------------------------------------------------------------------------------------------------


PTR EXPRESSIONS:

INDEX   -32 .. +31      Simple offset
INDEX     0 ..  31      ++ Auto-increments range
INDEX     0 ..  32      -- Auto-decrement range
SCALE     1             BYTE
SCALE     2             WORD
SCALE     4             LONG
SCALE    16             QUAD

    INDEX = -32..+31 for simple offsets, 0..31 for ++´s, or 0..32 for --´s
    SCALE = 1 for byte, 2 for word, 4 for long, or 16 for quad

    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)
    NNNNNN = INDEX
    nnnnnn = -INDEX


    SUP NNNNNN     PTR expression
    -----------------------------------------------------------------------------
    000 000000     PTRA               'use PTRA
    100 000000     PTRB               'use PTRB
    011 000001     PTRA++             'use PTRA,                PTRA += SCALE
    111 000001     PTRB++             'use PTRB,                PTRB += SCALE
    011 111111     PTRA--             'use PTRA,                PTRA -= SCALE
    111 111111     PTRB--             'use PTRB,                PTRB -= SCALE
```

```
    010 000001     ++PTRA             'use PTRA + SCALE,        PTRA += SCALE
    110 000001     ++PTRB             'use PTRB + SCALE,        PTRB += SCALE
    010 111111     --PTRA             'use PTRA - SCALE,        PTRA -= SCALE
    110 111111     --PTRB             'use PTRB - SCALE,        PTRB -= SCALE

    000 NNNNNN     PTRA[INDEX]        'use PTRA + INDEX*SCALE
    100 NNNNNN     PTRB[INDEX]        'use PTRB + INDEX*SCALE
    011 NNNNNN     PTRA++[INDEX]      'use PTRA,                PTRA += INDEX*SCALE
    111 NNNNNN     PTRB++[INDEX]      'use PTRB,                PTRB += INDEX*SCALE
    011 nnnnnn     PTRA--[INDEX]      'use PTRA,                PTRA -= INDEX*SCALE
    111 nnnnnn     PTRB--[INDEX]      'use PTRB,                PTRB -= INDEX*SCALE
    010 NNNNNN     ++PTRA[INDEX]      'use PTRA + INDEX*SCALE,  PTRA += INDEX*SCALE
    110 NNNNNN     ++PTRB[INDEX]      'use PTRB + INDEX*SCALE,  PTRB += INDEX*SCALE
    010 nnnnnn     --PTRA[INDEX]      'use PTRA - INDEX*SCALE,  PTRA -= INDEX*SCALE
    110 nnnnnn     --PTRB[INDEX]      'use PTRB - INDEX*SCALE,  PTRB -= INDEX*SCALE


Examples:

000000 Z01 1 CCCC DDDDDDDDD 000000000     RDBYTE  D,PTRA         'read byte at PTRA into D
000001 000 1 CCCC DDDDDDDDD 111000001     WRWORD  D,PTRB++       'write lower word in D at PTRB,      PTRB += 2
000010 Z01 1 CCCC DDDDDDDDD 011111111     RDLONG  D,PTRA--       'read long at PTRA into D,           PTRA -= 4
000011 001 1 CCCC 110000001 010110001     RDQUAD  ++PTRB         'read quad at PTRB+16 into QUADs,    PTRB += 16
000000 000 1 CCCC DDDDDDDDD 010111111     WRBYTE  D,--PTRA       'write lower byte in D at PTRA-1,    PTRA -= 1

000001 000 1 CCCC DDDDDDDDD 100000111     WRWORD  D,PTRB[7]      'write lower word in D to PTRB+7*2
000010 Z11 1 CCCC DDDDDDDDD 011001111     RDLONGC D,PTRA++[15]   'read cached long at PTRA into D,    PTRA += 15*4
000011 001 1 CCCC 111111101 010110000     WRQUAD  PTRB--[3]      'write QUADs at PTRB,                PTRB -= 3*16
000000 000 1 CCCC DDDDDDDDD 010000110     WRBYTE  D,++PTRA[6]    'write lower byte in D to PTRA+6*1,  PTRA += 6*1
000001 Z01 1 CCCC DDDDDDDDD 110110110     RDWORD  D,--PTRB[10]   'read word at PTRB-10*2 into D,      PTRB -= 10*2


Bytes, words, longs, and quads are addressed as follows:

    for WRBYTE/RDBYTE/RDBYTEC, address = %XXXXXXXXXXXXXXXXX (bits 16..0 are used)
    for WRWORD/RDWORD/RDWORDC, address = %XXXXXXXXXXXXXXXX- (bits 16..1 are used)
    for WRLONG/RDLONG/RDLONGC, address = %XXXXXXXXXXXXXXX-- (bits 16..2 are used)
    for WRQUAD/RDQUAD/RDQUADC, address = %XXXXXXXXXXXXX---- (bits 16..4 are used)
```

```
address  byte  word    long        quad
-------------------------------------------------------------------
00000-   50   *7250   *706F7250   *0C7CCC03_0C7C2000_20302E32_706F7250
00001-   72    7250    706F7250    0C7CCC03_0C7C2000_20302E32_706F7250
00002-   6F   *706F    706F7250    0C7CCC03_0C7C2000_20302E32_706F7250
00003-   70    706F    706F7250    0C7CCC03_0C7C2000_20302E32_706F7250
00004-   32   *2E32   *20302E32    0C7CCC03_0C7C2000_20302E32_706F7250
00005-   2E    2E32    20302E32    0C7CCC03_0C7C2000_20302E32_706F7250
00006-   30   *2030    20302E32    0C7CCC03_0C7C2000_20302E32_706F7250
00007-   20    2030    20302E32    0C7CCC03_0C7C2000_20302E32_706F7250
00008-   00   *2000   *0C7C2000    0C7CCC03_0C7C2000_20302E32_706F7250
00009-   20    2000    0C7C2000    0C7CCC03_0C7C2000_20302E32_706F7250
0000A-   7C   *0C7C    0C7C2000    0C7CCC03_0C7C2000_20302E32_706F7250
0000B-   0C    0C7C    0C7C2000    0C7CCC03_0C7C2000_20302E32_706F7250
0000C-   03   *CC03   *0C7CCC03    0C7CCC03_0C7C2000_20302E32_706F7250
0000D-   CC    CC03    0C7CCC03    0C7CCC03_0C7C2000_20302E32_706F7250
0000E-   7C   *0C7C    0C7CCC03    0C7CCC03_0C7C2000_20302E32_706F7250
0000F-   0C    0C7C    0C7CCC03    0C7CCC03_0C7C2000_20302E32_706F7250
00010-   45   *FE45   *0DC1FE45   *0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00011-   FE    FE45    0DC1FE45    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00012-   C1   *0DC1    0DC1FE45    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00013-   0D    0DC1    0DC1FE45    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00014-   E3   *B6E3   *0CFCB6E3    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00015-   B6    B6E3    0CFCB6E3    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00016-   FC   *0CFC    0CFCB6E3    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00017-   0C    0CFC    0CFCB6E3    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00018-   01   *C601   *0C7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
00019-   C6    C601    0C7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001A-   7C   *0C7C    0C7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001B-   0C    0C7C    0C7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001C-   01   *C601   *0D7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001D-   C6    C601    0D7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001E-   7C   *0D7C    0D7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45
0001F-   0D    0D7C    0D7CC601    0D7CC601_0C7CC601_0CFCB6E3_0DC1FE45

* new word/long/quad
```

```
PTRA/PTRB INSTRUCTIONS
----------------------

Each COG has two 17-bit pointers, PTRA and PTRB, which can be read, written, modified,
and used to access HUB memory.

At COG startup, the PTRA and PTRB registers are initialized as follows:

    PTRA = %X_XXXXXXXX_XXXXXXXX, data from launching COG, usually a pointer
    PTRB = %X_XXXXXXXX_XXXXXX00, long address in HUB where COG code was loaded from

    when COG starts, PTRA = PAR
    PTRB = address of COG image

instructions                                                                              clocks
--------------------------------------------------------------------------------------------------
000011 ZCR 1 CCCC DDDDDDDDD 000010010     GETPTRA D          get PTRA into D, C = PTRA[16]      1
000011 ZCR 1 CCCC DDDDDDDDD 000010011     GETPTRB D          get PTRB into D, C = PTRB[16]      1

000011 000 1 CCCC DDDDDDDDD 010110010     SETPTRA D          set PTRA to D                      1
000011 001 1 CCCC nnnnnnnnn 010110010     SETPTRA #n         set PTRA to 0..511                 1
000011 000 1 CCCC DDDDDDDDD 010110011     SETPTRB D          set PTRB to D                      1
000011 001 1 CCCC nnnnnnnnn 010110011     SETPTRB #n         set PTRB to 0..511                 1

000011 000 1 CCCC DDDDDDDDD 010110100     ADDPTRA D          add D into PTRA                    1
000011 001 1 CCCC nnnnnnnnn 010110100     ADDPTRA #n         add 0..511 into PTRA               1
000011 000 1 CCCC DDDDDDDDD 010110101     ADDPTRB D          add D into PTRB                    1
000011 001 1 CCCC nnnnnnnnn 010110101     ADDPTRB #n         add 0..511 into PTRB               1

000011 000 1 CCCC DDDDDDDDD 010110110     SUBPTRA D          subtract D from PTRA               1
000011 001 1 CCCC nnnnnnnnn 010110110     SUBPTRA #n         subtract 0..511 from PTRA          1
000011 000 1 CCCC DDDDDDDDD 010110111     SUBPTRB D          subtract D from PTRB               1
000011 001 1 CCCC nnnnnnnnn 010110111     SUBPTRB #n         subtract 0..511 from PTRB          1
--------------------------------------------------------------------------------------------------



QUAD-RELATED INSTRUCTIONS
```

```
-------------------------

Each COG has four QUAD registers which form a 128-bit conduit between the HUB memory and the COG .
This conduit can transfer four longs every 8 clocks via the WRQUAD/RDQUAD instructions. It can
also be used as a 4-long/8-word/16-byte read cache, utilized by RDBYTEC/RDWORDC/RDLONGC/RDQUADC .

Initially hidden, these QUAD registers are mappable into COG register space by using the SETQUAD
instruction to set an address where the base register is to appear, with the other three registers
following. To hide the QUAD registers, use SETQUAD to set an address which is $1F8, or higher.
SETQUAZ works just like SETQUAD, but also clears the four QUAD registers.


instructions                                                                                  clocks
------------------------------------------------------------------------------------------------------------
000011 000 1 CCCC 000000000 000001000     CACHEX       'invalidate cache                                   1
000011 Z01 1 CCCC DDDDDDDDD 000010001     GETTOPS D    'get top bytes of QUADs into D                      1
000011 000 1 CCCC DDDDDDDDD 011100010     SETQUAD D    'set QUAD base address to D                         1
000011 001 1 CCCC nnnnnnnnn 011100010     SETQUAD #n   'set QUAD base address to 0..511                    1
000011 010 1 CCCC DDDDDDDDD 011100010     SETQUAZ D    'set QUAD base address to D and clears the QUAD registers.      1
000011 011 1 CCCC nnnnnnnnn 011100010     SETQUAZ #n   'set QUAD base address to 0..511 and clears the QUAD registers. 1
------------------------------------------------------------------------------------------------------------

You can start the QUAD´s at any register now and clear them at the same time, if you want.


HUB 'CONTROL INSTRUCTIONS
------------------------

These instructions are used to control HUB circuits and cogs.

HUB instructions must wait for their COG's HUB cycle, which comes once every 8 clocks. In cases where
there is no result to wait for (ZCR = %000), these instructions complete on the HUB cycle, making
them take 1..8 clocks, depending on where the HUB cycle is in relation to the instruction. In cases
where a result is anticipated (ZCR <> %000), these instructions complete on the 1st clock after the
HUB cycle, making them take 2..9 clocks.


COGINIT D,S
-----------
```

COGINIT is used to start cogs. Any COG can be (re)started, whether it is idle or running. A COG can even execute a COGINIT to restart itself with a new program.

COGINIT uses D to specify a long address in HUB memory that is the start of the program that is to be loaded into a COG, while S is a 17-bit parameter (usually an address) that will be conveyed to PTRA of the started COG . PTRB of the started COG will be set to the start address of its program that was loaded from HUB memory.

SETCOG must be executed before COGINIT to set the number of the COG to be started (0..7). If SETCOG sets a value with bit 3 set (%1xxx), this will cause the next idle COG to be started when COGINIT is executed, with the number of the COG started being returned in D, and the C flag returning 0 if okay, or 1 if no idle COG was available. Upon COG startup, SETCOG is initialized to %0000.

When a COG is started, $1F8 contiguous longs are read from HUB memory and written to COG registers $000..$1F7. The COG will then begin execution at $000. This process takes 1,016 clocks.

Example:

```
        COGID   COGNUM          'what COG am I?
        SETCOG  COGNUM          'set my COG number
        COGINIT COGPGM,COGPTR   'restart me with the ROM Monitor

COGPGM  LONG    $0070C          'address of the ROM Monitor
COGPTR  LONG    90<<9 + 91      'tx = P90, rx = P91

COGNUM  RES     1

'If you want to inspect hub memory after your program has run,
' just put the following code at the end of your program:
```

Code:

```
        coginit monitor_pgm,monitor_ptr 'relaunch cog0 with monitor

monitor_pgm long    $70C            'monitor program address
monitor_ptr long    90<<9 + 91      'monitor parameter (conveys tx/rx pins)


'This will launch the ROM Monitor and let you view what your program did to hub memory.
```

```
' The monitor only affects the hub memory when you give it a command to do so. So, when
' the monitor starts up, hub memory is just as your program left it, ready to be inspected.

CLKSET  D
---------

CLKSET writes the lower 9 bits of D to the HUB clock register:

%R_MMMM_XX_SS

R = 1 for hardware reset, 0 for continued operation

MMMM = PLL multiplying factor for XI pin input:
        % 0000 for PLL disabled
        % 0001..% 1111 for 2..16 multiply (XX must be set for XI input or XI/XO crystal oscillator)

MMMM = PLL mode:
        % 0000 for disabled, else XX must be set for XI input or XI/XO crystal oscillator
        % 0001 for multiply XI by 2
        % 0010 for multiply XI by 3
        % 0011 for multiply XI by 4
        % 0100 for multiply XI by 5
        % 0101 for multiply XI by 6
        % 0110 for multiply XI by 7
        % 0111 for multiply XI by 8
        % 1000 for multiply XI by 9
        % 1001 for multiply XI by 10
        % 1010 for multiply XI by 11
        % 1011 for multiply XI by 12
        % 1100 for multiply XI by 13
        % 1101 for multiply XI by 14
        % 1110 for multiply XI by 15
        % 1111 for multiply XI by 16

XX = XI/XO pin mode:
        00 for XI reads low, XO floats
        01 for XI input, XO floats
        10 for XI/XO crystal oscillator with 15pF internal loading and 1M-ohm feedback
        11 for XI/XO crystal oscillator with 30pF internal loading and 1M-ohm feedback
```

```
SS = Clock selector:
        00 for RCFAST (~20MHz)
        01 for RCSLOW (~20KHz)
        10 for XTAL (10MHz-20MHz)
        11 for PLL


Because the the clock register is cleared to % 0_0000_00_00 on reset, the chip starts up in RCFAST mode
with both the crystal oscillator and the PLL disabled. Before switching to XTAL or PLL mode from RCFAST
or RCSLOW, the crystal oscillator must be enabled and given 10ms to stabilize. The PLL stabilizes within
10us, 'so it can be enbled at the sime time as the crystal oscillator. Once the crystal is stabilized, you
can switch between XTAL and RCFAST/RCSLOW without any stability concerns. If the PLL is also enabled, you
can switch freely among PLL, XTAL, and RCFAST/RCSLOW modes. You can change the PLL multiplier while being
in PLL mode, but beware that some frequency overshoot and undershoot will occur as the PLL settles to its
'new frequency. This only poses a hardware problem if you are switching upwards and the resulting overshoot
'might exceed the speed limit of the chip.


COGID   D
---------

COGID returns the number of the COG (0..7) into D.


COGSTOP D
---------

COGSTOP stops the COG specified in D (0..7).


LOCKNEW D
LOCKRET D
LOCKSET D
LOCKCLR D
---------

There are eight semaphore locks available in the chip which can be borrowed with LOCKNEW, returned with
LOCKRET, set with LOCKSET, and cleared with LOCKCLR.
```

While any COG can set or clear any lock without using LOCKNEW or LOCKRET, LOCKNEW and LOCKRET are provided so that COG programs have a dynamic and simple means of acquiring and relinquishing the locks at run-time.

When a lock is set with LOCKSET, its state is set to 1 and its prior state is returned in C. LOCKCLR works the same way, but clears the lock's state to 0. By having the HUB perform the atomic operation of setting/clearing and reporting the prior state, cogs can utilize locks to insure that only one COG has permission to do something at once. If a lock starts out cleared and multiple cogs vie for the lock by doing a 'LOCKSET locknum wc', the COG to get C=0 back 'wins' and he can have exclusive access to some shared resource while the other cogs get C=1 back. When the winning COG is done, he can do a 'LOCKCLR locknum' to clear the lock and give another COG the opportunity to get C=0 back.

LOCKNEW returns the next available lock into D, with C=1 if no lock was free.

LOCKRET frees the lock in D so that it can be checked out again by LOCKNEW .

LOCKSET sets the lock in D and returns its prior state in C.

LOCKCLR clears the lock in D and returns its prior state in C.

```
instructions                                                                                    clocks
------------------------------------------------------------------------------------------------------
000011 ZCR 0 CCCC DDDDDDDDD SSSSSSSSS     COGINIT D,S     'launch COG at D, COG PTRA = S     1..9
000011 000 1 CCCC DDDDDDDDD 000000000     CLKSET   D      'set clock to D                    1..8
000011 001 1 CCCC DDDDDDDDD 000000001     COGID    D      'get COG number into D             2..9
000011 000 1 CCCC DDDDDDDDD 000000011     COGSTOP D       'stop COG in D                     1..8
000011 ZC1 1 CCCC DDDDDDDDD 000000100     LOCKNEW D       'get new lock into D, C = busy     2..9
000011 000 1 CCCC DDDDDDDDD 000000101     LOCKRET D       'return lock in D                  1..8
000011 0C0 1 CCCC DDDDDDDDD 000000110     LOCKSET D       'set lock in D, C = prev state     1..9
000011 0C0 1 CCCC DDDDDDDDD 000000111     LOCKCLR D       'clear lock in D, C = prev state   1..9
------------------------------------------------------------------------------------------------------
```

'INDIRECT REGISTERS
'------------------

Each COG has two indirect registers: INDA and INDB. They are located at $1F6 and $1F7.

```
By using INDA or INDB for D or S, the register pointed at by INDA or INDB is addressed.

INDA and INDB each have three hidden 9-bit 'registers associated with them: the pointer, the bottom limit, and
'the top limit. The bottom and top limits are inclusive values which set automatic wrapping boundaries for the
pointer. This way, circular buffers can be established within COG RAM and accessed using simple INDA/INDB
references.

SETINDA/SETINDB/SETINDS is used to set or adjust the pointer value(S) while forcing the associated bottom and
top limit(S) to $000 and $1FF, respectively.

FIXINDA/FIXINDB/FIXINDS sets the pointer(S) to an inital value, while setting the bottom limit(s) to the
lower of the initial and terminal values and the top limit(S) to the higher.

'Because indirect addressing occurs very early in the pipeline and indirect pointers are affected earlier than
the final stage where the conditional bit field (CCCC) normally comes into use, the CCCC field is repurposed
for indirect operations. The top two bits of CCCC are used for indirect D and the bottom two bits are used
for indirect S . All 'instructions which use indirect registers will execute unconditionally, regardless of the
CCCC 'bits.

Here is the INDA/INDB usage scheme which repurposes the CCCC field:

OOOOOO ZCR I CCCC DDDDDDDDD SSSSSSSSS
-------------------------------------
xxxxxx xxx x 00xx 111110110 xxxxxxxxx       D = INDA         'use INDA
xxxxxx xxx x 00xx 111110111 xxxxxxxxx       D = INDB         'use INDB
xxxxxx xxx x 01xx 111110110 xxxxxxxxx       D = INDA++       'use INDA,      INDA += 1
xxxxxx xxx x 01xx 111110111 xxxxxxxxx       D = INDB++       'use INDB,      INDB += 1
xxxxxx xxx x 10xx 111110110 xxxxxxxxx       D = INDA--       'use INDA,      INDA -= 1
xxxxxx xxx x 10xx 111110111 xxxxxxxxx       D = INDB--       'use INDB       INDB -= 1
xxxxxx xxx x 11xx 111110110 xxxxxxxxx       D = ++INDA       'use INDA+1,    INDA += 1
xxxxxx xxx x 11xx 111110111 xxxxxxxxx       D = ++INDB       'use INDB+1,    INDB += 1

xxxxxx xxx 0 xx00 xxxxxxxxx 111110110       S = INDA         'use INDA
xxxxxx xxx 0 xx00 xxxxxxxxx 111110111       S = INDB         'use INDB
xxxxxx xxx 0 xx01 xxxxxxxxx 111110110       S = INDA++       'use INDA,      INDA += 1
xxxxxx xxx 0 xx01 xxxxxxxxx 111110111       S = INDB++       'use INDB,      INDB += 1
xxxxxx xxx 0 xx10 xxxxxxxxx 111110110       S = INDA--       'use INDA,      INDA -= 1
xxxxxx xxx 0 xx10 xxxxxxxxx 111110111       S = INDB--       'use INDB       INDB -= 1
```

```
xxxxxx xxx 0 xx11 xxxxxxxxx 111110110        S = ++INDA      'use INDA+1,    INDA += 1
xxxxxx xxx 0 xx11 xxxxxxxxx 111110111        S = ++INDB      'use INDB+1,    INDB += 1


If both D and S are the same indirect register, the two 2-bit fields in CCCC are OR'd together to get the
post-modifier effect:

101000 001 0 0011 111110110 111110110        MOV INDA,++INDA    'Move @INDA+1 into @INDA,   INDA += 1
100000 001 0 1100 111110111 111110111        ADD ++INDB,INDB    'Add @INDB into @INDB+1,    INDB += 1

Note that only '++INDx,INDx'/'INDx,++INDx' combinations can address different registers from the same INDx.


Here are the instructions which are used to set the pointer and limit values for INDA and INDB:

instructions *                                                            clocks    Descrinption
--------------------------------------------------------------------------------------------------------
-
111000 000 0 0001 000000000 AAAAAAAAA        SETINDA #addrA                  1    | 'Set or adjust the pointer value(s) while forcing
111000 000 0 0011 000000000 AAAAAAAAA        SETINDA ++/--deltA              1    | ' the associated bottom and top limit(s)
                                                                                  | ' to $000 and $1FF, respectively.
111000 000 0 0100 BBBBBBBBB 000000000        SETINDB #addrB                  1    | * addrA/addrB/terminal/initial
111000 000 0 1100 BBBBBBBBB 000000000        SETINDB ++/--deltB              1    | ' = register address (0..511),
                                                                                  | ' deltA/deltB = 9-bit signed delta --256..++255
111000 000 0 0101 BBBBBBBBB AAAAAAAAA        SETINDS #addrB,#addrA           1    |
111000 000 0 0111 BBBBBBBBB AAAAAAAAA        SETINDS #addrB,++/--deltA       1    |   AAAAAAAAA   addrA
111000 000 0 1101 BBBBBBBBB AAAAAAAAA        SETINDS ++/--deltB,#addrA       1    |   BBBBBBBBB   addrB
111000 000 0 1111 BBBBBBBBB AAAAAAAAA        SETINDS ++/--deltB,++/--deltA   1    |   TTTTTTTTT   terminal
                                                                                  |   IIIIIIIII   initial
111001 000 0 0001 TTTTTTTTT IIIIIIIII        FIXINDA #terminal,#initial      1    |
111001 000 0 0100 TTTTTTTTT IIIIIIIII        FIXINDB #terminal,#initial      1    |
111001 000 0 0101 TTTTTTTTT IIIIIIIII        FIXINDS #terminal,#initial      1    |
--------------------------------------------------------------------------------------------------------
* addrA/addrB/terminal/initial = register address (0..511),
  deltA/deltB = 9-bit signed delta --256..++255

INDIRECT POINTER Examples:

111000 000 0 0001 000000000 000000101        SETINDA #5        'INDA = 5, bottom = 0, top = 511
```

```
111000 000 0 0011 000000000 000000011        SETINDA ++3        'INDA += 3, bottom = 0, top = 511
111000 000 0 1100 111111100 000000000        SETINDB --4        'INDB -= 4, bottom = 0, top = 511
111000 000 0 0111 000000111 000001000        SETINDS #7,++8     'INDB = 7, INDA += 8, bottoms = 0, tops = 511

111001 000 0 0001 000001111 000001000        FIXINDA #15,#8     'INDA = 8, bottom = 8, top = 15
111001 000 0 0100 000010000 000011111        FIXINDB #16,#31    'INDB = 31, bottom = 16, top = 31
111001 000 0 0101 001100011 000110010        FIXINDS #99,#50    'INDA/INDB = 50, bottoms = 50, tops = 99
```

STACK RAM

```
  When the video generator is not in use the CLUT/RAM may be used as a general-purpose memory scratch space,
  or as a 256 Long FIFO buffer, or as a call stack and evaluation stack (at the same time).
  The CLUT/RAM has two pointers used to index it called SPA and SPB .

Each COG has a 256-long STACK RAM that is accessible via push and pop operations.

There are two STACK pointers called SPA and SPB which are used to address the STACK memory.
' Aside from automatically incrementing and decrementing on pushes and pops,
 SPA and SPB 'can be set, added to, subtracted from, read back, and checked:

SETSPA  D/#n      set SPA
SETSPB  D/#n      set SPB
ADDSPA  D/#n      add to SPA
ADDSPB  D/#n      add to SPB
SUBSPA  D/#n      subtract from SPA
SUBSPB  D/#n      subtract from SPB
GETSPA  D         get SPA, SPA==0 into Z, SPA.7 into C
GETSPB  D         get SPB, SPB==0 into Z, SPB.7 into C
GETSPD  D         get SPA minus SPB, SPA==SPB into Z, SPA<SPB into C
CHKSPA            check SPA, SPA==0 into Z, SPA.7 into C
CHKSPB            check SPB, SPB==0 into Z, SPB.7 into C
CHKSPD            check SPA minus SPB, SPA==SPB into Z, SPA<SPB into C

'Data can be pushed and popped in both normal and reverse directions:

PUSHA   D/#n      push using SPA
PUSHB   D/#n      push using SPB
```

```
PUSHAR  D/#n       push using SPA, use pop addressing
PUSHBR  D/#n       push using SPB, use pop addressing
POPA    D          pop using SPA
POPB    D          pop using SPB
POPAR   D          pop using SPA, use push addressing
POPBR   D          pop using SPB, use push addressing

'Aside from data, the program counter and flags can be pushed and popped using calls and returns:

CALLA   D/#n       call using SPA
CALLB   D/#n       call using SPB
CALLAD  D/#n       call using SPA,   'delay branch until three trailing instructions executed
CALLBD  D/#n       call using SPB,   'delay branch until three trailing instructions executed
RETA               return using SPA
RETB               return using SPB
RETAD              return using SPA, 'delay branch until three trailing instructions executed
RETBD              return using SPB, 'delay branch until three trailing instructions executed

The STACK RAM´s contents are undefined at COG start.

instructions (STACK RAM access is shown as [SPx++] and [--SPx])                           clocks
-------------------------------------------------------------------------------------------------
000011 ZC0 1 CCCC 000000000 000010101        CHKSPD          SPA==SPB into Z, SPA<SPB into C   1

000011 ZC1 1 CCCC DDDDDDDDD 000010101        GETSPD  D       SPA-SPB into D, Z/C as CHKSPD     1
                                                            'Stores ((SPA - SPB) & 0x7F) in register "D (0-511)". FOR FIFO MODE.

000011 ZC0 1 CCCC 000000000 000010110        CHKSPA          SPA==0 into Z, SPA.7 into C       1

000011 ZC1 1 CCCC DDDDDDDDD 000010110        GETSPA  D       SPA into D, Z/C as CHKSPA         1
                                                            'Stores SPA in register "D (0-511)".

000011 ZC0 1 CCCC 000000000 000010111        CHKSPB          SPB==0 into Z, SPB .7 into C       1

000011 ZC1 1 CCCC DDDDDDDDD 000010111        GETSPB  D       SPB into D, Z/C as CHKSPB         1
                                                            'Stores SPB in register "D (0-511)".

000011 ZC1 1 CCCC DDDDDDDDD 000011000        POPAR   D       read [SPA++] into D, MSB into C   1
                                                            'Store CLUT[SPA] in register "D (0-511)" and then increment SPA.
```

```
000011 ZC1 1 CCCC DDDDDDDDD 000011001       POPBR   D        read [SPB++] into D, MSB into C    1
                                                            'Store CLUT[SPB] in register "D (0-511)" and then increment SPA.

000011 ZC1 1 CCCC DDDDDDDDD 000011010       POPA    D        read [--SPA] into D, MSB into C    1
                                                            'Decrement SPA and then store CLUT[SPA] in register "D (0-511)".
000011 ZC1 1 CCCC DDDDDDDDD 000011011       POPB    D        read [--SPB] into D, MSB into C    1
                                                            'Decrement SPB and then store CLUT[SPB] in register "D (0-511)".

000011 ZC0 1 CCCC 000000000 000011100       RETA             read [--SPA] into Z/C/PC*          4
                                                            'Decrement SPA and then jump to instruction (CLUT[SPA] & 0x1FF).
                                                            'Flush pipeline before jump – results in a two-cycle loss.
000011 ZC0 1 CCCC 000000000 000011101       RETB             read [--SPB] into Z/C/PC*          4
                                                            'Decrement SPB and then jump to instruction (CLUT[SPB] & 0x1FF).
                                                            'Flush pipeline before jump – results in a two-cycle loss.

000011 ZC0 1 CCCC 000000000 000011110       RETAD            read [--SPA] into Z/C/PC*          1
                                                            'Decrement SPA and then jump to instruction (CLUT[SPA] & 0x1FF).
                                                            'Do not flush pipeline before jump – must be executed two
                                                            'instructions before intended jump space.
000011 ZC0 1 CCCC 000000000 000011111       RETBD            read [--SPB] into Z/C/PC*          1
                                                            'Decrement SPB and then jump to instruction (CLUT[SPB] & 0x1FF).
                                                            'Do not flush pipeline before jump – must be executed two
                                                            'instructions before intended jump space.

000011 000 1 CCCC DDDDDDDDD 010100010       SETSPA  D        set SPA to D                       1
                                                            'Set SPA to register "D (0-511)".
000011 001 1 CCCC 0nnnnnnnn 010100010       SETSPA  #n       set SPA to n                       1
                                                            'Set SPA to register "n (0-511)".
000011 000 1 CCCC DDDDDDDDD 010100011       SETSPB  D        set SPB to D                       1
                                                            'Set SPB to register "D (0-511)".
000011 001 1 CCCC 0nnnnnnnn 010100011       SETSPB  #n       set SPB to n                       1
                                                            'Set SPB to register "n (0-511)".

000011 000 1 CCCC DDDDDDDDD 010100100       ADDSPA  D        add D into SPA                     1
                                                            'Add to SPA register "D (0-511)"
000011 001 1 CCCC 0nnnnnnnn 010100100       ADDSPA  #n       add n into SPA                     1
                                                            'Add to SPA register "n (0-511)"
000011 000 1 CCCC DDDDDDDDD 010100101       ADDSPB  D        add D into SPB                     1
                                                            'Add to SPB register "D (0-511)"
```

```
000011 001 1 CCCC 0nnnnnnnn 010100101        ADDSPB   #n        add n into SPB                    1
                                                                'Add to SPB register "n (0-511)"

000011 000 1 CCCC DDDDDDDDD 010100110        SUBSPA   D         subtract D from SPA               1
                                                                'Subtract from SPA register "D (0-511)"
000011 001 1 CCCC 0nnnnnnnn 010100110        SUBSPA   #n        subtract n from SPA               1
                                                                'Subtract from SPA register "n (0-511)"
000011 000 1 CCCC DDDDDDDDD 010100111        SUBSPB   D         subtract D from SPB               1
                                                                'Subtract from SPB register "D (0-511)"
000011 001 1 CCCC 0nnnnnnnn 010100111        SUBSPB   #n        subtract n from SPB               1
                                                                'Subtract from SPB register "n (0-511)"

000011 000 1 CCCC DDDDDDDDD 010101000        PUSHAR   D         write D into [--SPA]              1 **
                                                                'Decrement SPA and then store register "D (0 511)"
000011 001 1 CCCC nnnnnnnnn 010101000        PUSHAR   #n        write n into [--SPA]              1 **
                                                                'Decrement SPA and then store register "n (0 511)"
000011 000 1 CCCC DDDDDDDDD 010101001        PUSHBR   D         write D into [--SPB]              1 **
                                                                'Decrement SPB and then store register "D (0-511)"
000011 001 1 CCCC nnnnnnnnn 010101001        PUSHBR   #n        write n into [--SPB]              1 **
                                                                'Decrement SPB and then store register "n (0-511)"

000011 000 1 CCCC DDDDDDDDD 010101010        PUSHA    D         write D into [SPA++]              1 **
                                                                'Store register "D (0-511)" in CLUT[SPA] and then increment SPA.
000011 001 1 CCCC nnnnnnnnn 010101010        PUSHA    #n        write n into [SPA++]              1 **
                                                                'Store register "n (0-511)" in CLUT[SPA] and then increment SPA.
000011 000 1 CCCC DDDDDDDDD 010101011        PUSHB    D         write D into [SPB++]              1 **
                                                                'Store register "D (0-511)" in CLUT[SPB] and then increment SPB.
000011 001 1 CCCC nnnnnnnnn 010101011        PUSHB    #n        write n into [SPB++]              1 **
                                                                'Store register "n (0-511)" in CLUT[SPB] and then increment SPB.

000011 000 1 CCCC DDDDDDDDD 010101100        CALLA    D         write Z/C/PC* into [SPA++], PC=D  4 **
                                                                'Store the program counter (PC) in CLUT[SPA] and then increment
                                                                ' SPA and then jump to the address in register "D (0-511)"
000011 001 1 CCCC nnnnnnnnn 010101100        CALLA    #n        write Z/C/PC* into [SPA++], PC=n  4 **
                                                                'Store the program counter (PC) in CLUT[SPA] and then increment
                                                                ' SPA and then jump to the address in register "n (0-511)".
                                                                ' Flush pipeline before jump – results in a two-cycle loss.
000011 000 1 CCCC DDDDDDDDD 010101101        CALLB    D         write Z/C/PC* into [SPB++], PC=D  4 **
                                                                'Store the program counter (PC) in CLUT[SPB] and then increment
```

```
                                                                ' SPB and then jump to the address in register “D (0-511)”
000011 001 1 CCCC nnnnnnnnn 010101101        CALLB   #n          write Z/C/PC* into [SPB++], PC=n  4 **
                                                                'Store the program counter (PC) in CLUT[SPB] and then increment
                                                                ' SPB and then jump to the address in register “n (0-511)”.
                                                                ' Flush pipeline before jump – results in a two-cycle loss.

000011 000 1 CCCC DDDDDDDDD 010101110        CALLAD  D           write Z/C/PC* into [SPA++], PC=D  1 **
                                                                'Store the program counter (PC) in CLUT[SPA] and then increment
                                                                ' SPA and then jump to the address in register “D (0-511)”
000011 001 1 CCCC nnnnnnnnn 010101110        CALLAD  #n          write Z/C/PC* into [SPA++], PC=n  1 **
                                                                'Store the program counter (PC) in CLUT[SPA] and then increment
                                                                ' SPA and then jump to the address in register “n (0-511)”
000011 000 1 CCCC DDDDDDDDD 010101111        CALLBD  D           write Z/C/PC* into [SPB++], PC=D  1 **
                                                                'Store the program counter (PC) in CLUT[SPB] and then increment
                                                                ' SPB and then jump to the address in register “D (0-511)”
000011 001 1 CCCC nnnnnnnnn 010101111        CALLBD  #n          write Z/C/PC* into [SPB++], PC=n  1 **
                                                                'Store the program counter (PC) in CLUT[SPB] and then increment
                                                                ' SPB and then jump to the address in register “n (0-511)”
---------------------------------------------------------------------------------------------------
* bit 10 is Z, bit 9 is C, bits 8..0 are PC, upper bits are ignored or cleared
** if a STACK RAM write is immediately followed by a STACK RAM read, add one clock




MULTI-TASKING
-------------

Each COG has four sets of flags and program counters (Z/C/PC), constituting four unique Tasks that
can execute and switch on each instruction cycle.

'At COG startup, the tasks are initialized as follows:


TASK Z  C  PC
---------------
0    0  0  $000
1    0  0  $001
2    0  0  $002
```

```
3    0  0  $003


There are 16 rotating time slots in the TASK register that determine TASK sequence. Initially, all
time slots are set to 0, causing TASK 0 to execute exclusively, starting at address $000:


   time slots:   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
                  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
TASK register:  %00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00


The two LSB's of TASK always determine which TASK will execute next. After each instruction cycle,
the TASK register is rotated right by two bits, recycling slot 0 to slot 15 and getting the next TASK
into the 2 LSB's.


To enable other Tasks, SETTASK is used to set the TASK register:

SETTASK D                write D to the TASK register
SETTASK #n               write {n[7:0], n[7:0], n[7:0], n[7:0]} to the TASK register

If a TASK is given no time slot, it doesn't execute and its flags and PC stay at initial values.
 If a TASK is given a time slot, it will execute and its flags and PC will be updated at every instruction,
or time slot. If an active TASK´s time slots are all taken away, that TASK´s flags and PC remain in the
state where they left off, until it is given another time slot.


To immediately force any of the four PC's to a new address, JMPTASK can be used.
 JMPTASK uses a 4-bit mask to select which PC's are going to be written. Mask bits 0..3 represent PC's 0..3.
 The mask value %1010 would write PC 3 and PC 1, while %0100 would write PC 2, only.

JMPTASK D,#mask          force PC's in mask to D
JMPTASK #addr,#mask      force PC's in mask to #addr

For every PC/TASK affected by a JMPTASK instruction, all affected-TASK instructions currently in the
pipeline are cancelled. This insures that once JMPTASK executes, the next instruction from each
affected TASK will be from the new address.
```

Here is an example in which all four tasks are started and each TASK toggles an I/O pin at a different rate:

```
        ORG

        JMP     #task0          'TASK 0 begins here when the COG starts (this JMP takes 4 clocks)
        JMP     #task1          'TASK 1 begins here after TASK 0 executes SETTASK (this JMP takes 1 clock)
        JMP     #task2          'TASK 2 begins here after TASK 0 executes SETTASK (this JMP takes 1 clock)
        JMP     #task3          'TASK 3 begins here after TASK 0 executes SETTASK (this JMP takes 1 clock)

task0   SETTASK #%%3210         'enable all tasks (TASK = %11_10_01_00_11_10_01_00_11_10_01_00_11_10_01_00)

:loop   NOTP    #0              'TASK 0, toggle pin 0       (loops every 8 clocks)
        JMP     #:loop          '(this JMP takes 1 clock)

task1   NOTP    #1              'TASK 1, toggle pin 1       (loops every 12 clocks)
        NOP
        JMP     #task1          '(this JMP takes 1 clock)

task2   NOTP    #2              'TASK 2, toggle pin 2       (loops every 16 clocks)
        NOP
        NOP
        JMP     #task2          '(this JMP takes 1 clock)

task3   NOTP    #3              'TASK 3, toggle pin 3       (loops every 20 clocks)
        NOP
        NOP
        NOP
        JMP     #task3          '(this JMP takes 1 clock)
```

NOTE: When a normal branch instruction (JMP, CALL, RET, etc.) executes in the fourth and final stage of the pipeline, all instructions progressing through the lower three stages, which belong to the same TASK as the
'branch instruction, are cancelled. This inhibits execution of incidental data that was trailing the branch
'instruction.

```
The delayed branch instructions (JMPD, CALLD, RETD, etc.) don't do any pipeline instruction cancellation and
exist to provide 1-clock branches to Single-Task programs, where the three instructions following the branch
'are allowed to execute before the new instruction stream begins to execute.

For Single-Task programs, normal branches take 4 clocks: 1 clock for the branch and 3 clocks for the
'cancelled instructions to come through the pipeline before the new instruction stream begins to execute.

For multi-tasking programs that use all four tasks in sequence (ie SETTASK #%%3210), there are never any
Same-Task instructions in the pipeline that would require cancellation due to branching, so all branches
take just 1 clock.
------------------------------------------------------------------------------------------------------------



Tips for coding multi-tasking programs
--------------------------------------

While all tasks in a multi-tasking program can execute atomic instructions without any Inter-Task conflict,
remember that there's only one of each of the following COG resources and only one TASK can use it at a time:

  SPA
  SPB
  INDA
  INDB
  PTRA
  PTRB
  ACCA
  ACCB
  32x32 multiplier
  64/32 divider
  64-bit square rooter
  CORDIC computer
  CTRA
  CTRB
  VID
  PIX                          (not usable in multi-tasking, requires single-task timing)
  XFR
  SER
  REPS/REPD                    I got the REPS/REPD working with multitasking now.
                               Any task can use it, but only one task at a time.
```

```
  Bitfield mover

When writing multi-task programs, be aware that instructions that take multiple clocks will stall the
pipeline and have a ripple effect on the tasks' timing. This may be impossible to avoid, as some task
might need to access HUB memory, and those instructions are not single-clock.

The WAITCNT/WAITPEQ/WAITPNE instructions should be recoded discretely using 1-clock instructions, to
avoid stalling the pipeline for excessive amounts of time.

The following instructions (WC versions) will take 1 clock, instead of potentially many, and return 1 in
C if they were successful:

  SNDSER  D  WC       attempt to send serial
  RCVSER  D  WC       attempt to receive serial
  GETMULL D  WC       attempt to get lower multiplier result
  GETMULH D  WC       attempt to get upper multiplier result
  GETDIVQ D  WC       attempt to get divider quotient result
  GETDIVR D  WC       attempt to get divider remainder result
  GETSQRT D  WC       attempt to get square root result
  GETQX   D  WC       attempt to get CORDIC X result
  GETQY   D  WC       attempt to get CORDIC Y result
  GETQZ   D  WC       attempt to get CORDIC Z result

'Other instruction alternatives:

  POLCTRA    WC       returns 1 in C if CTRA rolled over, use instead of SYNCTRA
  POLCTRB    WC       returns 1 in C if CTRB rolled over, use instead of SYNCTRB
  POLVID     WC       returns 1 in C if WAITVID is ready, use to execute WAITVID without stalling
  PASSCNT D           jumps to itself if some amount of time has not passed, use instead of WAITCNT
  JP/JNP  D,S         jumps based on pin states, use instead of WAITPEQ/WAITPNE
  DJNZ    D,#$        loops until done, use instead of NOP D/#n

The following instructions will not work in a Multi-Tasking program:

  GETPIX              needs steady pipeline delays for perspective divider time - Single-Task only


instructions                                                                              clocks
------------------------------------------------------------------------------------------------
```

```
000011 000 1 CCCC DDDDDDDDD 01001mmmm        JMPTASK D,#mask  'Set PC's in mask to D          1
000011 001 1 CCCC nnnnnnnnn 01001mmmm        JMPTASK #n,#mask 'Set PC's in mask to 0..511     1

000011 000 1 CCCC DDDDDDDDD 011001011        SETTASK D        'Set TASK to D                  1
000011 001 1 CCCC nnnnnnnnn 011001011        SETTASK #n       'Set TASK to n[7:0] copied 4x   1
-------------------------------------------------------------------------------------------------
```

PIPELINE

Each COG has a 4-stage pipeline which all instructions progress through, in order to execute:

```
  1st stage    - Read instruction
  2nd stage    - Determine indirect/remapped D and S addresses, update INDA/INDB
  3rd stage    - Read D and S
  4th stage    - Execute instruction, write D, Z/C/PC, and any other results
```

On every clock cycle, the instruction in each stage advances to the next stage, unless the instruction in the 4th stage is stalling the pipeline because it's waiting for something (i.e. WRBYTE waits for the HUB).

To keep D and S data current within the pipeline, the resultant D from the 4th stage is passed back to the 3rd stage to substitute for any obsoleted data being read from the COG register RAM. The same is done for instruction data in the 1st stage, but there is still a two-stage gap between when a register is modified and when it can be executed:

```
        MOVD    :inst,top9         'modify instruction
        NOP                        '1...
        NOP                        '2... at least two instructions in-between
:inst   ADD     A,B                'modified instruction executes
```

Tasks that execute in at least every 3rd time slot don't need to observe this 2-instruction rule because their instructions will always be sufficiently spread apart in the pipeline.

```
When a branch instruction executes, all instructions in the pipeline belonging to that same task are
cancelled, as the program counter has changed, rendering those instructions that were following the
branch instruction invalid. A new instruction stream, beginning at the new PC value, must make its
way through the pipeline before another instruction from that task will execute. For single-task
programs, this means that branches take 4 clocks: 1 for the branch, and 3 for the cancelled
instructions in stages 1..3 to make their way through the pipeline before the new instruction stream
reaches the execution stage. For multi-tasking programs, branch delays are a function of time slot
allocation.
```

```
_____________________________ Propeller2DetailedPreliminaryFeatureList-v2.0.pdf _________________________________________
```

```
Miscellaneous Hardware

Each cog has a free running LFSR (Linear Feedback Shift Register) and System Counter that change every clock cycle.
 Each access of the LFSR taps into a 32 bit wide sequence of numbers that is traversed in a pseudo random order, for a 232 .
 The system counter counts the number of clock ticks since power up – it is a 64-bit counter, the LFSR is 32 Bits.


Table 8: 'System Counter Instructions

Machine Code                           |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcr 1 cccc ddddddddd 000001101  |   GETCNT      |   D       | Store the bottom 32 Bits of the System Counter (CNT)
                                                                   | in register “D (0-511)”. If executed again
                                                                   | (no instruction in between previous execution)
                                                                   | store the top 32 Bits of the System Counter in register
                                                                   | “D (0-511)”.
                                                                   | If a roll over occurs between accesses TOP-1 is stored.
000011 zcr 1 cccc ddddddddd 000001100  |   SUBCNT      |   D       | Subtracts the system count value when the GETCNT
                                                                   | instruction was last executed from the current system
                                                                   | count value. Results are stored in the register
```

```
                                                                        | referenced by “D (0-511)”.
000011 zcr 1 cccc ddddddddd 000010000   |   GETLFSR      |   D          | Store the LSFR value in register “D (0-511)”.

Each COG additionally has a single cycle 24-bit hardware multiplier capable of unsigned and signed multiplications.
 The multiplication also adds into a 64-bit register for MAC ops.


Table 9: 'Multiply and Accumulate Instructions

Machine Code                            |   Mnemonic     |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000100 100 i cccc ddddddddd sssssssss   |   MACA         |   D,S      | Multiply unsigned register “D (0-511)” and unsigned
                                                                        |  register “S (0-511)” or an immediate value (0-511)
                                                                        | and add to the 64-bit accumulator A.
000100 110 i cccc ddddddddd sssssssss   |   MACB         |   D,S      | Multiply unsigned register “D (0-511)” and unsigned
                                                                        |  register “S (0-511)” or an immediate value (0-511)
                                                                        | and add to the 64-bit accumulator B.
000100 zc1 i cccc ddddddddd sssssssss   |   MUL          |   D,S      | Multiply unsigned register “D (0-511)” and unsigned
                                                                        |  register “S (0-511)” or an immediate value (0-511)
                                                                        | and store in register D.
000101 zc1 i cccc ddddddddd sssssssss   |   SCL          |   D,S      | Scale the result of the multiplication of two 24 bit
                                                                        | numbers (D,S) to fit into the 32 bit destination
                                                                        | register specified by “D (0-512)”.
000011 zcr 1 cccc 000000001 000001000   |   CLRACCA      |              | Zero Multiply Accumulator A (ACCA).
000011 zcr 1 cccc 000000010 000001000   |   CLRACCB      |              | Zero Multiply Accumulator B (ACCB).
000011 zcr 1 cccc 000000011 000001000   |   CLRACCS      |              | Zero both multiply accumulators (accumulator A and B).
000011 zcr 1 cccc ddddddddd 000001110   |   GETACCA      |   D          | Store the bottom 32 Bits of the A accumulator in register
                                                                        | “D (0-511)”. If executed again (no instruction in between
                                                                        | previous execution) store the top 32 Bits of the
                                                                        | A accumulator in register “D (0-511)”.
000011 zcr 1 cccc ddddddddd 000001111   |   GETACCB      |   D          | Store the bottom 32 Bits of the A accumulator in register
                                                                        | “D (0-511)”. If executed again (no instruction in between
                                                                        | previous execution) store the top 32 Bits of the
                                                                        | B accumulator in register “D (0-511)”.
000100 000 i cccc ddddddddd sssssssss   |   SETACCA      |   D,S      | Sets the high and low values of the 64 bit accumulator A.
                                                                        | The value contained in register “D (0-511)” sets the low long
                                                                        | while the value contained in “S (0-512)” sets the high long.
000100 010 i cccc ddddddddd sssssssss   |   SETACCB      |   D,S      | Sets the high and low values of the 64 bit accumulator B.
                                                                        | The value contained in register “D (0-511)” sets the low long
```

```
                                                               | while the value contained in "S (0-512)" sets the high long.
000011 zcr 1 cccc 000000101 000001000   |   FITACCA     |          | Shifts accumulator A's high long right into the low long so that
                                                               | the high long is MSB justified (discarding the low bits).
                                                               | Accumulator A's high long is then replaced with the number of
                                                               | bit places required to MSB justify Accumulator A's original
value.

000011 zcr 1 cccc 000000110 000001000   |   FITACCB     |          | Shifts accumulator B's high long right into the low long so that
                                                               | the high long is MSB justified (discarding the low bits).
                                                               | Accumulator B's high long is then replaced with the number of
                                                               | bit places required to MSB justify Accumulator B's original
value.
000011 zcr 1 cccc 000000111 000001000   |   FITACCS     |          | Similar operation to FITACCA/FITACCB. Examines both accumulator
                                                               |  A and B and right shifts both accumulators so that the greater
                                                               |  value of the two accumulators is MSB justified. The number of
                                                               | bits shifted is written to both accumulator's high long.
                                                               | This has the effect of scaling both accumulators equally.

Miscellaneous Instructions:

Each cog additionally features a number of new instructions to make many common operations much easier to perform than before.
 Most of the new instructions are in the extended instruction set while a few of the new instruction are in the original set.


Table 10: 'Extended Miscellaneous Instructions

Machine Code                            |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcr 1 cccc ddddddddd 000100000   |   DECOD5      |   D       | Overwrite register "D (0-511)" with decoded D[4:0] repeated 1
time.
                                                               | (e.g. $00000001 << D[4:0])
                                                               | DECOD5 decodes the 5 LSB's.
000011 zcr 1 cccc ddddddddd 000100001   |   DECOD4      |   D       | Overwrite register "D (0-511)" with decoded D[3:0] repeated 2
times.
                                                               | (e.g. $00010001 << D[3:0])
                                                               | DECOD4 decodes the 4 LSB's, replicating the result twice to
fill 32 bits.
000011 zcr 1 cccc ddddddddd 000100010   |   DECOD3      |   D       | Overwrite register "D (0-511)" with decoded D[2:0] repeated 4
times.
```

```
                                                                      | (e.g. $01010101 << D[2:0])
                                                                      | DECOD3 decodes the 3 LSB's, replicating the result four times
to fill 32 bits.
000011 zcr 1 cccc ddddddddd 000100011   |   DECOD2      |   D         | Overwrite register “D (0-511)” with decoded D[1:0] repeated 8
times.
                                                                      | (e.g. $11111111 << D[1:0])
                                                                      | DECOD2 decodes the 2 LSB's, replicating the result eight times
to fill 32 bits.
000011 zcr 1 cccc ddddddddd 000100100   |   BLMASK      |   D         | Overwrite register “D (0-511)” with a bit length mask specified
by D[5:0].
000011 zcr 1 cccc ddddddddd 000100101   |   NOT         |   D         | Overwrite register “D (0-511)” with the bitwise inverted
register “D (0-511)”.
000011 zcr 1 cccc ddddddddd 000100110   |   ONECNT      |   D         | Overwrite register “D (0-511)” with the count of ones in
register D
000011 zcr 1 cccc ddddddddd 000100111   |   ZERCNT      |   D         | Overwrite register “D (0-511)” with the count of zeros in
register D.
000011 zcr 1 cccc ddddddddd 000101000   |   INCPAT      |   D         | Overwrite register “D (0-511)” with the next bit pattern that
keeps
                                                                      | the number of ones and zeros the same in register D.
000011 zcr 1 cccc ddddddddd 000101001   |   DECPAT      |   D         | Overwrite register “D (0-511)” with the previous bit pattern
that keeps
                                                                      | the number of ones and zeros the same in register D.
000011 zcr 1 cccc ddddddddd 000101010   |   BINGRY      |   D         | Overwrite the binary pattern in register “D (0-511)” with its
gray code pattern.
000011 zcr 1 cccc ddddddddd 000101011   |   GRYBIN      |   D         | Overwrite the grey code pattern in register “D (0-511)” with its
 binary pattern.
000011 zcr 1 cccc ddddddddd 000101100   |   MERGEW      |   D         | Merge the high word and the low word of register “D (0-511)”
into each other and
                                                                      | overwrite register D with the new value. Bits of the low word
occupy bit spaces
                                                                      | 0, 2, 4, etc. Bits of the high word occupy bit spaces 1, 3, 5,
etc. (Interleave)
000011 zcr 1 cccc ddddddddd 000101101   |   SPLITW      |   D         | Split the bits of register “D (0-511)” into a high word and low
word and overwrite
                                                                      | register D with the new value. Bits of the low word come from
bit spaces 0, 2, 4,
                                                                      | etc. Bits of the high word come from bit spaces 1, 3, 5, etc. (
De-interleave)
000011 zcr 1 cccc ddddddddd 000101110   |   SEUSSF      |   D         | Overwrite register “D (0-511)” with a pseudo random bit pattern
```

```
seeded from the
                                                                       | value in register D. After 32 forward iterations, the original
                                                                       bit pattern is returned.
000011 zcr 1 cccc ddddddddd 000101111    |   SEUSSR      |   D         | Overwrite register "D (0-511)" with a pseudo random bit pattern
seeded from the
                                                                       | value in register D. After 32 reversed iterations, the original
                                                                       bit pattern is returned.
000011 zcr 1 cccc ddddddddd 1000bbbbb    |   ISOB        |   D.b       | Isolate bit "b (0-31)" of register "D (0-511)."
000011 zcr 1 cccc ddddddddd 1001bbbbb    |   NOTB        |   D.b       | Invert bit "b (0-31)" of register "D (0-511)."
000011 zcr 1 cccc ddddddddd 1010bbbbb    |   CLRB        |   D.b       | Clear bit "b (0-31)" of register "D (0-511)."
000011 zcr 1 cccc ddddddddd 1011bbbbb    |   SETB        |   D.b       | Set bit "b (0-31)" of register "D (0-511)."
000011 zcr 1 cccc ddddddddd 1100bbbbb    |   SETBC       |   D.b       | Set bit "b (0-31)" of register "D (0-511) to C."
000011 zcr 1 cccc ddddddddd 1101bbbbb    |   SETBNC      |   D.b       | Set bit "b (0-31)" of register "D (0-511) to NC."
000011 zcr 1 cccc ddddddddd 1110bbbbb    |   SETBZ       |   D.b       | Set bit "b (0-31)" of register "D (0-511) to Z."
000011 zcr 1 cccc ddddddddd 1111bbbbb    |   SETBNZ      |   D.b       | Set bit "b (0-31)" of register "D (0-511) to NZ."


Table 11: 'Extended Miscellaneous Flag Manipulation Instructions

Machine Code                             |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcr 1 cccc ddddddddd 000001010    |   PUSHZC      |   D         | Push the Z and C flags into D[1:0] and pop D[31:30]
                                                                       | into Z and C through WZ and WC.
000011 zcr 1 cccc ddddddddd 000001011    |   POPZC       |   D         | Pop D[1:0] into the Z and C flags and push D[31:30]
                                                                       | into Z and C through WZ and WC.
000011 zcn 1 cccc nnnnnnnnn 010100001    |   SETZC       |   D/#n      | Set the Z and C flags with D[1:0] through WZ and WC effects.


Table 12: 'Extended Miscellaneous Flow Control Instructions

Machine Code                             |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcn 1 cccc nnnnnnnnn 0100iiiii    |   REPD        |   D/#n ,i | Delayed repeat of the following "i (0-31)" instructions the
                                                                       | value in register "D(0-511)" or "n(0-511)" times.
                                                                       | The pipeline causes a delay of three instructions before
                              ????                                     | the repeated set of instructions begins to execute
000011 zcn 1 cccc nnnnnnnnn 0100iiiii    |   REPS        |   D/#n ,i | Repeat of the following "i (0-31)" instructions the
                                                                       | value in register "D(0-511)" or "n(0-511)" times.
000011 zcn 1 cccc nnnnnnnnn 010100000    |   NOPX        |   D/#n      | Repeat the NOP instruction the value in register "D(0-511)"
```

```
                                                                    | or "n(0-511)" times.
000011 zcn 1 cccc dddnnnnnn 011101011    |   SETSKIP      |   D/#n      | Executes up to the next 32 instructions as NOPs described by the
                                                                    | set bit pattern of a register "D(0-511)" or literal "N(0-63)".
```

Code:
Fast loading from HUB to COG ram can be done with just a few instructions:

```
' Load 64 longs from hub memory (@PTRA) into $100

    REPS    #64,#1
    SETINDA #$100
    RDLONGC INDA++,PTRA++
```

This way, you can load as much or as little as you please, to wherever in the COG you´d like.
Then, you can jump to it.

Table 13: Miscellaneous Instructions

Machine Code	Mnemonic	Operand	Operation
	NOP		
	RET		
	CALL		
	JMP		
000110 zcr i cccc ddddddddd sssssssss	ENC	D,S	Store encoded S in D.
000111 zcr i cccc ddddddddd sssssssss	JMPRET	D,S	Jump to address with intention to "return" to another address.
001000 zcr i cccc ddddddddd sssssssss	ROR	D,S	Rotate value right by specified number of bits.
001001 zcr i cccc ddddddddd sssssssss	ROL	D,S	Rotate value left by specified number of bits.
001010 zcr i cccc ddddddddd sssssssss	SHR	D,S	Shift value right by specified number of bits.
001011 zcr i cccc ddddddddd sssssssss	SHL	D,S	Shift value left by specified number of bits.
001100 zcr i cccc ddddddddd sssssssss	RCR	D,S	Rotate C right into value by specified number of bits.
001101 zcr i cccc ddddddddd sssssssss	RCL	D,S	Rotate C left into value by specified number of bits.
001110 zcr i cccc ddddddddd sssssssss	SAR	D,S	Shift value arithmetically right by specified number of bits.
001111 zcr i cccc ddddddddd sssssssss	REV	D,S	Reverse LSBs of value and zero-extend.
010000 zcr i cccc ddddddddd sssssssss	MINS	D,S	Limit minimum of signed value to another signed value.
010001 zcr i cccc ddddddddd sssssssss	MAXS	D,S	Limit maximum of signed value to another signed value.
010010 zcr i cccc ddddddddd sssssssss	MIN	D,S	Limit minimum of unsigned value to another unsigned value.

```
010011 zcr i cccc ddddddddd sssssssss   |   MAX       |   D,S     | Limit maximum of unsigned value to another unsigned value.
010100 zcr i cccc ddddddddd sssssssss   |   MOVS      |   D,S     | Set register's source field to a value.
010101 zcr i cccc ddddddddd sssssssss   |   MOVD      |   D,S     | Set register's destination field to a value.
010110 zcr i cccc ddddddddd sssssssss   |   MOVI      |   D,S     | Set register's instruction field to a value.
010111 zcr i cccc ddddddddd sssssssss   |   JMPRETD   |   D,S     | Jump to address with intention to "return" to another address.
                                                                  | Do not flush pipeline before jump - must be executed
                                                                  | two instructions before intended jump space
011000 zcr i cccc ddddddddd sssssssss   |   AND       |   D,S     | Bitwise AND values.
011001 zcr i cccc ddddddddd sssssssss   |   ANDN      |   D,S     | Bitwise AND value with NOT of another.
011010 zcr i cccc ddddddddd sssssssss   |   OR        |   D,S     | Bitwise OR values.
011011 zcr i cccc ddddddddd sssssssss   |   XOR       |   D,S     | Bitwise XOR values.
011100 zcr i cccc ddddddddd sssssssss   |   MUXC      |   D,S     | Set discrete bits of value to state of C.
011101 zcr i cccc ddddddddd sssssssss   |   MUXNC     |   D,S     | Set discrete bits of value to state of !C.
011110 zcr i cccc ddddddddd sssssssss   |   MUXZ      |   D,S     | Set discrete bits of value to state of Z.
011111 zcr i cccc ddddddddd sssssssss   |   MUXNZ     |   D,S     | Set discrete bits of value to state of !Z.
100000 zcr i cccc ddddddddd sssssssss   |   ADD       |   D,S     | Add unsigned values.
100001 zcr i cccc ddddddddd sssssssss   |   SUB       |   D,S     | Subtract unsigned values.
100010 zcr i cccc ddddddddd sssssssss   |   ADDABS    |   D,S     | Add absolute value to another value.
100011 zcr i cccc ddddddddd sssssssss   |   SUBABS    |   D,S     | Subtract absolute value from another value.
100100 zcr i cccc ddddddddd sssssssss   |   SUMC      |   D,S     | Sum signed value with another whose sign is inverted based on C.
100101 zcr i cccc ddddddddd sssssssss   |   SUMNC     |   D,S     | Sum signed value with another whose sign is inverted based on !C.
100110 zcr i cccc ddddddddd sssssssss   |   SUMZ      |   D,S     | Sum signed value with another whose sign is inverted based on Z.
100111 zcr i cccc ddddddddd sssssssss   |   SUMNZ     |   D,S     | Sum signed value with another whose sign is inverted based on !Z.
101000 zcr i cccc ddddddddd sssssssss   |   MOV       |   D,S     | Set register to a value.
101001 zcr i cccc ddddddddd sssssssss   |   NEG       |   D,S     | Get negative of a number.
101010 zcr i cccc ddddddddd sssssssss   |   ABS       |   D,S     | Get absolute value of a number
101011 zcr i cccc ddddddddd sssssssss   |   ABSNEG    |   D,S     | Get the negative of a number's absolute value.
101100 zcr i cccc ddddddddd sssssssss   |   NEGC      |   D,S     | Get value, or its additive inverse, based on C.
101101 zcr i cccc ddddddddd sssssssss   |   NEGNC     |   D,S     | Get value, or its additive inverse, based on !C.
101110 zcr i cccc ddddddddd sssssssss   |   NEGZ      |   D,S     | Get value, or its additive inverse, based on Z.
101111 zcr i cccc ddddddddd sssssssss   |   NEGNZ     |   D,S     | Get value, or its additive inverse, based on !Z.
110000 zcr i cccc ddddddddd sssssssss   |   CMPS      |   D,S     | Compare signed values.
110001 zcr i cccc ddddddddd sssssssss   |   CMPSX     |   D,S     | Compare signed values plus C.
110010 zcr i cccc ddddddddd sssssssss   |   ADDX      |   D,S     | Add unsigned values plus C.
110011 zcr i cccc ddddddddd sssssssss   |   SUBX      |   D,S     | Subtract unsigned value plus C from another unsigned value.
110100 zcr i cccc ddddddddd sssssssss   |   ADDS      |   D,S     | Add signed values.
110101 zcr i cccc ddddddddd sssssssss   |   SUBS      |   D,S     | Subtract signed values
110110 zcr i cccc ddddddddd sssssssss   |   ADDSX     |   D,S     | Add signed values plus C.
110111 zcr i cccc ddddddddd sssssssss   |   SUBSX     |   D,S     | Subtract signed value plus C from another signed value.
```

```
111000 zcr i cccc ddddddddd sssssssss   |   SUBR      |   D,S     | Subtract D from S and store in D.
111001 zcr i cccc ddddddddd sssssssss   |   CMPSUB    |   D,S     | Compare unsigned values,
                                                                  | subtract second if it is lesser or equal.
111010 zcr i cccc ddddddddd sssssssss   |   INCMOD    |   D,S     | Increment D between 0 and S. Wraps around to 0 when above S.
111011 zcr i cccc ddddddddd sssssssss   |   DECMOD    |   D,S     | Decrement D between S and 0. Wraps around to S when below 0.
111100 00r i cccc ddddddddd sssssssss   |   IJZ       |   D,S     | Increment D and jump to S if D is zero.
111100 01r i cccc ddddddddd sssssssss   |   IJZD      |   D,S     | Increment D and jump to S if D is zero. Do not flush pipeline
                                                                  | before jump - must be executed two instructions before intended
                                                                  | jump space
111100 10r i cccc ddddddddd sssssssss   |   IJNZ      |   D,S     | Increment D and jump to S if D is not zero.
111100 11r i cccc ddddddddd sssssssss   |   IJNZD     |   D,S     | Increment D and jump to S if D is not zero. Do not flush
                                                                  | pipeline before jump - must be executed two instructions
                                                                  | before intended jump space.
111101 00r i cccc ddddddddd sssssssss   |   DJZ       |   D,S     | Decrement D and jump to S if D is zero.
111101 01r i cccc ddddddddd sssssssss   |   DJZD      |   D,S     | Decrement D and jump to S if D is zero. Do not flush pipeline
                                                                  | before jump - must be executed two instructions before intended
                                                                  | jump space.
111101 10r i cccc ddddddddd sssssssss   |   DJNZ      |   D,S     | Decrement D and jump to S if D is not zero.
111101 11r i cccc ddddddddd sssssssss   |   DJNZD     |   D,S     | Decrement D and jump to S if D is not zero. Do not flush
                                                                  | pipeline before jump - must be executed two instructions
                                                                  | before intended jump space.
111110 000 i cccc ddddddddd sssssssss   |   TJZ       |   D,S     | Test value and jump to address if zero.
111110 010 i cccc ddddddddd sssssssss   |   TJZD      |   D,S     | Test value and jump to address if zero.Do not flush pipeline
                                                                  | before jump - must be executed two instructions
                                                                  | before intended jump space.
111110 100 i cccc ddddddddd sssssssss   |   TJNZ      |   D,S     | Test value and jump to address if not zero.
111110 110 i cccc ddddddddd sssssssss   |   TJNZD     |   D,S     | Test value and jump to address if not zero. Do not flush
                                                                  | pipeline before jump - must be executed two instructions
                                                                  | before intended jump space.
111110 001 i cccc ddddddddd sssssssss   |   SETINDA   |   D,S     | Setup indirection register address A bottom range and top range
                                                                  | where D is the top of the range and S is the bottom range.
                                                                  | The indirection register will allow access to COG registers
                                                                  | in this range.
111110 011 i cccc ddddddddd sssssssss   |   SETINDB   |   D,S     | Setup indirection register address B bottom range and top range
                                                                  | where D is the top of the range and S is the bottom range.
                                                                  | The indirection register will allow access to cog registers
                                                                  | in this range.
111110 111 i cccc ddddddddd sssssssss   |   WAITVID   |   D,S     | Wait to pass pixels to the video generator.
111111 0cr i cccc ddddddddd sssssssss   |   WAITCNT   |   D,S     | Wait for the CNT[31:0] register to equal D and then add S to D
```

```
                                                                | and store in D. If WC is specified then wait for CNT[63:32]
                                                                | to equal D .
111111 1c0 i cccc dddddddd sssssssss   |   WAITPEQ      |   D,S     | Pause execution until I/O pin(s) match designated state(s).
111111 1c1 i cccc dddddddd sssssssss   |   WAITPNE      |   D,S     | Pause execution until I/O pin(s) don't match designated state(s).


Table 15: 'Port Access Instructions

Machine Code                           |   Mnemonic     |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcn 1 cccc ddnndddd 011100100   |   SETPORA      |   D/#n    | Assign PORTA to physical I/O ports (0-2) or internal I/O port 3
                                                                    | given register "D (0-511)" or number "n (0-3)".
000011 zcn 1 cccc ddnndddd 011100101   |   SETPORB      |   D/#n    | Assign PORTB to physical I/O ports (0-2) or internal I/O port 3
                                                                    | given register "D (0-511)" or number "n (0-3)".
000011 zcn 1 cccc ddnndddd 011100110   |   SETPORC      |   D/#n    | Assign PORTC to physical I/O ports (0-2) or internal I/O port 3
                                                                    | given register "D (0-511)" or number "n (0-3)".
000011 zcn 1 cccc ddnndddd 011100111   |   SETPORD      |   D/#n    | Assign PORTD to physical I/O ports (0-2) or internal I/O port 3
                                                                    | given register "D (0-511)" or number "n (0-3)".


Table 16: 'Pin State Access Instructions

Machine Code                           |   Mnemonic     |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcn 1 cccc ddnnnnnnn 011010110  |   GETP         |   D/#n    | Get pin number given by register "D (0-511)"
                                                                    | or "n (0-127)"into !Z or C flags.
000011 zcn 1 cccc ddnnnnnnn 011010111  |   GETPN        |   D/#n    | Get pin number given by register "D (0-511)"
                                                                    | or "n (0-127)"into Z or !C flags.
000011 zcn 1 cccc ddnnnnnnn 011011000  |   OFFP         |   D/#n    | Toggle pin number given by register "D (0-511)"
                                                                    | or "n (0-127)" off or on. DIR
000011 zcn 1 cccc ddnnnnnnn 011011001  |   NOTP         |   D/#n    | Invert pin number given by the value in register "D (0-511)"
                                                                    | or "n (0-127)". OUT
000011 zcn 1 cccc ddnnnnnnn 011011010  |   CLRP         |   D/#n    | Clear pin number given by the value in register "D (0-511)"
                                                                    | or "n (0-127)". OUT
000011 zcn 1 cccc ddnnnnnnn 011011011  |   SETP         |   D/#n    | Set pin number given by the value in register "D (0-511)"
                                                                    | or "n (0-127)". OUT
000011 zcn 1 cccc ddnnnnnnn 011011100  |   SETPC        |   D/#n    | Set pin number given by the value in register "D (0-511)"
                                                                    | or "n (0-127)" to C.
000011 zcn 1 cccc ddnnnnnnn 011011101  |   SETPNC       |   D/#n    | Set pin number given by the value in register "D (0-511)"
```

```
                                                                   | or “n (0-127)” to !C
000011 zcn 1 cccc ddnnnnnnn 011011110   |   SETPZ       |   D/#n    | Set pin number given by the value in register “D (0-511)”
                                                                   | or “n (0-127)” to !Z.
000011 zcn 1 cccc ddnnnnnnn 011011111   |   SETPNZ      |   D/#n    | Set pin number given by the value in register “D (0-511)”
                                                                   | or “n (0-127)” Z.
```

External RAM
Each cog now features the ability, with the help of the I/O pins, to quickly stream parallel data in or out of the
I/O pins aligned to a clock source. Data is streamed to/from the CLUT or WRQUAD overlay.
From there it can be quickly feed to the video generator or to the internal HUB RAM.
XFR feeds data 16 Bits or 32 Bits at a time at the system clock speed.

Table 17: 'External RAM Instruction

```
Machine Code                            |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zcn 1 cccc dddnnnnnn 011101001   |   SETXFR      |   D/#n    | Setup the direction of the data stream, the source and
                                                                   | destination of the data stream, and the size of the data
                                                                   | stream given D or “n (0-63)”.
```

Chip-To-Chip Communication

Each cog now also features high-speed serial transfer and receive hardware for chip-to-chip communication.
The hardware requires three I/O pins (SO, SI, CLK).

Table 18: 'Chip-To-Chip Communication Instructions

```
Machine Code                            |   Mnemonic    |   Operand |   Operation
-----------------------------------------------------------------------------------------------
000011 zc0 1 cccc ddddddddd 000001001   |   SNDSER      |   D       | Sends a long (D) out of the special chip-to-chip serial port.
                                                                   | Blocks until the long is sent. Use C flag to avoid blocking.
000011 zc1 1 cccc ddddddddd 000001001   |   RCVSER      |   D       | Receives a long (D) in from the special chip-to-chip serial port.
                                                                   | Blocks until the long is received. Use C flag to avoid blocking.
000011 zcn 1 cccc ddddddddd 011101010   |   SETSER      |   D/#n    | Sets up the serial port I/O pins to use for SO, SI,
                                                                   | and CLK given D or “n (0-63)”.
```

Cog Memory 'Remapping
Cogs now have the ability to remap their internal memory to help facilitate context switching between register banks.
Instead of having to save a bunch of internal register to switch running programs all references to a set of register
can be changed instantaneously.

Table 19: Cog Memory 'Remapping Instruction

Machine Code	Mnemonic	Operand	Operation
000011 zcn 1 cccc dddnnnnnn 011100001	SETMAP	D/#n	Remap one cog register space to another COG register space given D or n.

Cog-To-Cog 'Communication
Cogs now have the ability to communicate directly to each other using the internal I/O Port D,
which connects each cog to every other cog.

Table 20: Cog-To-Cog 'Communication Instruction

Machine Code	Mnemonic	Operand	Operation
000011 zcn 1 cccc nnnnnnnnn 011101000	SETXCH	D/#n	Reconfigure Port D I/O masks given D or n to select which cogs to listen to.

'Pin Modes
Each I/O pin is now capable of setting itself into many different modes to more easily interface with the analog world.
By default, each I/O starts up in the basic robust digital I/O state. However, once configured the
I/O pin can be used for external RAM memory transfer, as an ADC, as a DAC, a Schmitt trigger, or a comparator, etc.
See Figure 2 for a table of pin modes and their associated properties.

Table 21: 'Pin Mode Access Instructions

Machine Code	Mnemonic	Operand	Operation
000011 zcn 1 cccc ddnnddddd 011100011	SETPORT	D/#n	Assign which port the CFGPINS instruction will configure given register “D (0-511)” or number “n (0-3)”.
111110 101 i cccc ddddddddd sssssssss	CFGPINS	D/#n	Setup pins masked by register “D (0-511)” to register “S (0-511)”. The pin configuration modes are below.

NOTE: PinA is the pin being set. PinB is its neighbor (All I/O pins have a cross coupled neighbor).

Input is the Boolean statement for what the pin returns when read. Output is the statement for what the pins outputs when it is an output (Some modes output their input to make feedback relaxation oscillators, etc). Each pin's high and low drivers can be configured to work in many different modes. Pins can also re-clock data sent to them locally to remove jitter in data. Every pin is setup by a 13-bit configuration value.

Figure 2: 'Pin Modes

```
Code            | Mode                      | Input             | PinA Out  |   PinB    | Compare
-------------------------------------------------------------------------------------------------
0000 CIOHHHLLL  | General I/O               | PinA Logic        | OUT       | -         | -      |
0001 CIOHHHLLL  |                   DIR=0   | PinA Logic        | Input     | -         | -      |
0010 CIOHHHLLL  |                   Float   | PinA Logic        | Input     | -         | -      |
0011 CIOHHHLLL  | C|OUT/IN          DIR=1   | PinA Logic        | Input     | 1M PinA   | -      |
0100 CIOHHHLLL  | 0|Live        HHH         | PinA Schmitt      | OUT       | -         | -      |
0101 CIOHHHLLL  | 1|Clocked     LLL|Drive   | PinA Schmitt      | Input     | -         | -      |
0110 CIOHHHLLL  |               000|Fast    | PinA Schmitt      | Input     | -         | -      |
0111 CIOHHHLLL  | I|IN          001|Slow    | PinA Schmitt      | Input     | 1M PinA   | -      |
1000 CIOHHHLLL  | 0|True        010|1500R   | PinA > VIO/2      | OUT       | -         | Fast   |
1001 CIOHHHLLL  | 1|Inverted    011|10k     | PinA > VIO/2      | Input     | -         | Fast   |
1010 CIOHHHLLL  |               100|100k    | PinA > VIO/2      | Input     | -         | Fast   |
1011 CIOHHHLLL  | O|Output      101|100uA   | PinA > VIO/2      | Input     | 1M PinA   | Fast   |
1100 CIOHHHLLL  | 0|True        110|10uA    | PinA > PinB       | OUT       | -         | Precise|
1101 CIOHHHLLL  | 1|Invert      111|Float   | PinA > PinB       | Input     | -         | Precise|
1110 CIOHHHLLL  |                           | PinA > PinB       | Input     | 1M PinA   | Precise|
----------------------------------------------------------------------------------------------------
1111 0LLLLLLLL  | Compare Level             | PinA > VIO/256*L  | -         | -         | Precise|
1111 1000xxxxx  | ADC Diff, 100k            | PinA > VIO/2 10k  | 10k VIO/2 | 10k VIO/2 | Fast   |
1111 10010xxxx  | ADC Precise, DIR/OUT=Cal  | ADC               | -         | -         | Fast   |
1111 10011xxxx  | ADC Fast,     DIR/OUT=Cal | ADC               | -         | -         | Fast   |
1111 101VxxCCC  | DAC 75R,  V=Video, C,COG  | 1                 | -         | -         | -      |
1111 110HHHLLL  | SDRAM Data I/O            | PinA Logic        | -         | -         | -      |
1111 111HHHLLL  | SDRAM Clock Out           | 1                 | -         | -         | -      |
```

_-

Video Generator

Each cog has a video generator capable of generating composite, component, s-video, and VGA video. The video generator is fed pixel data through the waitvid instruction and uses the pixel data to look up colors to output from the CLUT. The video generator understands R.G.B.A.X color grouping and can handle RGB565/555/444/etc formatted data.

Table 22: 'Video Generator Access Instructions

Machine Code	Mnemonic	Operand	Operation
000011 zcn 1 cccc nnnnnnnnn 011101100	SETVID	D/#n	Setup the video generator according to D or n to output video from the CLUT.
000011 zcn 1 cccc nnnnnnnnn 011101101	SETVIDY	D/#n	Setup the video generator color matrix transform term Y according to D or n.
000011 zcn 1 cccc ?nnnnnnnn 011101110	SETVIDI	D/#n	Setup the video generator color matrix transform term I according to D or n.
000011 zcn 1 cccc nnnnnnnnn 011101111	SETVIDQ	D/#n	Setup the video generator color matrix transform term Q according to D or n.

DAC Hardware

Each cog has four DACs capable of SIN/COS wave output, saw tooth wave output, triangle wave output, and square wave output. Additionally, the video generator, when operational, will use the four DACs to produce video output. Please refer to the information below.

? CFGDAC – 00 = 9-bit level with 9-bit dither.
? CFGDAC – 01 = 9-bit level from counter with 9-bit dither from counter.
  o DAC0 = CTRASIN, DAC1 = CTRACOS, DAC2 = CTRBSIN, DAC3 = CTRBCOS
? CFGDAC – 10 = 9-bit level from counter with 9-bit dither from counter.
  o DAC0/2 = CTRASIN + CTRBSIN, DAC1.3 = CTRACOS + CTRBCOS
? CFGDAC – 11 = Video generator controlled.
  o DAC0 = SYNC, DAC1 = Q/B, DAC2 = I/G, DAC3 = Y/R

Table 23: DAC Hardware Access Instructions

Machine Code	Mnemonic	Operand	Operation
000011 zcn 1 cccc dddddddnn 011001100	CFGDAC0	D/#n	Configure DAC0 to D or n. See above.
000011 zcn 1 cccc dddddddnn 011001101	CFGDAC1	D/#n	Configure DAC1 to D or n. See above.
000011 zcn 1 cccc dddddddnn 011001110	CFGDAC2	D/#n	Configure DAC2 to D or n. See above.

000011 zcn 1 cccc ddddddddnn 011001111	CFGDAC3	D/#n	Configure DAC3 to D or n. See above.
000011 zcn 1 cccc nnnnnnnnn 011010000	SETDAC0	D/#n	Set DAC0 to top 18 bits of D/n.
000011 zcn 1 cccc nnnnnnnnn 011010001	SETDAC1	D/#n	Set DAC1 to top 18 bits of D/n.
000011 zcn 1 cccc nnnnnnnnn 011010010	SETDAC2	D/#n	Set DAC2 to top 18 bits of D/n.
000011 zcn 1 cccc nnnnnnnnn 011010011	SETDAC3	D/#n	Set DAC3 to top 18 bits of D/n.
000011 zcn 1 cccc dnnnnnnnn 011010100	CFGDACS	D/#n	Configure DACs to D or n. See above.
000011 zcn 1 cccc nnnnnnnnn 011010101	SETDACS	D/#n	Set DACs to top 18 bits of D/n.

'Texture Mapping

Each cog has texture mapping hardware to assist the video generator with displaying textures and performing color blending on screen.

Table 24: 'Texture Mapping Instructions

Machine Code	Mnemonic	Operand	Operation
000011 zcr 1 cccc ddddddddd 000010100	GETPIX	D	Store texture pointer address in D.
000011 zcn 1 cccc nnnnnnnnn 010111000	SETPIX	D/#n	Set texture size and address to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111001	SETPIXU	D/#n	Set texture pointer x address to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111010	SETPIXV	D/#n	Set texture pointer y address to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111011	SETPIXZ	D/#n	Set texture pointer z address to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111101	SETPIXR	D/#n	Set texture pointer R blending to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111110	SETPIXG	D/#n	Set texture pointer G blending to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111111	SETPIXB	D/#n	Set texture pointer B blending to D/n.
000011 zcn 1 cccc nnnnnnnnn 010111100	SETPIXA	D/#n	Set texture pointer A blending to D/n.

'Counter Modules

Each cog has two counter modules - CTRA and CTRB. Each counter module has a FRQ, PHS, SIN, and COS register. The counter modules control the SIN and COS registers to track the phase and power of a signal. The FRQ and PHS registers work the same. Each counter module also has logic modes, which allow it to accumulate given different logic equations involving a selected pin A and pin B - see P8X32A. The counter modes now also feature quadrature encoder accumulation and automatic PWM generation.

Table 25: 'Counter Hardware Access Instructions

Machine Code	Mnemonic	Operand	Operation

```
000011 zcr 1 cccc dddddddd 000111000    |    GETPHSA     |    D         | Store PHSA in D.
000011 zcr 1 cccc dddddddd 000111001    |    GETPHZA     |    D         | Store PHSA in D and zero PHSA.
000011 zcr 1 cccc dddddddd 000111010    |    GETCOSA     |    D         | Store COSA in D.
000011 zcr 1 cccc dddddddd 000111011    |    GETSINA     |    D         | Store SINA in D.
000011 zcr 1 cccc dddddddd 000111100    |    GETPHSB     |    D         | Store PHSB in D.
000011 zcr 1 cccc dddddddd 000111101    |    GETPHZB     |    D         | Store PHSB in D and zero PHSB
000011 zcr 1 cccc dddddddd 000111110    |    GETCOSB     |    D         | Store COSB in D.
000011 zcr 1 cccc dddddddd 000111111    |    GETSINB     |    D         | Store SINB in D.
000011 zcn 1 cccc nnnnnnnnn 011110000   |    SETCTRA     |    D/#n      | Set CTRA mode to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110001   |    SETWAVA     |    D/#n      | Set CTRA wave mode to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110010   |    SETFRQA     |    D/#n      | Set FRQA to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110011   |    SETPHSA     |    D/#n      | Set PSHA to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110100   |    ADDPHSA     |    D/#n      | Add D/n to PSHA.
000011 zcn 1 cccc nnnnnnnnn 011110101   |    SUBPHSA     |    D/#n      | Subtract D/n from PSHA.
000011 zcn 1 cccc nnnnnnnnn 011110110   |    SYNCTRA     |              | Wait for PHSA to overflow.
000011 zcn 1 cccc nnnnnnnnn 011110111   |    CAPCTRA     |              | Remove current sum from PHSA.
000011 zcn 1 cccc nnnnnnnnn 011111000   |    SETCTRB     |    D/#n      | Set CTRB mode to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110001   |    SETWAVB     |    D/#n      | Set CTRB wave mode to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110010   |    SETFRQB     |    D/#n      | Set FRQB to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110011   |    SETPHSB     |    D/#n      | Set PSHB to D/n.
000011 zcn 1 cccc nnnnnnnnn 011110100   |    ADDPHSB     |    D/#n      | Add D/n to PSHB.
000011 zcn 1 cccc nnnnnnnnn 011110101   |    SUBPHSB     |    D/#n      | Subtract D/n from PSHB.
000011 zcn 1 cccc nnnnnnnnn 011111110   |    SYNCTRB     |              | Wait for PHSB to overflow.
000011 zcn 1 cccc nnnnnnnnn 011111111   |    CAPCTRB     |              | Remove current sum from PHSB.
```

```
            'Assembly Conditions
Condition  | Instruction Executes        | Condition      |  Instruction Executes
-------------------------------------------------------------------
IF_ALWAYS  | always                      | IF_NC_AND_Z    |  if C clear and Z set
IF_NEVER   | never                       | IF_NC_AND_NZ   |  if C clear and Z clear
IF_E       | if equal (Z)                | IF_C_OR_Z      |  if C set or Z set
IF_NE      | if not equal (!Z)           | IF_C_OR_NZ     |  if C set or Z clear
IF_A       | if above (!C & !Z)          | IF_NC_OR_Z     |  if C clear or Z set
IF_B       | if below (C)                | IF_NC_OR_NZ    |  if C clear or Z clear
IF_AE      | if above/equal (!C)         | IF_Z_EQ_C      |  if Z equal to C
IF_BE      | if below/equal (C | Z)      | IF_Z_NE_C      |  if Z not equal to C
IF_C       | if C set                    | IF_Z_AND_C     |  if Z set and C set
IF_NC      | if C clear                  | IF_Z_AND_NC    |  if Z set and C clear
```

```
IF_Z        | if Z set                 | IF_NZ_AND_C    |  if Z clear and C set
IF_NZ       | if Z clear               | IF_NZ_AND_NC   |  if Z clear and C clear
IF_C_EQ_Z   | if C equal to Z          | IF_Z_OR_C      |  if Z set or C set
IF_C_NE_Z   | if C not equal to Z      | IF_Z_OR_NC     |  if Z set or C clear
IF_C_AND_Z  | if C set and Z set       | IF_NZ_OR_C     |  if Z clear or C set
IF_C_AND_NZ| if C set and Z clear     | IF_NZ_OR_NC    |  if Z clear or C clear
```

Effects and Condition Codes
Every assembly instruction can conditionally update the Z and/or C flag with WC and WZ effects. Additionally, the result can conditionally be written using the NR and WR flags. In addition, instructions can be conditionally executed given the Z and/or C flag—see P8X32A.
