Parallax Propeller 2 Documentation

2018-09-01

v32

(not yet updated: Boot ROM, Debug)

Design Status

Date	Progress
2018_04_25	Final Verilog design files sent to On Semi (8 cogs, 512KB hub, 64 smart pins)
2018_05_29	Final ROM data sent to On Semi
2018_07_09	Final Sign-off with On Semi, reticles being made
2018_09_11	Wafers done! Only took 9 weeks, instead of 14.

2018_09_27	Received 10 glob-top prototype chips from On Semi. Chips are functional, but sign-extension problems in Verilog source files caused the following problems: 1) Cogs' IQ modulators' outputs are nonsensical. 2) Smart pin measurement modes which are supposed to count by +1/-1 are counting by +1/+3. 3) ALTx instructions aren't sign-extending S[17:09] before adding into D. These sign-extension problems have already been fixed in the Verilog source files and tested on the FPGA. There is also a low-glitch-on-high-to-float problem on some I/O pins due to crosstalk between DIR and OUT signals. This will be fixed by timing constraints and signal shielding. A respin of the silicon is planned after more testing.
2018_11_13	Received 135 Amkor-packaged prototype chips from On Semi. These chips will have better heat dissipation than the glob-top prototypes.

post-dft

OVERVIEW

The Propeller 2 is a microcontroller containing 1, 2, 4, 8, or 16 identical 32-bit processors called “cogs”, which connect to a common “hub”. The hub provides a shared RAM, a CORDIC math solver, and housekeeping facilities. The architecture supports up to 64 smart I/O pins, each capable of many autonomous analog and digital functions.

Each cog has:

- Access to all I/O pins, plus four fast DAC output channels
- 512 longs of dual-port register RAM for code and fast variables
- 512 longs of dual-port lookup RAM for code, streamer lookup, and variables
- Ability to execute code directly from register RAM, lookup RAM, and hub RAM
- ~350 unique instructions for math, logic, timing, and control operations
- 2-clock execution for all math and logic instructions, including 16 x 16 multiply
- 6-clock custom-bytecode executor for interpreted languages
- Ability to stream hub RAM and/or lookup RAM to DACs and pins, also pins to hub RAM
- Live colorspace conversion using a 3 x 3 matrix with 8-bit signed/unsigned coefficients
- Pixel blending instructions for 8:8:8:8 data
- 16 unique event trackers that can be polled and waited upon
- 3 prioritized interrupts that trigger on selectable events
- Hidden debug interrupt for single-stepping, breakpoint, and polling

The hub provides the cogs with:

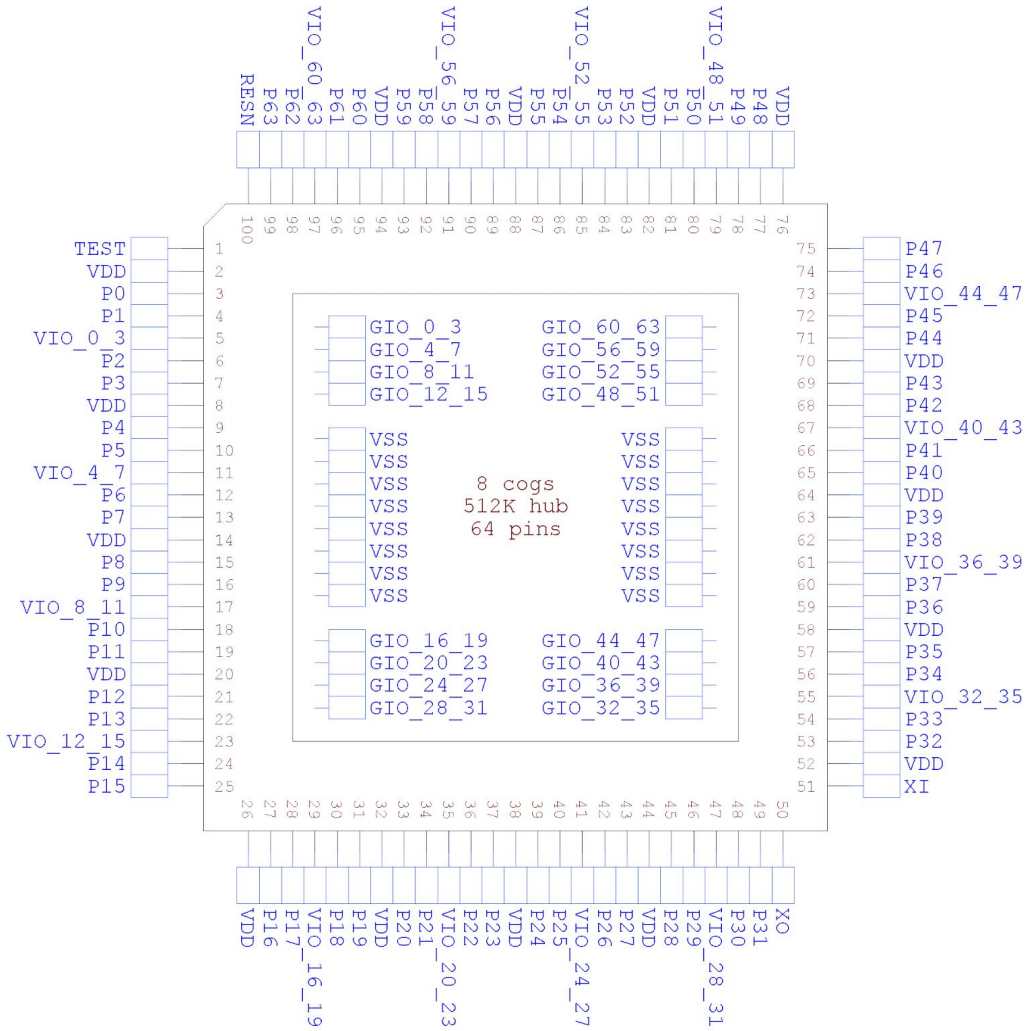
- Up to 1MB of contiguous RAM in a 20-bit address space
    - 32-bits-per-clock sequential read/write for all cogs, simultaneously
    - readable and writable as bytes, words, or longs
    - last 16KB of RAM also appears at end of 1MB map and is write-protectable
- 32-bit, pipelined CORDIC solver with scale-factor correction
    - 32-bit x 32-bit unsigned multiply with 64-bit result
    - 64-bit / 32-bit unsigned divide with 32-bit quotient and 32-bit remainder
    - 64-bit → 32-bit square root
    - Rotate (X32,Y32) by Theta32 → (X32,Y32)
    - (Rho32,Theta32) → (X32,Y32) polar-to-cartesian
    - (X32,Y32) → (Rho32,Theta32) cartesian-to-polar
    - 32 → 5.27 unsigned-to-logarithm
    - 5.27 → 32 logarithm-to-unsigned
    - Cogs can start CORDIC operations every 1/2/4/8/16 clocks and get results 55 clocks later
- 16 semaphore bits with atomic read-modify-write operations
- 32-bit free-running counter, increments every clock, cleared on reset
- High-quality PRNG (Xoroshiro128**), updates every clock, unique data to each cog and pin
- Mechanisms for starting, polling, and stopping cogs
- 16KB boot ROM
    - Loads into last 16KB of hub RAM on boot-up
    - SPI loader for automatic startup from 8-pin flash or SD card
    - Serial loader for startup from host

Each smart I/O pin has the following functions:

- 8-bit, 120-ohm (3ns) and 1k-ohm DACs with 16-bit oversampling, noise, and high/low digital modes
- Delta-sigma ADC with 5 ranges, 2 sources, and VIO/GIO calibration
- Logic, Schmitt, pin-to-pin-comparator, and 8-bit-level-comparator input modes
- 2/3/5/8-bit-unanimous input filtering with selectable sample rate
- Incorporation of inputs from relative pins, -3 to +3
- Negative or positive local feedback, with or without clocking
- Separate drive modes for high and low output: logic/1.5k/15k/150k/1mA/100uA/10uA/float
- Programmable 32-bit clock output, transition output, NCO/duty output
- Triangle/sawtooth/SMPS PWM output, 16-bit frame with 16-bit prescaler
- Quadrature decoding with 32-bit counter, both position and velocity modes
- 16 different 32-bit measurements involving one or two signals
- USB full-speed and low-speed (via odd/even pin pairs)
- Synchronous serial transmit and receive, 1 to 32 bits
- Asynchronous serial transmit and receive, 1 to 32 bits, up to clock/3

Six different clock modes, all under software control with glitch-free switching between sources:

- Internal 20MHz+ RC oscillator, nominally 24MHz, used as initial clock source
- Crystal oscillator with internal loading caps for 7.5pF/15pF crystals, can feed PLL
- Clock input, can feed PLL
- Fractional PLL with 1..64 crystal divider --> 1..1024 VCO multiplier --> optional (1..15)*2 VCO post-divider
- Internal ~20KHz RC oscillator for low-power operation (100uA)
- Clock can be stopped for lowest power until reset (34uA)



MEMORIES

Each cog has a 512 x 32-bit register RAM and a 512 x 32 lookup RAM. Program code can execute from both, but only the register RAM can be accessed as D and S registers. The lookup RAM must be read and written using RDLUT/WRLUT instructions.

The globally-accessible hub RAM can be read and written as bytes, words, and longs. Hub addresses are always byte-oriented. There are no special alignment rules for words and longs in hub RAM. Cogs can read and write bytes, words, and longs at any hub address, as well as execute instruction longs from any hub address starting at $400.

Cogs use 20-bit addresses for program counters (PC's) and hub pointers. This affords a data space of 1MB,

Depending on the state of a cog's PC, an instruction will be fetched from either its register RAM, its lookup RAM, or the hub RAM:

PC Address	Instruction Source	Word Size	PC Increment

$00000..$001FF	cog register RAM	32 bits (long)	1
$00200..$003FF	cog lookup RAM	32 bits (long)	1
$00400..$FFFFF	hub RAM	8 bits (byte)	4

For a cog to transition from internal register/lookup execution to hub execution, a branch to $00400+ must execute to establish the change. This will trigger the cog to begin reading and spooling hub RAM, so that a stream of instruction longs will be available for the cog. There is no special consideration when going from hub execution to register/lookup execution, as those memories are always accessible by the cog without any special setup sequence.

HUB RAM

On hub RAM implementations of less than the full 1MB, the last 16KB of hub RAM is normally addressable at both its normal address range, as well as at $FC000..$FFFFF. This provides a stable address space for the 16KB of internal ROM which gets cached into the last 16KB of hub RAM on startup. This upper 16KB mapping is also used by the cog debugging scheme.

The last 16KB of RAM can be disappeared from its normal address range and made read-only at $FC000..$FFFFF. This is useful for making the last 16KB of RAM persistent, like ROM. It is also how debugging is realized, as the RAM mapped to $FC000..$FFFFF can still be written to from within debug interrupt service routines, permitting the otherwise-protected RAM to be used as debugger-application space and cog-register swap buffers for debug interrupts.

See the HUBSET instruction definition for setting up write-protection.

Here are the hub memory maps for the various FPGA boards currently being supported during development. The "W" column represents write-protection status, set by HUBSET, for the last 16KB of hub RAM:

FPGA Board	Hub RAM Cogs	W	Lower RAM	Gap (reads $00)	Top 16KB RAM
DE0-Nano	32KB 1 cog	0 1	$00000..$07FFF $00000..$03FFF	$08000..$FBFFF $04000..$FBFFF	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read
BeMicro-A2	128KB 1 cog	0 1	$00000..$1FFFF $00000..$1BFFF	$20000..$FBFFF $1C000..$FBFFF	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read
DE2-115	256KB 4 cogs	0 1	$00000..$3FFFF $00000..$3BFFF	$40000..$FBFFF $3C000..$FBFFF	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read
Prop123-A7	512KB 4 cogs	0 1	$00000..$7FFFF $00000..$7BFFF	$80000..$FBFFF $7C000..$FBFFF	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read
Prop123-A9 BeMicro-A9	512KB 8 cogs	0 1	$00000..$7FFFF $00000..$7BFFF	$80000..$FBFFF $7C000..$FBFFF	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read
Prop123-A9 BeMicro-A9	1024KB 16 cogs	0 1	$00000..$FFFFF	none, full map	$FC000..$FFFFF, R/W $FC000..$FFFFF, Read

THE “EGG BEATER” HUB RAM INTERFACE

Hub RAM is comprised of 32-bit-wide single-port RAMs with byte-level write controls. For each cog, there is one of these RAMs, but it is multiplexed among all cogs. Let’s call these separate RAMs “slices”. Each RAM slice holds every single/2nd/4th/8th/16th (depending on number of cogs) set of 4 bytes in the composite hub RAM. At every clock, each cog can

access the “next” RAM slice, allowing for continuously-ascending bidirectional streaming of 32 bits per clock between the composite hub RAM and each cog.

When a cog wants to read or write the hub RAM, it must wait up to #cogs-1 clocks to access the initial RAM slice of interest. Once that occurs, subsequent slices can be accessed on every clock, thereafter, for continuous reading or writing of 32-bit longs.

To smooth out data flow for less than 32-bits-per-clock between hub RAM and the cog, each cog has a hub FIFO interface which can be set for hub-RAM-read or hub-RAM-write operation. This FIFO interface allows hub RAM to be either sequentially read or sequentially written in any combination of bytes, words, or longs, at any rate, up to one long per clock. No matter the transfer frequency or the word size, the FIFO will ensure that the cog's reads or writes are all properly conducted from or to the composite hub RAM.

Cogs can access hub RAM either via the sequential FIFO interface, or by waiting for RAM slices of interest, while yielding to the FIFO. If the FIFO is not busy, which is soon the case if data is not being read from or written to it, random accesses will have full opportunity to access the composite hub RAM.

There are three ways the hub FIFO interface can be used, and it can only be used for one of these, at a time:

- Hub execution (when the PC is $00400..$FFFFF)
- Streamer usage (background transfers from hub RAM → pins/DACs, or from pins → hub RAM)
- Software usage (fast sequential-reading or sequential-writing instructions)

For hub execution, FIFO operation is established automatically upon a branch to $00400+. For as long as the PC remains at $00400+, the FIFO will be used to feed instructions to the cog and it cannot be used for anything else.

For streamer or software usage, FIFO operation must be established by a RDFAST or WRFAST instruction executed from cog register RAM ($00000..$001FF) or cog lookup RAM ($00200..$003FF). After that, and while remaining in cog register or cog lookup RAM, the streamer can be enabled to begin moving data in the background, or the two-clock RFxxxx/WFxxxx instructions can be used to manually read and write sequential data.

USING THE HUB RAM FIFO INTERFACE FOR FAST SEQUENTIAL ACCESS

To configure the hub FIFO interface for streamer or software usage, use the RDFAST and WRFAST instructions. These instructions establish read or write operation, the hub start address, and the block count. The block count determines how many 64-byte blocks will be read or written before wrapping to the original start address and reloading the original block count. If you intend to use wrapping, your hub start address must be long-aligned (address ends in %00), since there won't be an extra cycle in which to read/write a portion of a long in an extra hub RAM slice. In cases where you don’t want wrapping, just use 0 for the block count, so that wrapping won't occur until the entire 1MB hub map is sequenced through.

The FBLOCK instruction provides a way to set a new start address and a new 64-byte block count for when the current blocks are fully read or written and the FIFO interface would have otherwise wrapped back to the prior start address and reloaded the prior block count. FBLOCK can be executed after RDFAST, WRFAST, or a FIFO block wrap event. Coordinating FBLOCK instructions with streamer-FIFO activity enables dynamic and seamless streaming between hub RAM and pins/DACs.

Here are the RDFAST, WRFAST, and FBLOCK instructions:

```
EEEE 1100011 1LI DDDDDDDDD SSSSSSSSS          RDFAST  D/#,S/#
EEEE 1100100 0LI DDDDDDDDD SSSSSSSSS          WRFAST  D/#,S/#
EEEE 1100100 1LI DDDDDDDDD SSSSSSSSS          FBLOCK  D/#,S/#
```

For these instructions, the D/# operand provides the block count, while the S/# operand provides the hub RAM start address:

```
D/#     %xxxx_xxxx_xxxx_xxxx_xx00_0000_0000_0000 = block count for limited r/w
        %xxxx_xxxx_xxxx_xxxx_xxBB_BBBB_BBBB_BBBB = block count for wrapping

S/#     %xxxx_xxxx_xxxx_AAAA_AAAA_AAAA_AAAA_AAAA = start address for limited r/w
        %xxxx_xxxx_xxxx_AAAA_AAAA_AAAA_AAAA_AA00 = start address for wrapping
```

RDFAST and WRFAST each have two modes of operation.

If D[31] = 0, RDFAST/WRFAST will wait for any previous WRFAST to finish and then reconfigure the hub FIFO interface for reading or writing. In the case of RDFAST, it will additionally wait until the FIFO has begun receiving hub data, so that it can start being used in the next instruction.

If D[31] = 1, RDFAST/WRFAST will not wait for FIFO reconfiguration, taking only two clocks. In this case, your code must allow a sufficient number of clocks before any attempt is made to read or write FIFO data.

FBLOCK doesn't need to wait for anything, so it always takes two clocks.

Once RDFAST has been used to configure the hub FIFO interface for reading, you can enable the streamer for any hub-reading modes or use the following instructions to manually read sequential data from the hub:

```
EEEE 1101011 CZ0 DDDDDDDDD 000010000          RFBYTE  D          {WC/WZ/WCZ}
EEEE 1101011 CZ0 DDDDDDDDD 000010001          RFWORD  D          {WC/WZ/WCZ}
EEEE 1101011 CZ0 DDDDDDDDD 000010010          RFLONG  D          {WC/WZ/WCZ}
EEEE 1101011 CZ0 DDDDDDDDD 000010011          RFVAR   D          {WC/WZ/WCZ}
EEEE 1101011 CZ0 DDDDDDDDD 000010100          RFVARS  D          {WC/WZ/WCZ}
```

These instructions all take 2 clocks and read bytes, words, longs, and variable-length data from the hub into D, via the hub FIFO interface.

If WC is expressed, the MSB of the byte, word, long, or variable-length data will be written to C.

If WZ is expressed, Z will be set if the data read from the hub equaled zero, otherwise Z will be cleared.

RFVAR and RFVARS read 1..4 bytes of data, depending upon the MSB of the first byte, and then subsequent bytes, waiting in the FIFO. While RFVAR returns zero-extended data, RFVARS returns sign-extended data. This mechanism is intended to provide a fast and memory-efficient means for bytecode interpreters to read numerical constants and offset addresses that were assembled at compile-time for efficient reading during run-time.

This table shows the relationship between upcoming bytes in the FIFO and what RFVAR and RFVARS will return:

FIFO 1st Byte	FIFO 2nd Byte	FIFO 3rd Byte	FIFO 4th Byte	RFVAR Returns RFVARS Returns
%0SAAAAAA	-	-	-	%00000000_00000000_00000000_0SAAAAAA %SSSSSSSS_SSSSSSSS_SSSSSSSS_SSAAAAAA
%1AAAAAAA	%0SBBBBBB	-	-	%00000000_00000000_00SBBBBB_BAAAAAAA %SSSSSSSS_SSSSSSSS_SSSBBBBB_BAAAAAAA
%1AAAAAAA	%1BBBBBBB	%0SCCCCCC	-	%00000000_000SCCCC_CCBBBBBB_BAAAAAAA

				%SSSSSSSS_SSSSCCCC_CCBBBBBB_BAAAAAAA
%1AAAAAAA	%1BBBBBBB	%1CCCCCCC	%SDDDDDDD	%000SDDDD_DDDCCCCC_CCBBBBBB_BAAAAAAA %SSSSDDDD_DDDCCCCC_CCBBBBBB_BAAAAAAA

Once WRFAST has been used to configure the hub FIFO interface for writing, you can enable the streamer for any hub-writing modes or use the following instructions to manually write sequential data:

```
EEEE 1101011 00L DDDDDDDDD 000010101        WFBYTE  D/#
EEEE 1101011 00L DDDDDDDDD 000010110        WFWORD  D/#
EEEE 1101011 00L DDDDDDDDD 000010111        WFLONG  D/#
```

These instructions all take 2 clocks and write byte, word, or long data in D into the hub via the hub FIFO interface.

If a cog has been writing to the hub via WRFAST, and it wants to immediately COGSTOP itself, a 'WAITX #20' should be executed first, in order to allow time for any lingering FIFO data to be written to the hub.

RANDOMLY ACCESSING HUB RAM

Here are the random-access hub RAM read instructions:

```
EEEE 1010110 CZI DDDDDDDDD SSSSSSSSS        RDBYTE  D,S/#/PTRx  {WC/WZ/WCZ}
EEEE 1010111 CZI DDDDDDDDD SSSSSSSSS        RDWORD  D,S/#/PTRx  {WC/WZ/WCZ}
EEEE 1011000 CZI DDDDDDDDD SSSSSSSSS        RDLONG  D,S/#/PTRx  {WC/WZ/WCZ}
```

For these instructions, the D operand is the register which will receive the data read from the hub.

The S/#/PTRx operand supplies the hub address to read from.

If WC is expressed, the MSB of the byte, word, or long read from the hub will be written to C.

If WZ is expressed, Z will be set if the data read from the hub equaled zero, otherwise Z will be cleared.

Here are the random-access hub RAM write instructions:

```
EEEE 1100010 0LI DDDDDDDDD SSSSSSSSS        WRBYTE  D/#,S/#/PTRx
EEEE 1100010 1LI DDDDDDDDD SSSSSSSSS        WRWORD  D/#,S/#/PTRx
EEEE 1100011 0LI DDDDDDDDD SSSSSSSSS        WRLONG  D/#,S/#/PTRx
EEEE 1010011 11I DDDDDDDDD SSSSSSSSS        WMLONG  D,S/#/PTRx
```

For these instructions, the D/# operand supplies the data to be written to the hub.

The S/#/PTRx operand supplies the hub address to write to.

WMLONG writes longs, like WRLONG; however, it does not write any byte fields whose data are $00. This is intended for things like sprite overlays, where $00 byte data represent transparent pixels.

In the case of the ‘S/#/PTRx’ operand used by RDBYTE, RDWORD, RDLONG, WRBYTE, WRWORD, WRLONG, and WMLONG, there are five ways to express a hub address:

$000..$1FF - register whose 20 LSBs will be used as the hub address
#$00..$FF - 8-bit immediate hub address
##$00000..$FFFFF - 20-bit immediate hub address (invokes AUGS)
PTRx {[index5]} - PTR expression with optional modifier and 5-bit scaled index (#$100..$1FF)
PTRx {[##index20]} - PTR expression with a 20-bit unscaled index and optional modifier (invokes AUGS) (##$800000..$FFFFFF)

If AUGS is used to augment the #S value to 32 bits, the #S value will be interpreted differently:

#%0AAAAAAAA - No AUGS, 8-bit immediate address
#%1SUPNNNNN - No AUGS, PTRx expression with 5-bit scaled index
##%000000000000AAAAAAAAAAA_AAAAAAAAA - AUGS, 20-bit immediate address
##%000000001SUPNNNNNNNNNN_NNNNNNNNN - AUGS, PTRx expression with 20-bit unscaled index

PTRx expressions without AUGS:

```
INDEX = -16..+15 for simple offsets, 0..15 for ++'s, or 0..16 for --'s
SCALE = 1 for RDBYTE/WRBYTE, 2 for RDWORD/WRWORD, 4 for RDLONG/WRLONG/WMLONG

S = 0 for PTRA, 1 for PTRB
U = 0 to keep PTRx same, 1 to update PTRx (PTRx += INDEX*SCALE)
P = 0 to use PTRx + INDEX*SCALE, 1 to use PTRx (post-modify)
NNNNN = INDEX
nnnnn = -INDEX

1SUPNNNNN     PTR expression
--------------------------------------------------------------------------------
100000000     PTRA              'use PTRA
110000000     PTRB              'use PTRB
101100001     PTRA++            'use PTRA,               PTRA += SCALE
111100001     PTRB++            'use PTRB,               PTRB += SCALE
101111111     PTRA--            'use PTRA,               PTRA -= SCALE
111111111     PTRB--            'use PTRB,               PTRB -= SCALE
101000001     ++PTRA            'use PTRA + SCALE,       PTRA += SCALE
111000001     ++PTRB            'use PTRB + SCALE,       PTRB += SCALE
101011111     --PTRA            'use PTRA - SCALE,       PTRA -= SCALE
111011111     --PTRB            'use PTRB - SCALE,       PTRB -= SCALE

1000NNNNN     PTRA[INDEX]       'use PTRA + INDEX*SCALE
1100NNNNN     PTRB[INDEX]       'use PTRB + INDEX*SCALE
1011NNNNN     PTRA++[INDEX]     'use PTRA,               PTRA += INDEX*SCALE
1111NNNNN     PTRB++[INDEX]     'use PTRB,               PTRB += INDEX*SCALE
1011nnnnn     PTRA--[INDEX]     'use PTRA,               PTRA -= INDEX*SCALE
1111nnnnn     PTRB--[INDEX]     'use PTRB,               PTRB -= INDEX*SCALE
1010NNNNN     ++PTRA[INDEX]     'use PTRA + INDEX*SCALE, PTRA += INDEX*SCALE
1110NNNNN     ++PTRB[INDEX]     'use PTRB + INDEX*SCALE, PTRB += INDEX*SCALE
1010nnnnn     --PTRA[INDEX]     'use PTRA - INDEX*SCALE, PTRA -= INDEX*SCALE
1110nnnnn     --PTRB[INDEX]     'use PTRB - INDEX*SCALE, PTRB -= INDEX*SCALE
```

Examples:

Read byte at PTRA into D

```
    1111 1010110 001 DDDDDDDDD 100000000     RDBYTE  D,PTRA
```

Write lower word in D to PTRB - 7*2

```
    1111 1100010 101 DDDDDDDDD 110011001     WRWORD  D,PTRB[-7]
```

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

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

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

```
    1111 1010111 001 DDDDDDDDD 101111111     RDWORD  D,PTRA--
```

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

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

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

```
    1111 1011000 001 DDDDDDDDD 111001010     RDLONG  D,++PTRB[10]
```

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

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

PTRx expressions with AUGS:

If "##" is used before the index value in a PTRx expression, the assembler will automatically insert an AUGS instruction and assemble the 20-bit index instruction pair:

```
    RDBYTE  D,++PTRB[##$12345]

...becomes...

    1111 1111000 000 000111000 010010001     AUGS    #$00E12345
    1111 1010110 001 DDDDDDDDD 101000101     RDBYTE  D,#$00E12345 & $1FF
```

FAST BLOCK MOVES

By preceding RDLONG with either SETQ or SETQ2, multiple hub RAM longs can be read into either cog register RAM or cog lookup RAM. This transfer happens at the rate of one long per clock, assuming the hub FIFO interface is not accessing the same hub RAM slice as RDLONG, on the same cycle. If WC/WZ/WCZ are used with RDLONG, the flags will be set according to the last long read in the sequence.

Use SETQ+RDLONG to read multiple hub longs into cog register RAM:

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

Use SETQ2+RDLONG to read multiple hub longs into cog lookup RAM:

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

Similarly, WRLONG and WMLONG can be preceded by either SETQ or SETQ2 to write either multiple register RAM longs or lookup RAM longs into hub RAM.

Use SETQ+WRLONG to write multiple register RAM longs into hub RAM:

```
SETQ    #x                      'x = number of longs, minus 1, to write
WRLONG  first_reg,S/#/PTRx      'write x+1 longs starting at first_reg
```

Use SETQ2+WRLONG to write multiple lookup RAM longs into hub RAM:

```
SETQ2   #x                      'x = number of longs, minus 1, to write
WRLONG  first_lut,S/#/PTRx      'write x+1 longs starting at first_lut
```

Note that the above two examples apply to WMLONG, as well.

Because these block moves yield to the hub FIFO interface, they can be used during hub execution.

Note that a PTRx expression will not be scaled by the block size in the RDLONG/WRLONG/WMLONG instruction that follows the SETQ/SETQ2 instruction, but will remain single-long scaled.

LOOKUP RAM SHARING BETWEEN PAIRED COGS

Adjacent cogs whose ID numbers differ by only the LSB (cogs 0 and 1, 2 and 3, 4 and 5, etc.) can each allow their lookup RAMs to be written by the other cog via its local lookup RAM writes. This allows adjacent cogs to share data very quickly through their lookup RAMs.

The 'SETLUTS D/#' instruction is used to enable the lookup RAM to receive writes from the adjacent cog:

```
SETLUTS #0                      'disallow writes from other cog (default)
SETLUTS #1                      'allow writes from other cog
```

Lookup RAM writes from the other cog are implemented on the 2nd port of the lookup RAM, which port is shared by the streamer in DDS/LUT modes. If an external write occurs on the same clock as a streamer read, the external write gets priority. It is not intended that external writes would be enabled at the same time the streamer is in DDS/LUT mode.

In order to find and start two adjacent cogs with which this write-sharing scheme can be used, the COGINIT instruction has a mechanism for finding an even/odd pair and then starting them both with the same parameters. It will be necessary for the program to differentiate between even and odd cogs and possibly restart one, or both, with the final, intended program. To have COGINIT find and start two adjacent cogs, use %x_1_xxx1 for the D/# operand.

To facilitate handshaking between cogs sharing lookup RAM, the SETSE1...4 instructions can be used to set up lookup RAM read and write events.

STARTING AND STOPPING COGS

Any cog can start or stop any other cog, or restart or stop itself. Each of the sixteen cogs has a unique four-bit ID which can be used to start or stop it. It's also possible to start free (stopped or never started) cogs, without needing to know their ID's. This way, entire applications can be written which simply start free cogs, as needed, and as those cogs retire by stopping themselves or getting stopped by others, they return to the pool of free cogs and become available, again, for restarting.

The COGINIT instruction is used to start cogs:

```
COGINIT D/#,S/# {WC}

D/# =  %0_x_xxxx     The target cog loads its own registers $000..$1F7 from the hub,
                     starting at address S/#, then begins execution at address $000.

       %1_x_xxxx     The target cog begins execution at address S/#.

       %x_0_CCCC     The target cog's ID is %CCCC.

       %x_1_xxx0     If a cog is free (stopped), then start it.
                     To know if this succeeded, D must be a register and WC must be
                     used. If successful, C will be cleared and D will be over-
                     written with the target cog's ID. Otherwise, C will be set and D will be
                     overwritten with $F.

       %x_1_xxx1     If an even/odd cog pair is free (stopped), then start them.
                     To know if this succeeded, D must be a register and WC must be
                     used. If successful, C will be cleared and D will be over-
                     written with the even/lower target cog's ID. Otherwise, C will be set
                     and D will be overwritten with $F.
```

In each case of COGINIT, the last SETQ value is written into the target cog's PTRA register. This is intended as a convenient means of pointing the cog program to some runtime data structure or passing it a 32-bit parameter. As well, the S/# value is written into the target cog's PTRB register, in case the cog program needs to know where it started from.

```
       COGINIT #1,#$100           'load and start cog 1 from $100

       COGINIT #%1_0_1010,PTRA    'start cog 10 at PTRA

       SETQ    ptra_val           'ptra_val will go into target cog's PTRA
       COGINIT #%0_1_0000,addr    'load and start a free cog at addr

       COGINIT #%1_1_0001,addr    'start a pair of free cogs (lookup RAM sharing)

       COGINIT id,addr WC         '(id=$30) start a free cog, C=0 and D=ID if okay

       COGID   myID               'reload and restart me at PTRB
       COGINIT myID,PTRB
```

The COGSTOP instruction is used to stop cogs. The 4 LSB's of the D/# operand supply the target cog ID.

```
COGSTOP #0                  'stop cog 0

COGID   myID                'stop me
COGSTOP myID
```

A cog can discover its own ID by doing a COGID instruction, which will return its ID into D[3:0], with upper bits cleared. This is useful, in case the cog wants to restart or stop itself, as shown above.

If COGID is used with WC, it will not overwrite D, but will return the status of cog D/# into C, where C=0 indicates the cog is free (stopped or never started) and C=1 indicates the cog is busy (started).

```
COGID   ThatCog  WC         'C=1 if ThatCog is busy
```

COG ATTENTION

Each cog can request the attention of other cogs by using the COGATN instruction:

```
COGATN  D/#                                 'get attention of cog(s), 2 clocks
```

The D/# operand supplies a 16-bit value in which bits 0..15 represent cogs 0..15. For each set bit, the corresponding cog will be strobed, causing an 'attention' event for POLLATN/WAITATN and interrupt use. The 16 attention strobe outputs from all cogs are OR'd together to form a composite set of 16 strobes, from which each cog receives its particular strobe.

```
COGATN  #%0000_0000_1111_0000           'request attention of cogs 4..7

POLLATN WC                              'has attention been requested?

WAITATN                                 'wait for attention request

JATN    S/#                             'jump to S/# if attention requested

JNATN   S/#                             'jump to S/# if attention not requested
```

In cases where multiple cogs may be requesting the attention of a single cog, some messaging structure may need to be implemented in hub RAM, in order to differentiate requests. In the main intended use case, the cog that is receiving an attention request knows which other cog is strobing it and how it is to respond.

COG REGISTER LAYOUT

The cog's 512 x 32 dual-port RAM, used for code and data, is mapped as follows:

```
$000..$1EF    RAM                 general-use code/data registers
$1F0          RAM / IJMP3         interrupt call   address for INT3
$1F1          RAM / IRET3         interrupt return address for INT3
$1F2          RAM / IJMP2         interrupt call   address for INT2
$1F3          RAM / IRET2         interrupt return address for INT2
$1F4          RAM / IJMP1         interrupt call   address for INT1
```

```
$1F5          RAM / IRET1         interrupt return address for INT1
$1F6          RAM / PA            CALLD-imm return, CALLPA parameter, or LOC address
$1F7          RAM / PB            CALLD-imm return, CALLPB parameter, or LOC address
$1F8          PTRA                pointer A to hub RAM
$1F9          PTRB                pointer B to hub RAM
$1FA          DIRA                output enables for P31..P0
$1FB          DIRB                output enables for P63..P32
$1FC          OUTA                output states for P31..P0
$1FD          OUTB                output states for P63..P32
$1FE          INA *               input states for P31..P0
$1FF          INB **              input states for P63..P32

 * also debug interrupt call address
** also debug interrupt return address
```

REGISTER INDIRECTION

Cog registers can be accessed indirectly most easily by using the ALTS/ALTD/ALTR instructions. These instructions sum their D[8:0] and S/#[8:0] values to compute an address that is directly substituted into the next instruction's S field, D field, or result register address (normally, this is the same as the D field). This all happens within the pipeline and does not affect the actual program code. The idea is that S/# can serve as a register base address and D can be used as an index.

Additionally, S[17:9] is always sign-extended and added to the D register for index updating. Normally, a nine-bit #address will be used for S, causing S[17:9] to be zero, so that D is unaffected:

```
      ALTS    index,#table       'set next S field to table+index
      MOV     OUTA,0             'output register[table+index] to OUTA

      ALTD    index,#table       'set next D field to table+index
      MOV     0,INA              'write INA to register[table+index]

      ALTR    index,#table       'set next write to table+index
      XOR     INA,INB            'write INA^INB to register[table+index]
```

For cases where base+index is not required, and a register holds the desired address, the S/# field can be omitted and it will be set to '#0' by the assembler:

```
      ALTS    pointer            'set next S field to pointer
      MOV     OUTA,0             'output register[pointer] to OUTA

      ALTD    pointer            'set next D field to pointer
      MOV     0,INA              'write INA to register[pointer]

      ALTR    pointer            'set next write to pointer
      XOR     INA,INB            'write INA^INB to register[pointer]
```

For accessing bit fields that span multiple registers, there is the ALTB instruction which sums D[13:5] and S/#[8:0] values to compute an address which is substituted into the next instruction's D field. It can be used with and without S/#:

```
      ALTB    bitindex,#base     'set next D field to base+bitindex[13:5]
```

```
BITC    0,bitindex          'write C to bit[bitindex[4:0]]

ALTB    bitindex            'set next D field to bitindex[13:5]
TESTB   0,bitindex  WC      'read bit[bitindex[4:0]] into C
```

There are also ALTxx instructions for facilitating nibble, byte, and word addressing of registers. They modify either the S or D field, as well as the N field of their associated and subsequent nibble, byte or word instruction. Like the other ALTx instructions, they can be used with or without S/#. Note that the associated nibble, byte, or word instruction can be a shortened-syntax alias of the full instruction, since two of its three fields will be filled in by the ALTxx instruction.

Nibble addressing:

```
ALTSN   index,#base         'set next D field to base+index[11:3], next N to index[2:0]
SETNIB  value               'set nibble to value ('SETNIB S/#' = 'SETNIB 0,S/#,#0')

ALTGN   index,#base         'set next S field to base+index[11:3], next N to index[2:0]
GETNIB  value               'get nibble into value ('GETNIB D' = 'GETNIB D,0,#0')

ALTGN   index,#base         'set next S field to base+index[11:3], next N to index[2:0]
ROLNIB  value               'ROL nibble into value ('ROLNIB D' = 'ROLNIB D,0,#0')
```

Byte addressing:

```
ALTSB   index,#base         'set next D field to base+index[10:2], next N to index[1:0]
SETBYTE value               'set byte to value ('SETBYTE S/#' = 'SETBYTE 0,S/#,#0')

ALTGB   index,#base         'set next S field to base+index[10:2], next N to index[1:0]
GETBYTE value               'get byte into value ('GETBYTE D' = 'GETBYTE D,0,#0')

ALTGB   index,#base         'set next S field to base+index[10:2], next N to index[1:0]
ROLBYTE value               'ROL byte into value ('ROLBYTE D' = 'ROLBYTE D,0,#0')
```

Word addressing:

```
ALTSW   index,#base         'set next D field to base+index[9:1], next N to index[0]
SETWORD value               'set word to value ('SETWORD S/#' = 'SETWORD 0,S/#,#0')

ALTGW   index,#base         'set next S field to base+index[9:1], next N to index[0]
GETWORD value               'get word into value ('GETWORD D' = 'GETWORD D,0,#0')

ALTGW   index,#base         'set next S field to base+index[9:1], next N to index[0]
ROLWORD value               'ROL word into value ('ROLWORD D' = 'ROLWORD D,0,#0')
```

For more complex S field, D field, and result register substitutions, there is the ALTI instruction. ALTI actually does a few different things. First, ALTI can be used to individually increment or decrement three different nine-bit fields within a register. Second, ALTI can substitute each of those fields (before incrementing or decrementing) into the next instruction's S field, D field, or result register address, in the same way ALTS, ALTD, and ALTR do. Lastly, ALTI can substitute D[31..18] into the next instruction's upper bits [31..18] to enable full instruction substitution with a register's contents.

```
ALTI    D,S/#               'modify D and/or next instruction's fields according to S/#
```

```
S/# = %rrr_ddd_sss_RRR_DDD_SSS

%rrr          Result register field D[27..19] increment/decrement masking
%ddd          D register field D[17..9] increment/decrement masking
%sss          S register field D[8..0] increment/decrement masking

              %rrr/%ddd/%sss:
              000 = 9 bits increment/decrement (default, full span)
              001 = 8 LSBs increment/decrement (256-register looped buffer)
              010 = 7 LSBs increment/decrement (128-register looped buffer)
              011 = 6 LSBs increment/decrement (64-register looped buffer)
              100 = 5 LSBs increment/decrement (32-register looped buffer)
              101 = 4 LSBs increment/decrement (16-register looped buffer)
              110 = 3 LSBs increment/decrement (8-register looped buffer)
              111 = 2 LSBs increment/decrement (4-register looped buffer)

%RRR          result register / instruction modification:
              000 = D[27..19] stays same, no result register substitution
              001 = D[27..19] stays same, but result register writing is canceled
              010 = D[27..19] decrements per %rrr, no result register substitution
              011 = D[27..19] increments per %rrr, no result register substitution
              100 = D[27..19] sets next instruction's result register, stays same
              101 = D[31..18] substitutes into next instruction's [31..18] (execute D)
              110 = D[27..19] sets next instruction's result register, decrements per %rrr
              111 = D[27..19] sets next instruction's result register, increments per %rrr

%DDD          D field modification:
              x0x = D[17..9] stays same
              x10 = D[17..9] decrements per %ddd
              x11 = D[17..9] increments per %ddd
              0xx = no D field substitution
              1xx = D[17..9] substitutes into next instruction's D field [17..9]

%SSS          S field modification:
              x0x = D[8..0] stays same
              x10 = D[8..0] decrements per %sss
              x11 = D[8..0] increments per %sss
              0xx = no S field substitution
              1xx = D[8..0] substitutes into next instruction's S field [8..0]
```

Here are some examples of ALTI usage:

```
       ALTI    ptrs,#%111_111     'set next D and S fields, increment ptrs[17:9] and
ptrs[8:0]
       ADD     0,0                'add registers
```

```
        ALTI    inst,#%101_100_100 'execute inst (same as 'ALTI inst')
        NOP                        'NOP becomes inst
```

The SETS/SETD/SETR instructions allow you to write the S field, D field and instruction field of a register without affecting other bits. They copy the lower 9 bits of S/# into their respective 9-bit field within D. These instructions are useful for establishing the fields that will be used by ALTI:

```
        SETS    D,S/#              'set D[8:0] to S/#[8:0]
        SETD    D,S/#              'set D[17:9] to S/#[8:0]
        SETR    D,S/#              'set D[27:19] to S/#[8:0]
```

SETS/SETD/SETR can also be used in self-modifying cog-register code. After modifying a cog register, It is necessary to elapse two instructions before executing the modified register, due to pipelining:

```
        SETR    inst,op            'set reg[27:19] to op[8:0]
        NOP                        'first spacer instruction, could be anything
        NOP                        'second spacer instruction, could be anything
inst    MOV     x,y                'operate on x using y, MOV can become AND/OR/XOR/etc.
```

BRANCH ADDRESSING

The following are branch instructions which use D[19:0] as an absolute address:

```
EEEE 1101011 CZ0 DDDDDDDDD 000101100        JMP     D
EEEE 1101011 CZ0 DDDDDDDDD 000101101        CALL    D
EEEE 1101011 CZ0 DDDDDDDDD 000101110        CALLA   D
EEEE 1101011 CZ0 DDDDDDDDD 000101111        CALLB   D
```

The JMPREL instruction uses D[19:0] as a relative address that steps whole instructions (in hub mode, D[17:0] << 2 is added to the program counter). If D is immediate, D[19:0] is a 9-bit zero-extended value:

```
EEEE 1101011 00L DDDDDDDDD 000110000        JMP     {#}D
```

These next branch instructions use S[19:0] as an absolute address, or, if S is immediate, they sign-extend the 9-bit S field and use that value as a relative address that steps whole instructions (in hub mode, the value gets shifted left two bits before being added to the program counter). This means that their immediate range is -256 to +255 instructions, relative to the instruction following the branch:

```
EEEE 1011001 CZI DDDDDDDDD SSSSSSSSS        CALLD   D,{#}S
EEEE 1011010 0LI DDDDDDDDD SSSSSSSSS        CALLPA  D,{#}S
EEEE 1011010 1LI DDDDDDDDD SSSSSSSSS        CALLPB  D,{#}S
EEEE 1011011 00I DDDDDDDDD SSSSSSSSS        DJZ     D,{#}S
EEEE 1011011 01I DDDDDDDDD SSSSSSSSS        DJNZ    D,{#}S
EEEE 1011011 10I DDDDDDDDD SSSSSSSSS        DJF     D,{#}S
EEEE 1011011 11I DDDDDDDDD SSSSSSSSS        DJNF    D,{#}S
EEEE 1011100 00I DDDDDDDDD SSSSSSSSS        IJZ     D,{#}S
EEEE 1011100 01I DDDDDDDDD SSSSSSSSS        IJNZ    D,{#}S
EEEE 1011100 10I DDDDDDDDD SSSSSSSSS        TJZ     D,{#}S
EEEE 1011100 11I DDDDDDDDD SSSSSSSSS        TJNZ    D,{#}S
EEEE 1011101 00I DDDDDDDDD SSSSSSSSS        TJF     D,{#}S
```

```
EEEE 1011101 01I DDDDDDDDD SSSSSSSSS        TJNF    D,{#}S
EEEE 1011101 10I DDDDDDDDD SSSSSSSSS        TJS     D,{#}S
EEEE 1011101 11I DDDDDDDDD SSSSSSSSS        TJNS    D,{#}S
EEEE 1011110 00I DDDDDDDDD SSSSSSSSS        TJV     D,{#}S
EEEE 1011110 01I 00000EEEE SSSSSSSSS        Jevent  {#}S
EEEE 1011110 01I 00001EEEE SSSSSSSSS        JNevent {#}S
```

There are five branch instructions and one 'locate' instruction which involve 20-bit immediate addresses. Their addresses can be either relative to the program counter (R=1) or absolute (R=0):

```
EEEE 1101100 RAA AAAAAAAAA AAAAAAAAA        JMP     #A
EEEE 1101101 RAA AAAAAAAAA AAAAAAAAA        CALL    #A
EEEE 1101110 RAA AAAAAAAAA AAAAAAAAA        CALLA   #A
EEEE 1101111 RAA AAAAAAAAA AAAAAAAAA        CALLB   #A
EEEE 11100WW RAA AAAAAAAAA AAAAAAAAA        CALLD   PA/PB/PTRA/PTRB,#A
EEEE 11101WW RAA AAAAAAAAA AAAAAAAAA        LOC     PA/PB/PTRA/PTRB,#A
```

Relative addressing is convenient for relocatable code, or code which can run from either cog RAM or hub RAM. Relative addressing is the default when cog code references cog labels or hub code references hub labels. On the other hand, absolute addressing is highly recommended, and forced by the assembler, when crossing between cog and hub domains.

Absolute addressing can be forced by the use of "\" after the "#".

The "@" operator can be used before an address label to return the hub address of that label, in case it was defined under an ORG directive to generate cog code, and the label would normally return the cog address..

The cases below illustrate use of the 20-bit immediate-address instructions and "\" and "@":

```
        ORGH    $01000
        ORG     0       'cog code

cog     JMP     #cog    '$FD9FFFFC      cog to cog, relative
        JMP     #\cog   '$FD800000      cog to cog, force absolute
        JMP     #@cog   '$FD801000      cog to hub, always absolute
        JMP     #\@cog  '$FD801000      cog to hub, always absolute

        JMP     #hub    '$FD802000      cog to hub, always absolute
        JMP     #\hub   '$FD802000      cog to hub, always absolute
        JMP     #@hub   '$FD802000      cog to hub, always absolute
        JMP     #\@hub  '$FD802000      cog to hub, always absolute

        ORGH    $02000  'hub code

hub     JMP     #cog    '$FD800000      hub to cog, always absolute
        JMP     #\cog   '$FD800000      hub to cog, always absolute
        JMP     #@cog   '$FD9FEFF4      hub to hub, relative
        JMP     #\@cog  '$FD801000      hub to hub, force absolute

        JMP     #hub    '$FD9FFFEC      hub to hub, relative
        JMP     #\hub   '$FD802000      hub to hub, force absolute
        JMP     #@hub   '$FD9FFFE4      hub to hub, relative
        JMP     #\@hub  '$FD802000      hub to hub, force absolute
```

INSTRUCTION REPEATING

Single or multiple instructions can be repeated without branching delays in cog/LUT memory using the REP instruction:

```
        REP     {#}D,{#}S           'execute {#}D[8:0] instructions {#}S[31:0] times
```

If D[8:0] = 0, nothing will be repeated. If D[8:0] > 0 and S[31:0] = 0 then D[8:0] instructions will be repeated indefinitely.

By changing the #1000 to #0, the DRVN instruction would be repeated indefinitely:

```
        REP     #1,#1000            'toggle pin 0 1000 times (1 instruction x 1000)
        DRVNOT  #0                  'output and toggle pin 0 (2 clocks per toggle)
```

In cases where you'd rather have the assembler keep track of the number of instructions, @label can be used:

```
        REP     @.end,reps          'repeat instruction block 'reps' times
        WFBYTE  x                   'write x to next byte in hub
        ADD     x,#1                'increment x
.end
```

REP works in hub memory, as well, but executes a hidden jump to get back to the top of the repeated instructions.

Any branch within the repeating instruction block will cancel REP activity. Interrupts will be ignored during REP looping.

INSTRUCTION SKIPPING

Cogs can initiate skipping sequences to selectively skip any of the next 32 instructions encountered. Skipping is accomplished by either cancelling instructions as they come through the pipeline from hub or cog/LUT memory - effectively turning them into 2-clock NOP instructions - or by leaping over them in cog/LUT memory. Skipping only works outside of interrupt service routines, in main code.

There are three instructions that initiate skipping:

```
        SKIP    {#}D        'skip by cancelling instructions sequentially per D[0]..D[31]
        SKIPF   {#}D        'like SKIP, but fast due to PC steps of 1..8 - cog/LUT only!
        EXECF   {#}D        'jump to D[9:0] in cog/LUT and initiate SKIPF using D[31:10]
```

In each case, D provides a bit pattern which is used LSB-first to determine whether the next instruction is cancelled/skipped (bit=1) or executed (bit=0). The D bit pattern is initially captured and subsequently shifted right by one bit for each instruction encountered.

Within in a skipping sequence, a CALL/CALLPA/CALLPB that is not skipped will execute all its nested subroutines normally, with the skipping sequence resuming after the returning RET/_RET_. This allows subroutines to be skipped or entirely executed without affecting the top-level skip sequence. As well, an interrupt service routine will execute normally during a skipping sequence, with the skipping sequence resuming upon its completion.

While SKIP-initiated skipping can take place in both hub and cog/LUT memory, SKIPF-initiated and EXECF-initiated skipping can only take place in cog/LUT memory. This is because the PC can be randomly stepped in cog/LUT memory, whereas the hub memory FIFO can only provide the next instruction, unless a full branch takes place, triggering a FIFO reload.

Here is a simplistic example of SKIP:

```
        SKIP   #%010110      'initiate skip sequence (skip 2nd, 3rd, 5th instruction)
        DRVN   #0            'drive and invert pin 0 (executes)
        DRVN   #1            'drive and invert pin 1 (NOP)
        DRVN   #2            'drive and invert pin 2 (NOP)
        DRVN   #3            'drive and invert pin 3 (executes)
        DRVN   #4            'drive and invert pin 4 (NOP)
        DRVN   #5            'drive and invert pin 5 (executes)
```

Skipping is very useful for getting increased functionality out of an otherwise-static sequence of instructions. Consider this sequence, which contains all the instructions needed to realize some address calculation:

```
addr    RFBYTE m             'offset - one of these three (3 possibilities)
        RFWORD m
        RFLONG m

        ADD    m,pbase       'base - one of these three (3 possibilities)
        ADD    m,vbase
        ADD    m,dbase

        SHL    i,#1          'index - zero to two of these three (4 possibilities)
        SHL    i,#2
        ADD    m,i
```

In the above sequence, the intention is to compute an address using an offset, a base, and an optional index. There are 3 x 3 x 4, or 36, useful permutations. If you wanted to use a byte offset, pbase, and a long index, you would want to execute only these four instructions from the 'addr' sequence:

```
        RFBYTE m             'offset
        ADD    m,pbase       'base
        SHL    i,#2          'index
        ADD    m,i
```

The skip pattern for just those four instructions would be %001_110_110. Assuming 'pat' holds that pattern, here is what the execution would look like using SKIP. Note that the 'addr' instruction sequence, shown above, follows the SKIP instruction and skipped instructions in the 'addr' sequence are now shown as NOPs:

```
        SKIP   pat           'initiate skip sequence (%001_110_110 in this case)

addr    RFBYTE m             'offset
        NOP
        NOP

        ADD    m,pbase       'base
        NOP
        NOP

        NOP                  'index
        SHL    i,#2
```

```
              ADD     m,i
```

If this code were located in cog/LUT memory, SKIPF could be used to speed things up by stepping over skipped instructions, instead of cancelling them in the pipeline. Here is what the execution would look like using SKIPF:

```
              SKIPF   pat             'initiate skip sequence (%001_110_110 in this case)

addr          RFBYTE  m               'offset
              ADD     m,pbase         'base
              SHL     i,#2            'index
              ADD     m,i
```

Now things are very efficient, with no cycles being wasted on NOPs. If SKIPF is used in hub exec, it will revert to SKIP behavior, cancelling instructions in the pipeline, instead of stepping over them.

Both SKIP and SKIPF can be preceded by _RET_ for an automatic branch before skipping commences:

```
              PUSH    #addr           'point to the addr routine
       _RET_  SKIPF   pat             'jump to addr and begin skipping fast using pat
```

The EXECF instruction is similar to PUSH+SKIPF, but uses a single long (D) to get both a 10-bit branch address and a 22-bit skip pattern. Here is the heart of a simple bytecode interpreter which uses EXECF:

```
              REP     #1,#8           'pre-stuff 8-level hardware stack with 'loop' address
              PUSH    #loop           'all RETs without CALLs will branch to 'loop'

loop          RFBYTE  i               'get a bytecode
              RDLUT   e,i             'lookup long in LUT
              EXECF   e               'jump to e[9:0] and SKIPF e[31:10], RETs branch to 'loop'
```

That bytecode interpreter takes only 2+3+4, or 9, clocks to get the next bytecode, look it up, then execute that bytecode's routine in cog/LUT memory with a custom 22-bit SKIPF pattern. If that bytecode's routine is just a 2-clock instruction preceded by a _RET_, it will take 4 clocks, due to the _RET_, for a total of 13 clocks, looping. Those 13 clocks can be reduced to only 8 clocks by using XBYTE, which is explained in the next section.

While SKIPF and EXECF normally step over skipped instructions in cog/LUT memory, there are some circumstances where they must cancel an instruction, instead, since it is already in the pipeline:

1) The first instruction is being skipped after the SKIPF/EXECF instruction (the LSB of the skip pattern is '1')
2) The 8th instruction in a row is being skipped (only 7 instructions can be stepped over at once)
3) Execution is from hub memory, not cog/LUT memory.

Each of these cancellations results in a 2-clock NOP instruction.

SKIP is fully compatible with REP, since SKIP only cancels instructions, allowing REP to maintain accurate instruction counts.

SKIPF would only work with REP if all SKIPF patterns resulted in the same instruction counts, which REP would have to be initiated with, as opposed to just length-of-code.

Special SKIPF Branching Rules

Within SKIPF sequences where CALL/CALLPA/CALLPB are used to execute subroutines in which skipping will be suspended until after RET, all CALL/CALLPA/CALLPB immediate branch addresses must be absolute in cases where the instruction after the CALL/CALLPA/CALLPB might be skipped. This is not possible for CALLPA/CALLPB but CALL can use '#\address' syntax to achieve absolute immediate addressing. CALL/CALLPA/CALLPB can all use registers as branch addresses, since they are absolute.

For non-CALL\CALLPA\CALLPB branches within SKIPF sequences, SKIPF will work through all immediate-relative branches, which are the default for immediate branches within cog/LUT memory. If an absolute-address branch is being used (#\label, register, or RET, for example), you must not skip the instruction after the branch. This is not a problem with immediate-relative branches, however, since the variable PC stepping works to advantage, by landing the PC at the first instruction of interest at, or beyond, the branch address.

BYTECODE EXECUTION

Cogs can execute custom bytecodes from hub RAM using XBYTE. XBYTE is a hidden instruction that executes on a hardware stack return (RET/_RET_) to $1FF. Such a return does not pop the stack, so that each RET/_RET_ causes another bytecode to be fetched and executed. This process has a total overhead of only 6 clocks, excluding the bytecode routine. The bytecode routine could be just a 2-clock instruction with a _RET_ prefix, making the total XBYTE loop take only 8 clocks.

XBYTE performs the following steps to make a complete bytecode executor:

Clock	Phase	Hidden Activity	Description
1	go	RFBYTE bytecode	Last clock of instruction which is executing a RET/_RET_ to $1FF **Fetch bytecode from FIFO initialized via prior RDFAST**
2	get	RDLUT fx(bytecode), write bytecode to $1F6	1st clock of 1st cancelled instruction **Read lookup RAM according to bytecode, write bytecode to PA**
3	go	LUT data → D	2nd clock of 1st cancelled instruction **Get lookup RAM long into D for EXECF**
4	get	EXECF D	1st clock of 2nd cancelled instruction **Execute EXECF**
5	go	EXECF D, branch, write GETPTR to $1F7, write C and Z (optional)	2nd clock of 2nd cancelled instruction **Do EXECF branch, write FIFO pointer to PB, write C/Z if enabled**
6	get	flush pipeline	1st clock of 3rd cancelled instruction
7	go	reload pipeline	2nd clock of 3rd cancelled instruction
8	get	<none>	1st clock of 1st instruction of bytecode routine, loop to clock 1 if _RET_

The bytecode translation table in LUT memory must consist of long data which EXECF would use.

Starting XBYTE and establishing its operating mode is done all at once by a '_RET_ SETQ {#}D' instruction, with the top of the hardware stack holding $1FF.

Additional '_RET_ SETQ {#}D' instructions can be executed to alter the XBYTE mode for subsequent bytecodes.

To alter the XBYTE mode for the next bytecode, only, a '_RET_ SETQ2 {#}D' instruction can be executed. This is useful for engaging singular bytecodes from alternate sets, without having to do any XBYTE mode restoration, afterwards.

Bits	SETQ/SETQ2 {#}D value	LUT base address	LUT index b = bytecode	LUT EXECF address
8	%A000000xF	%A00000000	I = b[7:0]	AIIIIIIII
8	%ABBBB00xF %BBBB > 0	%A00000000	if b[7:4] < %BBBB then I = b[7:0] if b[7:4] >= %BBBB then I = b[7:4] - %BBBB	%AIIIIIIII %ABBBBIIII
7	%AAxx0010F	%AA0000000	I = b[6:0]	%AAIIIIIII
7	%AAxx0011F	%AA0000000	I = b[7:1]	%AAIIIIIII
6	%AAAx1010F	%AAA000000	I = b[5:0]	%AAAIIIIII
6	%AAAx1011F	%AAA000000	I = b[7:2]	%AAAIIIIII
5	%AAAAx100F	%AAAA00000	I = b[4:0]	%AAAAIIIII
5	%AAAAx101F	%AAAA00000	I = b[7:3]	%AAAAIIIII
4	%AAAAA110F	%AAAAA0000	I = b[3:0]	%AAAAAIIII
4	%AAAAA111F	%AAAAA0000	I = b[7:4]	%AAAAAIIII

The %ABBBB00xF setting allows sets of 16 bytecodes which use identical LUT values to be represented by a single LUT value, saving 15 LUT locations. This is useful when the bytecode, which is always written to PA, is used as an operand within the bytecode routine.

The %F bit of the SETQ/SETQ2 {#}D value enables C and Z to receive bits 1 and 0 of the final LUT address. This is useful for having the flags differentiate behavior within a bytecode routine, especially in cases of conditional looping, where a SKIPF pattern would have been insufficient, on its own:

SETQ/SETQ2 {#}D value	Flag Writing
%xxxxxxxx0	Do not affect flags on XBYTE
%xxxxxxxx1	Write LUT EXECF address bits 1 and 0 to C and Z

To start executing bytecodes, use the following instruction sequence, but with the appropriate SETQ operand:

```
        PUSH    #$1FF           'push #$1FF onto the hardware stack
_RET_   SETQ    #$100           '256-long EXECF table at lut $100, start XBYTE
```

See the "xbyte.spin2" file in the zip file.

PIXEL OPERATIONS

Each cog has a pixel mixer which can combine one pixel with another pixel in many different ways. A pixel consists of four byte fields within a 32-bit cog register. Pixel operations occur between each pair of D and S bytes, and they take seven clock cycles to complete:

```
ADDPIX  D,S/#           'add bytes with saturation
MULPIX  D,S/#           'multiply bytes ($FF = 1.0)
BLNPIX  D,S/#           'alpha-blend bytes according to SETPIV value
MIXPIX  D,S/#           'mix bytes according to SETPIX/SETPIV value
```

There are two pixel mixer setup instructions:

```
SETPIV  D/#             'set blend factor V[7:0] to D/#[7:0]
SETPIX  D/#             'set MIXPIX mode M[5:0] to D/#[5:0]
```

When a pixel mixer instruction executes, a sum-of-products-with-saturation computation is performed on each D and S byte pair:

```
D[31:24] = ((D[31:24] * DMIX + S[31:24] * SMIX + $FF) >> 8) max $FF
D[23:16] = ((D[23:16] * DMIX + S[23:16] * SMIX + $FF) >> 8) max $FF
D[15:08] = ((D[15:08] * DMIX + S[15:08] * SMIX + $FF) >> 8) max $FF
D[07:00] = ((D[07:00] * DMIX + S[07:00] * SMIX + $FF) >> 8) max $FF
```

Here are the DMIX and SMIX terms, according to each instruction:

	DMIX	SMIX
ADDPIX	$FF	$FF
MULPIX	S[byte]	$00
BLNPIX	!V	V
MIXPIX	M[5:3] = %000 → $00 M[5:3] = %001 → $FF M[5:3] = %010 → V M[5:3] = %011 → !V M[5:3] = %100 → S[byte] M[5:3] = %101 → !S[byte] M[5:3] = %110 → D[byte] M[5:3] = %111 → !D[byte]	M[2:0] = %000 → $00 M[2:0] = %001 → $FF M[2:0] = %010 → V M[2:0] = %011 → !V M[2:0] = %100 → S[byte] M[2:0] = %101 → !S[byte] M[2:0] = %110 → D[byte] M[2:0] = %111 → !D[byte]

I/O PIN TIMING

I/O pins are controlled by cogs via the following cog registers:

DIRA - output enable bits for P0..P31 (active high)
DIRB - output enable bits for P32..P63 (active high)
OUTA - output state bits for P0..P31 (corresponding DIRA bit must be high to enable output)
OUTB - output state bits for P32..P63 (corresponding DIRB bit must be high to enable output)

I/O pins are read by cogs via the following cog registers:

INA - input state bits for P0..P31
INB - input state bits for P32..P63

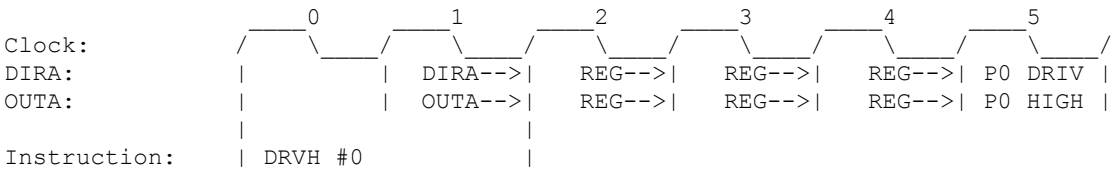
Aside from general-purpose instructions which may operate on DIRA/DIRB/OUTA/OUTB, there are special pin instructions which operate on singular bits within these registers:

DIRL/DIRH/DIRC/DIRNC/DIRZ/DIRNZ/DIRRND/DIRNOT {#}D - affect pin D bit in DIRx
OUTL/OUTH/OUTC/OUTNC/OUTZ/OUTNZ/OUTRND/OUTNOT {#}D - affect pin D bit in OUTx
FLTL/FLTH/FLTC/FLTNC/FLTZ/FLTNZ/FLTRND/FLTNOT {#}D - affect pin D bit in OUTx, clear bit in DIRx
DRVL/DRVH/DRVC/DRVNC/DRVZ/DRVNZ/DRVRND/DRVNOT {#}D - affect pin D bit in OUTx, set bit in DIRx

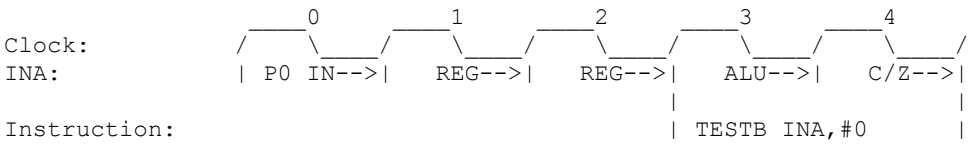
As well, aside from general-purpose instructions which may read INA/INB, there are special pin instructions which can read singular bits within these registers:

TESTP {#}D WC/WZ/ANDC/ANDZ/ORC/ORZ/XORC/XORZ -read pin D bit in INx and affect C or Z
TESTPN {#}D WC/WZ/ANDC/ANDZ/ORC/ORZ/XORC/XORZ -read pin D bit in !INx and affect C or Z

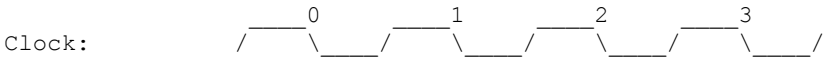
When a DIRx/OUTx bit is changed by any instruction, it takes THREE additional clocks after the instruction before the pin starts transitioning to the new state. Here this delay is demonstrated using DRVH:



When an INx register is read by an instruction, it will reflect the state of the pins registered TWO clocks before the start of the instruction. Here this delay in demonstrated using TESTB:



When a TESTP/TESTPN instruction is used to read a pin, the value read will reflect the state of the pin registered ONE clock before the start of the instruction. So, TESTP/TESTPN get fresher INx data than is available via the INx registers:



```
INA:            | P0 IN-->|   REG-->|   REG-->|   C/Z-->|
                                    |                   |
Instruction:                        | TESTP #0          |
```

EVENTS

Cogs monitor and track 16 different background events:

- An interrupt occurred
- CT passed CT1 (CT is the 32-bit free-running global counter)
- CT passed CT2
- CT passed CT3
- Selectable event 1 occurred
- Selectable event 2 occurred
- Selectable event 3 occurred
- Selectable event 4 occurred
- A pattern match or mismatch occurred on either INA or INB
- Hub FIFO block-wrap occurred - a new start address and block count were loaded
- Streamer command buffer is empty - it's ready to accept a new command
- Streamer finished - it ran out of commands, now idle
- Streamer NCO rollover occurred
- Streamer read lookup RAM location $1FF
- Attention was requested by another cog or other cogs
- GETQX/GETQY executed without any CORDIC results available

Events are tracked and can be polled, waited for, and used as interrupt sources.

Before explaining the details, consider the event-related instructions.

First are the POLLxxx instructions which simultaneously return their event-occurred flag into C and clear their event-occurred flag (unless it’s being set again by the event sensor):

		Interrupt source (0=off):
POLLINT	Poll the interrupt-occurred event flag	-
POLLCT1	Poll the CT-passed-CT1 event flag	1
POLLCT2	Poll the CT-passed-CT2 event flag	2
POLLCT3	Poll the CT-passed-CT3 event flag	3
POLLSE1	Poll the selectable-event-1 event flag	4
POLLSE2	Poll the selectable-event-2 event flag	5
POLLSE3	Poll the selectable-event-3 event flag	6
POLLSE4	Poll the selectable-event-4 event flag	7
POLLPAT	Poll the pin-pattern-detected event flag	8
POLLFBW	Poll the hub-FIFO-interface-block-wrap event flag	9
POLLXMT	Poll the streamer-empty event flag	10
POLLXFI	Poll the streamer-finished event flag	11
POLLXRO	Poll the streamer-NCO-rollover event flag	12
POLLXRL	Poll the streamer-lookup-RAM-$1FF-read event flag	13
POLLATN	poll the attention-requested event flag	14
POLLQMT	Poll the CORDIC-read-but-no-results event flag	15

Next are the WAITxxx instructions, which will wait for their event-occurred flag to be set (in case it's not, already) and then clear their event-occurred flag (unless it's being set again by the event sensor), before resuming.

By doing a SETQ right before one of these instructions, you can supply a future CT target value which will be used to end the wait prematurely, in case the event-occurred flag never went high before the CT target was reached. When using SETQ with 'WAITxxx WC', C will be set if the timeout occurred before the event; otherwise, C will be cleared.

WAITINT	Wait for an interrupt to occur, stalls the cog to save power
WAITCT1	Wait for the CT-passed-CT1 event flag
WAITCT2	Wait for the CT-passed-CT2 event flag
WAITCT3	Wait for the CT-passed-CT3 event flag
WAITSE1	Wait for the selectable-event-1 event flag
WAITSE2	Wait for the selectable-event-2 event flag
WAITSE3	Wait for the selectable-event-3 event flag
WAITSE4	Wait for the selectable-event-4 event flag
WAITPAT	Wait for the pin-pattern-detected event flag
WAITFBW	Wait for the hub-FIFO-interface-block-wrap event flag
WAITXMT	Wait for the streamer-empty event flag
WAITTXFI	Wait for the streamer-finished event flag
WAITXRO	Wait for the streamer-NCO-rollover event flag
WAITXRL	Wait for the streamer-lookup-RAM-$1FF-read event flag
WAITATN	Wait for the attention-requested event flag

There's no 'WAITQMT' because the event could not happen while waiting.

Last are the 'Jxxx/JNxxx S/#' instructions, which each jump to S/# if their event-occurred flag is set (Jxxx) or clear (JNxxx). Whether or not a branch occurs, the event-occurred flag will be cleared, unless it's being set again by the event sensor.

JINT/JNINT	Jump to S/# if the interrupt-occurred event flag is set/clear
JCT1/JNCT1	Jump to S/# if the CT-passed-CT1 event flag is set/clear
JCT2/JNCT2	Jump to S/# if the CT-passed-CT2 event flag is set/clear
JCT3/JNCT3	Jump to S/# if the CT-passed-CT3 event flag is set/clear
JSE1/JNSE1	Jump to S/# if the selectable-event-1 event flag is set/clear
JSE2/JNSE2	Jump to S/# if the selectable-event-2 event flag is set/clear
JSE3/JNSE3	Jump to S/# if the selectable-event-3 event flag is set/clear
JSE4/JNSE4	Jump to S/# if the selectable-event-4 event flag is set/clear
JPAT/JNPAT	Jump to S/# if the pin-pattern-detected event flag is set/clear
JFBW/JNFBW	Jump to S/# if the hub-FIFO-interface-block-wrap event flag is set/clear
JXMT/JNXMT	Jump to S/# if the streamer-empty event flag is set/clear
JXFI/JNXFI	Jump to S/# if the streamer-finished event flag is set/clear
JXRO/JNXRO	Jump to S/# if the streamer-NCO-rollover event flag is set/clear
JXRL/JNXRL	Jump to S/# if the streamer-lookup-RAM-$1FF-read event flag is set/clear
JATN/JNATN	Jump to S/# if the attention-requested event flag is set/clear
JQMT/JNQMT	Jump to S/# if the CORDIC-read-but-no-results event flag is set/clear

Here are detailed descriptions of each event flag. Understand that the 'set' events can also be used as interrupt sources (except in the case of the first flag which is set when an interrupt occurs):

POLLINT/WAITINT event flag

- Cleared on cog start.

- Set whenever interrupt 1, 2, or 3 occurs (debug interrupts are ignored).
- Also cleared on POLLINT/WAITINT/JINT/JNINT.

POLLCT1/WAITCT1 event flag

- Cleared on ADDCT1.
- Set whenever CT passes the result of the ADDCT1 (MSB of CT minus CT1 is 0).
- Also cleared on POLLCT1/WAITCT1/JCT1/JNCT1.

POLLCT2/WAITCT2 event flag

- Cleared on ADDCT2.
- Set whenever CT passes the result of the ADDCT2 (MSB of CT minus CT2 is 0).
- Also cleared on POLLCT2/WAITCT2/JCT2/JNCT2.

POLLCT3/WAITCT3 event flag

- Cleared on ADDCT3.
- Set whenever CT passes the result of the ADDCT3 (MSB of CT minus CT3 is 0).
- Also cleared on POLLCT3/WAITCT3/JCT3/JNCT3.

POLLSE1/WAITSE1 event flag

- Cleared on 'SETSE1 D/#', for which D/# selects the event:

  %000_00_00AA = this cog reads LUT address %1111111AA
  %000_00_01AA = this cog writes LUT address %1111111AA
  %000_00_10AA = odd/even companion cog reads LUT address %1111111AA
  %000_00_11AA = odd/even companion cog writes LUT address %1111111AA

  %000_01_LLLL = hub lock %LLLL rises
  %000_10_LLLL = hub lock %LLLL falls
  %000_11_LLLL = hub lock %LLLL changes

  %001_PPPPPP = INA/INB bit of pin %PPPPPP rises
  %010_PPPPPP = INA/INB bit of pin %PPPPPP falls
  %011_PPPPPP = INA/INB bit of pin %PPPPPP changes

  %10x_PPPPPP = INA/INB bit of pin %PPPPPP is low
  %11x_PPPPPP = INA/INB bit of pin %PPPPPP is high

- Set whenever the selected event occurs.
- Also cleared on POLLSE1/WAITSE1/JSE1/JNSE1.

POLLSE2/WAITSE2 event flag

- Cleared on 'SETSE2 D/#', for which D/# selects the event (see POLLSE1/WAITSE1 event flag).
- Set whenever the selected event occurs.

- Also cleared on POLLSE2/WAITSE2/JSE2/JNSE2.

POLLSE3/WAITSE3 event flag

- Cleared on ‘SETSE3 D/#’, for which D/# selects the event (see POLLSE1/WAITSE1 event flag).
- Set whenever the selected event occurs.
- Also cleared on POLLSE3/WAITSE3/JSE3/JNSE3.

POLLSE4/WAITSE4 event flag

- Cleared on ‘SETSE4 D/#’, for which D/# selects the event (see POLLSE1/WAITSE1 event flag).
- Set whenever the selected event occurs.
- Also cleared on POLLSE4/WAITSE4/JSE4/JNSE4.

POLLPAT/WAITPAT event flag

- Cleared on SETPAT
- Set whenever (INA & D) != S after ‘SETPAT D/#,S/#’ with C=0 and Z=0.
- Set whenever (INA & D) == S after ‘SETPAT D/#,S/#’ with C=0 and Z=1.
- Set whenever (INB & D) != S after ‘SETPAT D/#,S/#’ with C=1 and Z=0.
- Set whenever (INB & D) == S after ‘SETPAT D/#,S/#’ with C=1 and Z=1.
- Also cleared on POLLPAT/WAITPAT/JPAT/JNPAT.

POLLFBW/WAITFBW event flag

- Cleared on RDFAST/WRFAST/FBLOCK.
- Set whenever the the hub RAM FIFO interface exhausts its block count and reloads its ‘block count’ and ‘start address’.
- Also cleared on POLLFBW/WAITFBW/JFBW/JNFBW.

POLLXMT/WAITXMT event flag

- Cleared on XINIT/XZERO/XCONT.
- Set whenever the the streamer is ready for a new command.
- Also cleared on POLLXMT/WAITXMT/JXMT/JNXMT.

POLLXFI/WAITXFI event flag

- Cleared on XINIT/XZERO/XCONT.
- Set whenever the the streamer runs out of commands.
- Also cleared on POLLXFI/WAITXFI/JXFI/JNXFI.

POLLXRO/WAITXRO event flag

- Cleared on XINIT/XZERO/XCONT.
- Set whenever the the streamer NCO rolls over.

- Also cleared on POLLXRO/WAITXRO/JXRO/JNXRO.

POLLXRL/WAIXTRL event flag

- Cleared on cog start.
- Set whenever location $1FF of the lookup RAM is read by the streamer.
- Also cleared on POLLXRL/WAITXRL/JXRL/JNXRL.

POLLATN/WAITATN event flag

- Cleared on cog start.
- Set whenever any cogs request attention.
- Also cleared on POLLATN/WAITATN/JATN/JNATN.

POLLQMT event flag

- Cleared on cog start.
- Set whenever GETQX/GETQY executes without any CORDIC results available or in progress.
- Also cleared on POLLQMT/WAITQMT/JQMT/JNQMT.

Example: ADDCT1/WAITCT1

'ADDCT1 D,S/#' must be used to establish a CT target. This is done by first using 'GETCT D' to get the current CT value into a register, and then using ADDCT1 to add into that register, thereby making a future CT target, which, when passed, will trigger the CT-passed-CT1 event and set the related event flag.

```
        GETCT   x               'get initial CT
        ADDCT1  x,#500          'make initial CT1 target

.loop   WAITCT1                 'wait for CT to pass CT1 target
        ADDCT1  x,#500          'update CT1 target
        DRVNOT  #0              'toggle P0
        JMP     #.loop          'loop to the WAITCT1
```

It doesn't matter what register is used to keep track of the CT1 target. Whenever ADDCT1 executes, S/# is added into D, and the result gets copied into a dedicated CT1 target register that is compared to CT on every clock. When the CT1 target passes CT, the event flag is set. ADDCT1 clears the CT-passed-CT1 event flag to help with initialization and cycling.

INTERRUPTS

Each cog has three interrupts: INT1, INT2, and INT3.

INT1 has the highest priority and can interrupt INT2 and INT3.

INT2 has the middle priority and can interrupt INT3.

INT3 has the lowest priority and can only interrupt non-interrupt code.

The STALLI instruction can be used to hold off INT1, INT2 and INT3 interrupt branches indefinitely, while the ALLOWI instruction allows those interrupt branches to occur. Critical blocks of code can, therefore, be protected from interruption by beginning with STALLI and ending with ALLOWI.

There are 16 interrupt event sources, selected by a 4-bit pattern:

0	<off>, default on cog start for INT1/INT2/INT3 event sources
1	CT-passed-CT1, established by ADDCT1
2	CT-passed-CT2, established by ADDCT2
3	CT-passed-CT3, established by ADDCT3
4	SE1 event occurred, established by SETSE1
5	SE2 event occurred, established by SETSE2
6	SE3 event occurred, established by SETSE3
7	SE4 event occurred, established by SETSE4
8	Pin pattern match or mismatch occurred, established by SETPAT
9	Hub RAM FIFO interface wrapped and reloaded, established by RDFAST/WRFAST/FBLOCK
10	Streamer is ready for another command, established by XINIT/XZERO/ZCONT
11	Streamer ran out of commands, established by XINIT/XZERO/ZCONT
12	Streamer NCO rolled over, established by XINIT/XZERO/XCONT
13	Streamer read location $1FF of lookup RAM
14	Attention requested by other cog(s)
15	GETQX/GETQY executed without any CORDIC results available or in progress

To set up an interrupt, you need to first point its IJMP register to your interrupt service routine (ISR). When the interrupt occurs, it will jump to where the IJMP register points and simultaneously store the C/Z flags and return address into the adjacent IRET register:

```
$1F0          RAM / IJMP3          interrupt call   address for INT3
$1F1          RAM / IRET3          interrupt return address for INT3
$1F2          RAM / IJMP2          interrupt call   address for INT2
$1F3          RAM / IRET2          interrupt return address for INT2
$1F4          RAM / IJMP1          interrupt call   address for INT1
$1F5          RAM / IRET1          interrupt return address for INT1
```

When your ISR is done, it can do a RETIx instruction to return to the interrupted code. The RETIx instructions are actually CALLD instructions:

```
RETI1           =    CALLD  INB,IRET1   WCZ
RETI2           =    CALLD  INB,IRET2   WCZ
RETI3           =    CALLD  INB,IRET3   WCZ
```

The CALLD with D = <any register>, S = IRETx, and WCZ, signals the cog that the interrupt is complete. This causes the cog to clear its internal interrupt-busy flag for that interrupt, so that another interrupt can occur. INB (read-only) is used as D for RETIx instructions to effectively make the CALLD into a JMP back to the interrupted code.

Instead of using RETIx, though, you could use RESIx to have your ISR resume at the next instruction when the next interrupt

occurs:

```
RESI1    =    CALLD   IJMP1,IRET1   WCZ
RESI2    =    CALLD   IJMP2,IRET2   WCZ
RESI3    =    CALLD   IJMP3,IRET3   WCZ
```

Once you've got the IJMPx register configured to point to your ISR, you can enable the interrupt. This is done using the SETINTx instruction:

```
SETINT1 D/#     Set INT1 event to 0..15 (see table above)
SETINT2 D/#     Set INT2 event to 0..15 (see table above)
SETINT3 D/#     Set INT3 event to 0..15 (see table above)
```

Interrupts may be forced in software by the TRGINTx instructions:

```
TRGINT1         Trigger INT1
TRGINT2         Trigger INT2
TRGINT3         Trigger INT3
```

Interrupts that have been triggered and are waiting to branch may be nixed in software by the NIXINTx instructions. These instructions are only useful in main code after STALLI executes or in an ISR which needs to stop a lower-level interrupt from executing after the current ISR exits:

```
NIXINT1         Nix INT1
NIXINT2         Nix INT2
NIXINT3         Nix INT3
```

Interrupts can be stalled or allowed using the following instructions:

```
ALLOWI          Allow waiting and future interrupt branches to occur indefinitely (default mode on cog start)
STALLI          Stall interrupt branches indefinitely until ALLOWI executes
```

When an interrupt event occurs, certain conditions must be met before the interrupt branch can happen:

- ALTxx / CRCNIB / SCLU / SCL / GETXACC / SETQ / SETQ2 / XORO32 / XBYTE must not be executing
- AUGS must not be executing or waiting for a S/# instruction
- AUGD must not be executing or waiting for a D/# instruction
- REP must not be executing or active
- STALLI must not be executing or active
- The cog must not be stalled in any WAITx instruction

Once these conditions are all met, any pending interrupt is allowed to branch, with priority given to INT1, then INT2, and then INT3.

Interrupt branches are realized, internally, by inserting a 'CALLD IRETx,IJMPx WCZ' into the instruction pipeline while holding the program counter at its current value, so that the interrupt later returns to the proper address.

Interrupts loop through these three states:

1) Waiting for interrupt event
2) Waiting for interrupt branch
3) Executing interrupt service routine

During states 2 and 3, any intervening interrupt events are ignored. When state 1 is returned to, a new interrupt event will be waited for.

The status of interrupts and events can be read into a register via the 'GETINT D' instruction. D will have the following fields:

```
%SSSS_SSSS_KICC_BBAA_TTTT_TTTT_TTTT_TTTT

       %SSSSSSSS are pending SKIP[7:0] bits

       %K indicates SKIP[31:8] is non-zero

       %I indicates STALLI is in effect

       %CC, %BB, %AA are the interrupt states for INT3, INT2, INT1, respectively:

              %0x    = waiting for interrupt event
              %10    = waiting for interrupt branch
              %11    = executing interrupt service routine

       %TTTT_TTTT_TTTT_TTTT are the event trap flags, listed from top to bottom:

              bit 15 = GETQX/GETQY executed without prior CORDIC command
              bit 14 = attention requested by cog(s)
              bit 13 = streamer read location $1FF of lookup RAM
              bit 12 = streamer NCO rolled over
              bit 11 = streamer finished, now idle
              bit 10 = streamer ready to accept new command
              bit 9  = hub RAM FIFO interface loaded block count and start address
              bit 8  = pin pattern match occurred
              bit 7  = SE4 event occurred
              bit 6  = SE3 event occurred
              bit 5  = SE2 event occurred
              bit 4  = SE1 event occurred
              bit 3  = CT-passed-CT1
              bit 2  = CT-passed-CT2
              bit 1  = CT-passed-CT3
              bit 0  = INT1, INT2, or INT3 occurred
```

Example: Using INT1 as a CT1 interrupt

```
        org

start   mov     ijmp1,#isr1      'set int1 vector

        setint1 #1               'set int1 for ct-passed-ct1 event
```

```
        getct   ct1             'set initial ct1 target
        addct1  ct1,#50

                                'main program, gets interrupted
loop    drvnot  #0              'toggle p0
        jmp     #loop           'loop

                                'int1 isr, runs once every 50 clocks
isr1    drvnot  #1              'toggle p1
        addct1  ct1,#50         'update ct1 target
        reti1                   'return to main program

ct1     res                     'reserve long for ct1
```

DEBUG INTERRUPT

Each cog has three prioritized interrupts: INT1, INT2, and INT3.

In addition to these three visible interrupts, there is a fourth “hidden” interrupt that has priority over all the others. It is the debug interrupt, and it is inaccessible to normal cog programs. It is controlled by enabling debug interrupts on cogs of interest, and then setting up initial 16-long programs at the end of hub RAM which

setting up code in the hub which intercepts an initial debug interrupt that occurs every time a cog is started via COGINIT.

When the chip boots from reset, for each cog, a long at the end hub RAM is initialized with a single-instruction debug ISR (interrupt service routine). This initial ISR simply exits the debug interrupt and returns to the intended program. Note that while 16 ISR's are shown below, there may only be 8, 4, 2, or 1, building downwards from the end of hub memory, depending on how many cogs the particular chip contains:

```
$FFFFC = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 0  initial debug ISR
$FFFF8 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 1  initial debug ISR
$FFFF4 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 2  initial debug ISR
$FFFF0 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 3  initial debug ISR
$FFFEC = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 4  initial debug ISR
$FFFE8 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 5  initial debug ISR
$FFFE4 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 6  initial debug ISR
$FFFE0 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 7  initial debug ISR
$FFFDC = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 8  initial debug ISR
$FFFD8 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 9  initial debug ISR
$FFFD4 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 10 initial debug ISR
$FFFD0 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 11 initial debug ISR
$FFFCC = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 12 initial debug ISR
$FFFC8 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 13 initial debug ISR
$FFFC4 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 14 initial debug ISR
$FFFC0 = $FB3BFFFF  RETI0 (CALLD INB,INB WCZ)        cog 15 initial debug ISR
```

These RETI0 instructions can be replaced with JMPs to more complex debug ISR’s. More on that later.

During normal program execution, INA and INB are read-only registers which reflect the states of the I/O pins. Writing to INA/INB has no effect on anything.

During debug ISR's, INA and INB become readable/writable RAM registers which are used for debug interrupt call and return addresses.

When a cog is started, the RAM register hidden behind INA is initialized with ($FFFFC - cogid*4). This establishes an initial debug ISR call address which points to one of those 16 locations listed above.

Just before the first instruction of the intended cog program executes, a debug interrupt occurs, causing a 'CALLD INB,INA WCZ' to execute. This results in a branch to ($FFFFC - cogid*4).

If the instruction at ($FFFFC - cogid*4) is the initially-planted RETI0 (CALLD INB,INB WCZ), the debug interrupt ends and the cog program executes normally.

If the instruction at ($FFFFC - cogid*4) is a JMP to a more complex debug interrupt handler, it could do many different things:

- It could dump the cog RAM into hub RAM, using INA as workspace to cover its tracks.
- It could wait for some user input.
- It could repoint the debug interrupt call address in INA to somewhere else.
- It could load a special debug interrupt handler into cog or lookup RAM and point INA to it.
- It could set up the next debug interrupt condition before returning to the cog program.

One thing it would likely do is set the next debug interrupt condition using the BRK instruction, before doing an RETI0 to return to the cog program. If no new condition is set before RETI0 executes, no more debug interrupts will occur until the cog is restarted by another COGINIT.

The BRK instruction has a dual personality - one during normal program execution, and another during debug ISR's.

During normal program execution, BRK {#}D sets an 8-bit code in D[7:0] sends an asynchronous 'break' pulse to a cog, causing a debug interrupt to occur. This is useful for polling a cog to see what it is doing, assuming it has been configured to respond the 'break'.

During normal program execution, SETBRK sends an asynchronous 'break' pulse to a cog, causing a debug interrupt to occur. This is useful for polling a cog to see what it is doing, assuming it has been configured to respond the 'break'.

SETBRK D/# - during normal program execution

```
D/# = %xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_CCCC

                %CCCC: the cog which will be sent the 'break' pulse
```

During debug ISR's, 'SETBRK D/#' does three things:

- It reveals normally-hidden state information about the cog.
- It allows you to force INA/INB to read back pin states, in case you need to see the pins.
- It allows you to set the next debug interrupt condition.

SETBRK D/# - during debug ISR's

```
D/# = %xxxx_PPPPPPPPPPPPPPPPPPP_x_GFEDCBA

  %PPPPPPPPPPPPPPPPPPP: 20-bit breakpoint address
                    %G: 1 = make INA/INB read pin states, not RAM
```

```
                %F: 1 = interrupt on asynchronous 'break'
                %E: 1 = interrupt on breakpoint address match
                %D: 1 = interrupt on INT3 ISR code (single step)
                %C: 1 = interrupt on INT2 ISR code (single step)
                %B: 1 = interrupt on INT1 ISR code (single step)
                %A: 1 = interrupt on non-ISR code  (single step)

If D is a register, %HHHH_HGFF_FFFF_FFFF_EEED_CCCC_BBBB_AAAA is written back to D

            %HHHHH: CORDIC result inventory count
                %G: XBYTE, top stack value is $001F8..$001FF
        %FFFFFFFFFF: XBYTE, SETQ value
               %EEE: XBYTE, 0..7 = 8..1 LSBs/MSBs
                 %D: LUT sharing is enabled
              %CCCC: INT3 selector, established by SETINT3
              %BBBB: INT2 selector, established by SETINT2
              %AAAA: INT1 selector, established by SETINT1
```

Within a debug ISR, you can execute any number of SETBRK's. The last one, though, will determine the condition, if any, on which the next debug interrupt will occur.

Upon entry into a debug ISR, it's as if a 'SETBRK #0' has executed. That's why executing only an RETI0 causes debug interrupts to cease. To keep debug interrupts going, you need to keep doing SETBRK's with one or more of those lower 6 bits sets. Otherwise, there will be no more interrupts until the cog is restarted.

What terminates a debug interrupt is not only RETI0, but any D-register variant (CALLD anyreg,INB WCZ). Instead of executing RETI0, you could execute RESI0 (CALLD INA,INB WCZ) to have your debug ISR resume at the next instruction, upon the next debug interrupt.

This debug interrupt scheme was designed to operate stealthily, without any cooperation from the cog programs being debugged. All control has been placed in the debug ISR, which is configurable only via those last longs in hub RAM. This isolation from normal programming is intended to prevent, or at least discourage, programmers from making any aspect of the debug interrupt system part of their application, thereby rendering it compromised as a standard debugging mechanism.
qa

CORDIC Solver

In the hub, there is a 54-stage pipelined CORDIC solver that can compute the following functions for all cogs:

- 32 x 32 unsigned multiply with 64-bit product
- 64 / 32 unsigned divide with 32-bit quotient and 32-bit remainder
- Square root of 64-bit value with 32-bit result
- 32-bit signed (X,Y) rotation around (0,0) by a 32-bit unsigned angle with 32-bit signed (X,Y) results
- 32-bit signed (X,Y) to 32-bit unsigned (length,angle) - cartesian to polar
- 32-bit unsigned integer to 5:27-bit logarithm
- 5:27-bit logarithm to 32-bit unsigned integer

When a cog issues a CORDIC instruction, it must wait for its hub slot, which is 0..(cogs-1) clocks away, in order to hand off the command to the CORDIC solver. Fifty-five clocks later, results will be available via the GETQX and GETQY instructions, which will wait for the results, in case they haven't arrived yet.

Because each cog's hub slot comes around every 1/2/4/8/16 clocks and the pipeline is 54 clocks long, it is possible to overlap CORDIC commands, where several commands are initially given to the CORDIC solver, and then results are read and another command is given, indefinitely, until, at the end, the trailing results are read. You must not have interrupts enabled during such a juggle, or enough clocks could be stolen by the interrupt service routine that one or more of your results could be overwritten before you can read them. If you ever attempt to read results when none are available and none are in progress, the QMT (CORDIC empty) event flag will be set.

MULTIPLY

To multiply two unsigned 32-bit numbers together, use the QMUL instruction:

```
        QMUL    D/#,S/#             - Multiply D by S
```

To get the results:

```
        GETQX   lower_long
        GETQY   upper_long
```

DIVIDE

For convenience, two different divide instructions exist, each with an optional SETQ prefix instruction which establishes a non-0 value for one 32-bit part of the 64-bit numerator:

```
        QDIV    D/#,S,#             - Divide {$00000000:D} by S
...or...
        SETQ    Q/#                 - Set top part of numerator
        QDIV    D/#,S,#             - Divide {Q:D} by S
...or...
        QFRAC   D/#,S,#             - Divide {D:$00000000} by S
...or...
        SETQ    Q/#                 - Set bottom part of numerator
        QFRAC   D/#,S,#             - Divide {D:Q} by S

To get the results:

        GETQX   quotient
        GETQY   remainder
```

SQUARE ROOT

To get the square root of a 64-bit integer:

```
        QSQRT   D/#,S,#             - Compute square root of {S:D}

To get the result:

        GETQX   root
```

(X,Y) ROTATION

The rotation function inputs three terms: 32-bit signed X and Y values, and an unsigned 32-bit angle, where $00000000..$FFFFFFFF = 0..359.9999999 degrees. The Y term, if non-zero, is supplied via an optional SETQ prefix instruction:

```
       SETQ    Q/#                - Set Y
       QROTATE D/#,S,#            - Rotate (D,Q) by S
```
...or...
```
       QROTATE D/#,S,#            - Rotate (D,$00000000) by S
```

Notice that in the second example, a polar-to-cartesian conversion is taking place.

To get the results:

```
       GETQX   X
       GETQY   Y
```

(X,Y) VECTORING

The vectoring function converts (X,Y) cartesian coordinates into (length,angle) polar coordinates:

```
       QVECTOR D/#,S,#            - (D,S) cartesian into (length,angle) polar
```

To get the results:

```
       GETQX   length
       GETQY   angle
```

LOGARITHM

To convert an unsigned 32-bit integer into a 5:27-bit logarithm, where the top 5 bits hold the whole part of the power-of-2 exponent and the bottom 27 bits hold the fractional part:

```
       QLOG    D/#                - Compute log base 2 of D
```

To get the result:

```
       GETQX   logarithm
```

EXPONENT

To convert a 5:27-bit logarithm into a 32-bit unsigned integer:

```
       QEXP    D/#                - Compute 2 to the power of D
```

To get the result:

```
       GETQX   integer
```

DACs

Each cog outputs four 8-bit DAC channels that can directly drive the DAC's of pins.

DAC0 can drive the DAC's of all pins numbered %XXXX00.
DAC1 can drive the DAC's of all pins numbered %XXXX01.
DAC2 can drive the DAC's of all pins numbered %XXXX10.
DAC3 can drive the DAC's of all pins numbered %XXXX11.

The background state of these four 8-bit channels can be established by SETDACS:

```
SETDACS D/#         - Write bytes 3/2/1/0 of D/# to DAC3/DAC2/DAC1/DAC0
```

The DAC values established by SETDACS will be constantly output, except at times when the streamer and/or colorspace converter override them.

STREAMER

Each cog has a streamer which can automatically output timed state sequences to pins and DACs. It can also periodically capture pin states to hub RAM and perform Goertzel computations from smart pins configured as ADC's.

There are five instructions directly associated with the streamer:

```
SETXFRQ D/#         - Set NCO frequency
XINIT   D/#,S/#     - Issue command immediately, zeroing phase
XZERO   D/#,S/#     - Issue command on final NCO rollover, zeroing phase
XCONT   D/#,S/#     - Issue command on final NCO rollover, continuing phase
GETXACC D           - Get Goertzel X into D and Y into next S, clear X and Y
```

The streamer uses a numerically-controlled oscillator (NCO) to time its operation. On every clock while the streamer is active, it adds a 32-bit frequency value into a 32-bit phase accumulator, while masking the MSB of the original phase. The NCO can be understood as such:

```
phase = (phase & $7FFF_FFFF) + frequency
```

The MSB of the resultant phase value indicates NCO rollover and is used as a trigger to advance the state of the streamer. This is true for every mode except DDS/Goertzel, in which case the streamer runs continuously.

The frequency of the streamer's NCO rollover is set by the 'SETXFRQ D/#' instruction, where D/# expresses a fractional 0-to-1 multiplier for the system clock, which value must be multiplied by $8000_0000. Here are some system clock multipliers and the D/# values that realize them:

```
1           $8000_0000  (default value on cog start)
1 / 2       $4000_0000
1 / 3       $2AAA_AAAB *
1 / 4       $2000_0000
1 / 5       $1999_999A *
1 / 6       $1555_5556 *
```

```
1 / 7         $1249_2493 *
1 / 8         $1000_0000
```

* For fractions with remainders, increment the D/# value in order to get desired initial rollover behavior.

The NCO frequency may also be set/changed via a 'SETQ D/#' instruction immediately preceding an XINIT/XZERO/XCONT instruction. When the streamer command executes, the new frequency will be set during the first clock of the command. If no SETQ is used before the instruction, the frequency will remain the same when the command executes.

The streamer may be activated by a command from an XINIT/XZERO/XCONT instruction. For these instructions, D/# expresses the streamer mode and duration, while S/# supplies various data, or is ignored, depending upon the mode expressed in D/#.

There is a single-level command buffer in the streamer, enabling you to give it two initial commands before it makes you wait for the first command to finish before accepting another. This command buffer enables you to coordinate streamer activity with smart pin activity. By executing an XINIT and then an XCONT, you get time during the XINIT command to instantiate a smart pin to perform some operation which will then correlate with the queued XCONT command. Think of tossing a ball up gently, so that you can then hit it with a bat.

For the XINIT/XZERO/XCONT instructions, D/#[31:16] conveys the command, while D/#[15:0] conveys the number of NCO rollovers that the command will be active for. S/# is used to select sub-modes for some commands:

```
 D/#[31:16]
 mode dacs pins base   S/#       description                 dac output
 ---- ---- ---- ----   ------    --------------------------  --------------------
%0000_dddd_xppp_pppx   config    DDS/Goertzel LUT            $xxxxxxxx

%0001_dddd_eppp_pppx   %00r00    1-bit RFBYTE, r=reorder     $000000xx, %aaaaaaaa
%0001_dddd_eppp_ppxx   %00r01    2-bit RFBYTE, r=reorder     $000000xx, %babababa
%0001_dddd_eppp_pxxx   %00r10    4-bit RFBYTE, r=reorder     $000000xx, %dcbadcba
%0001_dddd_eppp_xxxx   %00011    8-bit RFBYTE                $000000xx
%0001_dddd_eppp_xxxx   %01rgb    8-bit RFBYTE LUMA8          $RRGGBB00
%0001_dddd_eppp_xxxx   %10xxx    8-bit RFBYTE RGBI8          $RRGGBB00
%0001_dddd_eppp_xxxx   %11xxx    8-bit RFBYTE RGB8 (3:3:2)   $RRGGBB00

%0010_dddd_eppp_xxxx       %0    16-bit RFWORD               $0000xxxx
%0010_dddd_eppp_xxxx       %1    16-bit RFWORD RGB16 (5:6:5) $RRGGBB00

%0011_dddd_eppp_xxxx       %0    32-bit RFLONG               $xxxxxxxx
%0011_dddd_eppp_xxxx       %1    32-bit RFLONG RGB24 (8:8:8) $RRGGBB00

%0100_dddd_eppp_bbbb     %rxx    1-bit RFLONG LUT, r=reorder $xxxxxxxx
%0101_dddd_eppp_bbbb     %rxx    2-bit RFLONG LUT, r=reorder $xxxxxxxx
%0110_dddd_eppp_bbbb     %rxx    4-bit RFLONG LUT, r=reorder $xxxxxxxx
%0111_dddd_eppp_bbbb        -    8-bit RFLONG LUT            $xxxxxxxx

%1000_dddd_eppp_bbbb   <long>    1-bit immediate LUT         $xxxxxxxx
%1001_dddd_eppp_bbbb   <long>    2-bit immediate LUT         $xxxxxxxx
%1010_dddd_eppp_bbbb   <long>    4-bit immediate LUT         $xxxxxxxx
%1011_dddd_eppp_bbbb   <long>    8-bit immediate LUT         $xxxxxxxx

%1100_dddd_eppp_xxxx   <long>    32-bit immediate            $xxxxxxxx
```

```
%1101_xxxx_xppp_pppx     %r00    1-pin -> WFBYTE, r=reorder    <none>
%1101_xxxx_xppp_ppxx     %r01    2-pin -> WFBYTE, r=reorder    <none>
%1101_xxxx_xppp_pxxx     %r10    4-pin -> WFBYTE, r=reorder    <none>
%1101_xxxx_xppp_xxxx      %11    8-pin -> WFBYTE               <none>

%1110_xxxx_xppp_xxxx        -    16-pin -> WFWORD              <none>

%1111_xxxx_xppp_xxxx        -    32-pin -> WFLONG              <none>

(x = don't care)
```

Each of these modes requires some explanation, but there are some overlapping matters that can be covered first.

The 16-bit D[15:0] field expresses an initial counter value that will be decremented on each subsequent NCO rollover, with each rollover causing new streamer data to be output or input. When the counter equals 1 and the NCO is rolling over for the last time for the current command, a new command may be seamlessly dovetailed into by a buffered XZERO/XCONT instruction. If no XZERO/XCONT instruction is waiting, the counter goes to 0. When the counter reaches 0, or is set to 0, streamer operation stops and all streamer DAC overrides and streamer pin outputs cease.

By setting the count field to its maximal value of $FFFF, a streamer command will run perpetually.

XINIT (re)starts the streamer, no matter what state it is in. 'XINIT #0,#0' will always stop the streamer immediately. XSTOP (no operands) is an alias for 'XINIT #0,#0'.

XZERO and XCONT are used to maintain seamless streamer I/O, from command to command. They wait for the prior command's last clock cycle. If the streamer count has already run down to 0, there is no waiting. Also, if the prior command used $FFFF for its initial count, in which case the streamer is running perpetually without decrementing its counter, a new XZERO/XCONT command will only wait for the next NCO rollover, at which point the streamer will begin executing the new command.

XZERO clears out the phase accumulator when it executes. This clearing is desirable when, say, pixels are being output at 1/3 Fclk and and you don't want a 1-clock delay (glitch) every ~30 seconds, due to imperfect fractions like %5555_5555 = ~1/3. In such a case, it would be good to use XZERO to initiate the horizontal sync pulse, while using XCONT everywhere else. It may also be desirable to increment such frequency values by 1, so that the initial NCO rollover occurs on the Nth clock, and not on the Nth-1 clock.

XCONT is like XZERO, but does not affect the phase accumulator. XCONT is useful in cases where NCO frequency should be strictly maintained and streamer activity will ride along with it.

The streamer has four DAC output channels, X0, X1, X2 and X3, which can selectively override the four SETDACS values, on a per-DAC basis. The %dddd field selects which streamer DAC channels will override which SETDACS values. In the table below, "-" indicates no override and "!" indicates one's-complement:

	DAC				
dddd	3	2	1	0	description
0000	-	-	-	-	no streamer DAC output
0001	X0	X0	X0	X0	output X0 on all four DAC channels
0010	-	-	X0	X0	output X0 on DAC channels 1 and 0
0011	X0	X0	-	-	output X0 on DAC channels 3 and 2
0100	-	-	-	X0	output X0 on DAC channel 0

```
0101     -   -   X0  -      output X0 on DAC channel 1
0110     -   X0  -   -      output X0 on DAC channel 2
0111     X0  -   -   -      output X0 on DAC channel 3
1000    !X0  X0 !X0  X0     output X0 diff pairs on all four DAC channels
1001     -   -  !X0  X0     output X0 diff pairs on DAC channels 1 and 0
1010    !X0  X0  -   -      output X0 diff pairs on DAC channels 3 and 2
1011     X1  X0  X1  X0     output X1, X0 pairs on all four DAC channels
1100     -   -   X1  X0     output X1, X0 on DAC channels 1 and 0
1101     X1  X0  -   -      output X1, X0 on DAC channels 3 and 2
1110    !X1  X1 !X0  X0     output X1, X0 diff pairs on all four DAC channels
1111     X3  X2  X1  X0     output X3, X2, X1, X0 on all four DAC channels
```

The streamer always deals with 32-bit, or 4 byte, data. When outputting to DACs, these 4 bytes are assigned, in descending order, to X3, X2, X1 and X0.

All pure output modes (top nibble = %0001..%1100) can output to pins, as well as to DACs. The data output to pins are always 32 bits, shifted up by some multiple of 8 bits, and then OR'd with {OUTB, OUTA} to get the final 64 pin output states for the cog. By only having certain output bits potentially "1", you can use the streamer to create timed output streams on up to 32 pins. The %eppp field controls which pins are output to:

```
%eppp : 0xxx = disable pin output
        1000 = enable output on pins 31..0
        1001 = enable output on pins 39..8
        1010 = enable output on pins 47..16
        1011 = enable output on pins 55..24
        1100 = enable output on pins 63..32
        1101 = enable output on pins 7..0, 63..40
        1110 = enable output on pins 15..0, 63..48
        1111 = enable output on pins 23..0, 63..56
```

For the 1/2/4-bit RFBYTE submodes, extra %p bits below the %eppp field are use to select the bit, twit, or nibble within the byte selected by %eppp. For DAC output in these modes, the bit, twit, or nibble is repeated to 8, 4, or 2 times, in order to fill the lowest DAC channel X0. For example, a nibble value of %0111 will result in an X0 value of %01110111. This way, lowest bit/twit/nibble values are $00 and highest values are $FF, giving full range to the DAC.

In the case of the pure input modes (top nibble = %1101..%1111), the %ppp field selects which pins will be captured on streamer input cycles:

```
%ppp : 000 = pins 31..0
       001 = pins 39..8
       010 = pins 47..16
       011 = pins 55..24
       100 = pins 63..32
       101 = pins 7..0, 63..40
       110 = pins 15..0, 63..48
       111 = pins 23..0, 63..56
```

For the 1/2/4-bit WFBYTE submodes, extra %p bits below the %ppp field are use to select the bit, twit, or nibble within the byte selected by %ppp.

DDS/Goertzel LUT mode

This mode is unique in that it inputs and outputs on every clock in which the command is active. Its purpose is to perform direct digital synthesis (DDS) of signals up to the Nyquist limit of Fclk/2 and/or to perform simultaneous Goertzel analysis on incoming sigma-delta ADC bit streams from smart pins.

Goertzel analysis can be thought of as a single slice of a Fourier transform, where energy of a single frequency is measured amid potential noise. Goertzel analysis returns sine and cosine accumulations which can be converted into polar coordinates, yielding power and phase information. This mode uses the lookup RAM as a source of sine/cosine samples, such that bytes 3 and 2 must be unbiased signed sine and cosine values, and bytes 1 and 0 are biased (positive) sine and cosine values suitable for driving DACs. By incorporating DDS output with Goertzel input, many interactive real-world measurements can be made to determine things like time-of-flight and resonance.

The %pppppp field in D/# supplies a pin number whose input state is sampled on every clock. This pin should be configured for ADC mode so that the pin's input is a sigma-delta bit stream.

S/# supplies a 12-bit value which is used to select how much of the LUT will be used, what part of the LUT will be used, and what offset will be used:

S/#	Loop Size	NCO Bits	LUT Range
%000_TTTTTTTTT	512	30..22	%000000000..%111111111
%001_ATTTTTTTT	256	30..23	%A00000000..%A11111111
%010_AATTTTTTT	128	30..24	%AA0000000..%AA1111111
%011_AAATTTTTT	64	30..25	%AAA000000..%AAA111111
%100_AAAATTTTT	32	30..26	%AAAA00000..%AAAA11111
%101_AAAAATTTT	16	30..27	%AAAAA0000..%AAAAA1111
%110_AAAAAATTT	8	30..28	%AAAAAA000..%AAAAAA111
%111_AAAAAAATT	4	30..29	%AAAAAAA00..%AAAAAAA11

On each clock, the lookup RAM is read at the location bound by the %A bits, with the lower bits being the sum of the %T bits and NCO bits. The lookup RAM returns a 32-bit, or 4-byte, value. These four bytes, in descending order, become X3, X2, X1 and X0.

For the purpose of sine and cosine accumulation, the X3 (sine) and X2 (cosine) values will each be sign-extended to 32 bits and then added or subtracted into/from their respective accumulators, based on the state of the ADC input pin (0=add, 1=subtract).

After some number of complete NCO cycles, both accumulators can be simultaneously captured and cleared using the GETXACC instruction. GETXACC writes the cosine accumulation to D and places the sine accumulation into the next instruction's S value.

RFBYTE mode, 1/2/4-bit submodes

On the initial and every 8th/4th/2nd NCO rollover, a background RFBYTE is executed, returning 8 bits of data, which will be output 1, 2, or 4 bits at a time, on each NCO rollover.

When bit 2 of S/# is 0, bits/twits/nibbles will be shifted out LSB(s)-first from the RFBYTE data.

When bit 2 of S/# is 1, bits/twits/nibbles will be shifted out MSB(s)-first from the RFBYTE data.

It is necessary to do a RDFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to deliver data.

RFBYTE mode, 8-bit submodes

On each NCO rollover, a background RFBYTE is executed, returning 8 bits of data which will be output.

When bits 4..0 of S/# are %00011, bytes are read from hub (zero-extended to 32 bits) and output.

When bits 4..0 of S/# are %01rgb, bytes are read from hub, expanded from %IIIIIIII to 8:8:8:0 RGB, and output.

When bits 4..0 of S/# are %10xxx, bytes are read from hub, expanded from %RGBIIIII to 8:8:8:0 RGB, and output.

When bits 4..0 of S/# are %11xxx, bytes are read from hub, expanded from 3:3:2 RGB to 8:8:8:0 RGB, and output.

It is necessary to do a RDFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to deliver data.

RFWORD mode

On each NCO rollover, a background RFWORD is executed, returning 16 bits of data which will be output.

When bit 0 of S/# is 0, words are read from hub (zero-extended to 32 bits) and output.

When bit 0 of S/# is 1, words are read from hub, expanded from 5:6:5 RGB to 8:8:8:0 RGB, and output.

It is necessary to do a RDFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to deliver data.

RFLONG mode

On each NCO rollover, a background RFLONG is executed, returning 32 bits of data which will be output.

When bit 0 of S/# is 0, longs are read from hub and output.

When bit 0 of S/# is 1, longs are read from hub, AND'd with $FF_FF_FF_00 to make 8:8:8:0 RGB, and output.

It is necessary to do a RDFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to deliver data.

RFLONG LUT modes

A background RFLONG is executed initially and then again whenever more data is needed, in order to supply new 1/2/4/8-bit values on each NCO rollover, while shifting remaining RFLONG bits right. These 1/2/4/8-bit values are used as offset addresses in lookup RAM, while the %bbbb field of D/# furnishes the base address of %bbbb00000. The resultant 32 bits of data read from lookup RAM (at %bbbb00000 + 1/2/4/8-bit value) are output.

For the 1/2/4-bit modes, bit 2 of S/# is used to reorder bit fields within bytes of the initial RFLONG data. If bit 2 of S/# is 0, bits/twits/nibbles will be shifted out low-bit(s)-first from each byte. If bit 2 of S/# is 1, bits/twits/nibbles will be shifted out high-bit(s)-first from each byte.

It is necessary to do a RDFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to deliver data.

Immediate LUT modes

S/# provides 32 bits of data which supply 1/2/4/8-bit values, starting from the lowest bits, with remaining bits shifting right by 1/2/4/8 bits on each NCO rollover, while the top 1/2/4/8 bits stay in place. These 1/2/4/8-bit values are used as offset addresses in lookup RAM, while the %bbbb field of D/# furnishes the base address of %bbbb00000. The resultant 32 bits of data read from lookup RAM (at %bbbb00000 + 1/2/4/8-bit value) are output.

Immediate mode

S/# provides 32 bits of data which are directly output for the duration of the command.

WFBYTE mode, 1/2/4-bit submodes

On every 8th/4th/2nd NCO rollover, a background WFBYTE is executed to store the input pin bits/twits/nibbles that were captured on each NCO rollover, into hub.

Bit 2 of S/# is used to select bit field reordering within the WFBYTE data. If bit 2 of S/# is 0, bits/twits/nibbles will be right-shifted into the WFBYTE data. If bit 2 of S/# is 1, bits/twits/nibbles will be left-shifted into the WFBYTE data.

It is necessary to do a WRFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to receive data.

WFBYTE/WFWORD/WFLONG modes

On every NCO rollover, input pins are captured and a background WFBYTE/WFWORD/WFLONG is executed to store them into the hub.

It is necessary to do a WRFAST sometime beforehand, to ensure that the hub RAM FIFO is ready to receive data.

COLOR SPACE CONVERTER

Each cog has a color space converter which can perform ongoing matrix transformations and modulation of the cog's 8-bit DAC channels. The colorspace converter is intended primarily for baseband video modulation, but it can also be used as a general-purpose RF modulator.

The color space converter is configured via the following instructions:

```
SETCY   {#}D        - Set color space converter CY parameter to D[31:0]
SETCI   {#}D        - Set color space converter CI parameter to D[31:0]
SETCQ   {#}D        - Set color space converter CQ parameter to D[31:0]
SETCFRQ {#}D        - Set color space converter CFRQ parameter to D[31:0]
SETCMOD {#}D        - Set color space converter CMOD parameter to D[6:0]
```

It is intended that DAC3/DAC2/DAC1 serve as R/G/B channels. On each clock, new matrix and modulation calculations are performed through a pipeline. There is a group delay of five clocks from DAC-channel inputs to outputs when the color space converter is in use.

For the following signed multiply-accumulate computations, CMOD[4] determines whether the CY/CI/CQ terms will be sign-extended (CMOD[4] = 1) or zero-extended (CMOD[4] = 0). If zero-extended, using 128 for a CY/CI/CQ term will result in no attenuation of the related DAC term:

```
Y[7:0]      = (DAC3 * CY[31:24] + DAC2 * CY[23:16] + DAC1 * CY[15:8]) / 128
I[7:0]      = (DAC3 * CI[31:24] + DAC2 * CI[23:16] + DAC1 * CI[15:8]) / 128
Q[7:0]      = (DAC3 * CQ[31:24] + DAC2 * CQ[23:16] + DAC1 * CQ[15:8]) / 128
```

The modulator works by cumulatively subtracting CFRQ from PHS, in order to get a clockwise angle rotation in the upper bits of PHS. PHS[31:24] is then used to rotate the coordinate pair (I, Q). The rotated Q coordinate becomes IQ. Because a 5-stage CORDIC rotator is used to perform the rotation, IQ gets scaled by 1.646. When using the modulator, this scaling will need to be taken into account when computing your CI/CQ terms, in order to avoid IQ overflow:

```
PHS[31:0]   = PHS[31:0] - CFRQ[31:0]
IQ[7:0]     = Q of (I,Q) after being rotated by PHS and multiplied by 1.646
```

The formula for computing CFRQ for a desired modulation frequency is: (desired_frequency / clock_frequency) * $1_0000_0000. For example, if you wanted 3.579545 MHz and your clock frequency was 80 MHz, you would get (3.579545 / 80) * $1_0000_0000 = $0B74_5CFE, which you would set using the SETCFRQ instruction.

The preliminary output terms are computed as follows:

```
FY[7:0]     = CY[7:0] + (DAC0 & {8{CMOD[3]}}) + Y[7:0]     (VGA R / HDTV Y)
FI[7:0]     = CI[7:0] + (DAC0 & {8{CMOD[2]}}) + I[7:0]     (VGA G / HDTV Pb)
FQ[7:0]     = CQ[7:0] + (DAC0 & {8{CMOD[1]}}) + Q[7:0]     (VGA B / HDTV Pr)

FS[7:0]     = {8{DAC0[0] ^ CMOD[0]}}                       (VGA H-Sync)

FIQ[7:0]    = CQ[7:0] + IQ[7:0]                            (Chroma)

FYS[7:0]    = DAC0[1]    ?     8'b0                        (1x = Luma Sync)
            : DAC0[0]    ?     CI[7:0]                     (01 = Luma Blank/Burst)
                         :     CY[7:0] + Y[7:0]            (00 = Luma Visible)

FYC[7:0]    = FYS[7:0] + IQ[7:0]                                  (Composite
Luma+Chroma)
```

The final output terms are selected by CMOD[6:5]:

CMOD[6:5]	Mode	DAC3	DAC2	DAC1	DAC0
00	<off>	DAC3 (bypass)	DAC2 (bypass)	DAC1 (bypass)	DAC0 (bypass)
01	VGA (R-G-B) / HDTV (Y-Pb-Pr)	FY (R / Y)	FI (G / Pb)	FQ (B / Pr)	FS (H-Sync)
10	NTSC/PAL Composite + S-Video	FYC (Composite)	FYC (Composite)	FIQ (Chroma)	FYS (Luma)

11	NTSC/PAL Composite	FYC (Composite)	FYC (Composite)	FYC (Composite)	FYC (Composite)

HUB CONFIGURATION

The hub contains several global circuits which are configured using the HUBSET instruction. HUBSET uses a single D operand to both select the circuit to be configured and to provide the configuration data:

```
HUBSET  {#}D     - Configure global circuit selected by MSBs

%0000_xxxE_DDDD_DDMM_MMMM_MMMM_PPPP_CCSS     Set clock generator mode
%0001_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx     Hard reset, reboots chip
%0010_xxxx_xxxx_xxLW_DDDD_DDDD_DDDD_DDDD     Set write-protect and debug enables
%0100_xxxx_xxxx_xxxx_xxxx_xxxR_RLLT_TTTT     Set filter R to length L and tap T
%1DDD_DDDD_DDDD_DDDD_DDDD_DDDD_DDDD_DDDD     Seed Xoroshiro128** PRNG with D
```

Configuring the Clock Generator

The Prop2 can generate its system clock in several different ways.

There are two separate internal RC clock oscillators that can be used, a 20MHz+ and a ~20KHz. The 20MHz+ oscillator is designed to always run at least 20MHz, worst-case, in order to accommodate 2M baud serial loading during boot. The ~20KHz oscillator is intended for low-power operation.

The XI and XO pins can also be used for clocking, with XI being an input and XO being a crystal-feedback output for 10MHz-20MHz crystals. Internal loading caps can also be enabled on XI and XO for crystal impedance matching.

If the XI pin is used as a clock input or crystal oscillator input, its frequency can be modified through an internal phase-locked loop (PLL). The PLL divides the XI pin frequency from 1 to 64, then multiplies the resulting frequency from 1 to 1024 in the VCO. The VCO frequency can be used directly, or divided by 2, 4, 6, ...30, to get the final PLL clock frequency which can be used as the system clock.

The clock configuration setting is comprised of 25 bits. The four LSBs are all that are needed to switch among clock sources and select all but the PLL settings.

```
HUBSET  ##%0000_000E_DDDD_DDMM_MMMM_MMMM_PPPP_CCSS     'set clock mode
```

The tables below explain the various bit fields within the HUBSET operand:

PLL Setting	Value	Effect	Notes
%E	0/1	PLL off/on	XI input must be enabled by %CC. Allow 10ms for crystal+PLL to stabilize before switching over to PLL clock source.

%DDDDDD	0..63	1..64 division of XI pin frequency	This divided XI frequency feeds into the phase-frequency comparator's 'reference' input.
%MMMMMMMMMM	0..1023	1..1024 division of VCO frequency	This divided VCO frequency feeds into the phase-frequency comparator's 'feedback' input. This frequency division has the effect of *multiplying* the divided XI frequency (per %DDDDDD) inside the VCO. The VCO frequency should be kept within 100MHz to 400MHz.
%PPPP	0..14 15	2, 4, 6, ...30 division of VCO frequency VCO frequency	This divided VCO frequency is selectable as the system clock when SS = %11. The VCO output is selectable as the system clock when SS = %11.

%CC	XI status	XO status	XI / XO impedance	XI / XO loading caps
%00	ignored	float	Hi-Z	OFF
%01	input	600-ohm drive	1M-ohm	OFF
%10	input	600-ohm drive	1M-ohm	15pF per pin
%11	input	600-ohm drive	1M-ohm	30pF per pin

%SS	Clock Source	Notes
%11	PLL	CC != %00 and E=1, allow 10ms for crystal+PLL to stabilize before switching to PLL
%10	XI	CC != %00, allow 5ms for crystal to stabilize before switching to XI
%01	~20KHz	~20KHz, can be switched to at any time, low-power
%00	20MHz+	20MHz+, can be switched to at any time, used on boot-up.

PLL Example

The PLL's VCO is designed to run between 100MHz and 200MHz and should be kept within that range.

The VCO frequency will be (Freq(XI) / (%DDDDDD + 1)) * (%MMMMMMMMMM + 1).

The PLL output frequency will be Freq(VCO) if %PPPP = 15, else Freq(VCO) / (%PPPP + 1) / 2.

Let's say you have a 20MHz crystal attached to XI and XO and you want to run the Prop2 at 148.5MHz. You could divide the crystal by 40 (%DDDDDD = 39) to get a 500KHz reference, then multiply that by 297 (%MMMMMMMMMM = 296) in the VCO to get 148.5MHz. You would set %PPPP to %1111 to use the VCO output directly. The configuration value would be %1_100111_0100101000_1111_10_11. The last two 2-bit fields select 15pf crystal mode and the PLL. In order to realize this clock setting, though, it must be done over a few steps:

```
HUBSET  #0                                  'set 20MHz+ mode
HUBSET  ##%1_100111_0100101000_1111_10_00   'enable crystal+PLL, stay in 20MHz+ mode
```

```
WAITX   ##20_000_000/100                    'wait ~10ms for crystal+PLL to stabilize
HUBSET  ##%1_100111_0100101000_1111_10_11   'now switch to PLL running at 148.5MHz
```

The clock selector controlled by the %SS bits has a deglitching circuit which waits for a positive edge on the old clock source before disengaging, holding its output high, and then waiting for a positive edge on the new clock source before switching over to it. It is necessary to select mode %00 or %01 while waiting for the crystal and/or PLL to settle into operation, before switching over to either.

NOTE TO FPGA USERS: The only supported clock-setting values are $00 for 20MHz and $FF for 80MHz.

Rebooting the Chip

HUBSET can be used to reset and reboot the chip:

```
HUBSET  ##$1000_0000    'generate an internal reset pulse to reboot
```

Write-Protecting the Last 16KB of Hub RAM and Enabling Debug Interrupts

```
HUBSET  {#}D                    'set write-protect and enable debug interrupts

{#}D = %0010_xxxx_xxxx_xxLW_DDDD_DDDD_DDDD_DDDD

%L:  Lock W and D bit settings until next reset
         0 = establish W and D bit settings and allow subsequent modification
         1 = establish W and D bit settings and disallow subsequent modification

%W:  Write-protect last 16KB of hub RAM
         0 = Last 16KB of hub RAM is accessible at both its normal range and at
             $FC000..$FFFFF (default)
         1 = Last 16KB of hub RAM disappears from its normal range and is write-
             protected at $FC000..$FFFFF, except from within debug ISR's

%D:  Debug interrupt enables for cogs 15..0, respectively
         0 = Debug interrupt is disabled for cog n (default)
         1 = Debug interrupt is enabled for cog n
```

Examples:

```
HUBSET  ##$2000_0001    'enable debug interrupt for cog 0

HUBSET  ##$2001_FFFF    'enable debug interrupts for cogs 15..0
                        '..and write-protect the last 16KB of hub RAM

HUBSET  ##$2003_00FF    'enable debug interrupts for cogs 7..0
                        '..and write-protect the last 16KB of hub RAM
```

```
                                        '..and disallow subsequent changes to this scheme
```

See the DEBUG INTERRUPT section to learn how debug interrupts work.

Configuring the Digital Filters for Smart Pins

There are four global digital filter settings which can be used by each smart pin to low-pass filter its incoming pin states.

Each filter setting includes a filter length and a timing tap. The filter length is 2, 3, 5, or 8 flipflops, selected by values 0..3. The flipflops shift pin state data at the timing tap rate and must be unanimously high or low to change the filter output to high or low. The timing tap is one of the 32 bits of CT (the free-running 32-bit global counter), selected by values 0..31. Each time the selected tap transitions, the current pin state is shifted into the flipflops and if the flipflops are all in agreement, the filter output goes to that state. The filter will be reflected in the INA/INB bits if no smart pin mode is selected, or the filter states will be used by the smart pin mode as its inputs.

The D operand selects both the filter to configure and the data to configure it with:

```
HUBSET  ##$4000_0000 + Length<<5 + Tap      'set filt0
HUBSET  ##$4000_0080 + Length<<5 + Tap      'set filt1
HUBSET  ##$4000_0100 + Length<<5 + Tap      'set filt2
HUBSET  ##$4000_0180 + Length<<5 + Tap      'set filt3
```

"Length" is 0..3 for 2, 3, 5, or 8 flipflops.

"Tap" is 0..31 for every single clock, every 2nd clock, every 4th clock,... every 2,147,483,648th clock.

The filters are set to the following defaults on reset:

Filter	Length (flipflops)	Tap (clocks per sample)	Low-pass time (at 6.25ns/clock)
filt0	0 (2 flipflops)	0 (1:1)	12.5ns (6.25ns * 2 * 1)
filt1	1 (3 flipflops)	5 (32:1)	600ns (6.25ns * 3 * 32)
filt2	2 (5 flipflops)	19 (512K:1)	16.4ms (6.25ns * 5 * 512K)
filt3	3 (8 flipflops)	22 (4M:1)	210ms (6.25ns * 8 * 4M)

Seeding the Xoroshiro128 PRNG**

To seed 32 bits of state data into the 128-bit PRNG, use HUBSET with the MSB of D set. This will write {1'b1, D[30:0]} into 32 bits of the PRNG, affecting 1/4th of its total state. The 1'b1 bit ensures that the overall state will not go to zero. Because the PRNG's 128 state bits rotate, shift, and XOR against each other, they are thoroughly spread around within a few clocks, so

seeding from a fixed set of 32 bits should not pose a limitation on seeding quality.

After reset, the boot ROM uses HUBSET to seed the Xoroshiro128** PRNG fifty times, each time with 31 bits of thermal noise gleaned from pin 63 while in ADC calibration mode. This establishes a very random seed which the PRNG iterates from, thereafter. There is no need to do this again, but here is how you would do it if 'x' contained a seed value:

```
        SETB    x,#31   'set the MSB of x to make a PRNG seed command
        HUBSET  x       'seed 32 bits of the Xoroshiro128** state
```

The Xoroshiro128** PRNG iterates on every clock, generating 64 fresh bits which get spread among all cogs and smart pins. Each cog receives a unique set of 32 different bits, in a scrambled arrangement with some bits inverted, from the 64-bit pool. Each smart pin receives a similarly-unique set of 8 different bits. Cogs can sample these bits using the GETRND instruction and directly apply them using the BITRND and DRVRND instructions. Smart pins utilize their 8 bits as noise sources for DAC dithering and noise output.

LOCKS - Semaphore Bits

The hub contains 16 semaphore bits, called LOCKs, which can be captured and released by cogs at runtime for the purpose of permitting one cog at a time to gain exclusive access to some resource. The hub can allocate LOCKs and recycle them, as needed by an application.

On reset, all LOCKs are unallocated by the hub and in a 'released' state. LOCKs can be used directly, without the hub allocating them, if there is some application-wide agreement on what each LOCK is to be used for. Otherwise, LOCKNEW and LOCKRET can dynamically allocate and recycle LOCKs:

```
        LOCKNEW D       WC      'Get a LOCK from the hub, D=LOCK, C=1 if none available
        LOCKRET {#}D            'Return LOCK D to the hub for recycling
```

LOCKTRY is used to attempt capture of a LOCK. When a LOCK is in a released state, the first cog to execute a LOCKTRY on that LOCK will capture it and become its owner. No other cog will be able to become the owner of that LOCK until either the owner cog releases it, or the owner cog stops or restarts (via COGSTOP or COGINIT), which forces the release of any LOCKs that cog owned:

```
        LOCKTRY {#}D    WC      'Try to capture LOCK D. C=1 if successful
```

Because LOCK arbitration is performed by the hub in a round-robin fashion, any cog waiting in a loop to capture a LOCK will get its fair turn:

```
.try    LOCKTRY {#}D    WC      'Keep trying to capture LOCK until successful
if_nc   JMP     #.try
```

LOCKREL is used to release a captured LOCK. Only a cog that owns a LOCK can release it, making LOCKREL benign in cases where the cog does not own the lock:

```
        LOCKREL  {#}D            'Release LOCK D if owned by this cog
```

LOCKREL can also be used to investigate the current status of a LOCK:

```
LOCKREL  D       WC      'Release LOCK D if owned, D=current/last owner, C=captured
```

What a LOCK represents is completely up to the application using it. LOCKs are just a means of allowing one cog at a time the exclusive status of 'owner'. All participant cogs must agree on a LOCK's number and its purpose for a LOCK to be useful.

SMART PINS

Each I/O pin has a ‘smart pin’ circuit which, when enabled, performs some autonomous function on the pin. Smart pins free the cogs from needing to micro-manage many I/O operations by providing high-bandwidth concurrent hardware functions which cogs could not perform as well on their own by manipulating I/O pins via instructions.

Normally, an I/O pin’s output enable is controlled by its DIR bit and its output state is controlled by its OUT bit, while the IN bit returns the pin's read state. In smart pin modes, the DIR bit is used as an active-low reset signal to the smart pin circuitry, while the output enable state is controlled by a configuration bit. In some modes, the smart pin takes over driving the output state, in which case the OUT bit gets ignored. The IN bit serves as a flag to indicate to the cog(s) that the smart pin has completed some function or an event has occurred, and acknowledgment is perhaps needed.

Smart pins have four 32-bit registers inside of them:

mode	- smart pin mode, as well as low-level I/O pin mode (write-only)
X	- mode-specific parameter (write-only)
Y	- mode-specific parameter (write-only)
Z	- mode-specific result (read-only)

These four registers are written and read via the following 2-clock instructions, in which S/# is used to select the pin number (0..63) and D/# is the 32-bit data conduit:

```
WRPIN   D/#,S/#         - Set smart pin S/# mode to D/#, ack pin
WXPIN   D/#,S/#         - Set smart pin S/# parameter X to D/#, ack pin
WYPIN   D/#,S/#         - Set smart pin S/# parameter Y to D/#, ack pin
RDPIN   D,S/# {WC}      - Get smart pin S/# result Z into D, flag into C, ack pin
RQPIN   D,S/# {WC}      - Get smart pin S/# result Z into D, flag into C, don't ack
AKPIN   S/#             - Acknowledge pin S/#
```

Each cog has a 34-bit bus to each smart pin for write data and acknowledgment signalling. Each smart pin OR’s all incoming 34-bit buses from the cogs in the same way DIR and OUT bits are OR’d before going to the pins. Therefore, if you intend to have multiple cogs execute WRPIN / WXPIN / WYPIN / RDPIN / AKPIN instructions on the same smart pin, you must be sure that they do so at different times, in order to avoid clobbering each other’s bus data. Any number of cogs can read a smart pin simultaneously, without bus conflict, though, by using RQPIN (‘read quiet’), since it does not utilize the 34-bit cog-to-smart-pin bus for acknowledgement signalling, like RDPIN does.

Each smart pin has an outgoing 33-bit bus which conveys its Z result and a special flag. RDPIN and RQPIN are used to multiplex and read these buses, so that a pin's Z result is read into D and its special flag can be read into C. C will be either a mode-related flag or the MSB of the Z result.

For the WRPIN instruction, which establishes both the low-level and smart-pin configuration for each I/O pin, the D operand is composed as:

```
D/# = %AAAA_BBBB_FFF_PPPPPPPPPPPPP_TT_MMMMM_0

 %AAAA:  'A' input selector
             0xxx = true (default)
             1xxx = inverted
             x000 = this pin's read state (default)
             x001 = relative +1 pin's read state
             x010 = relative +2 pin's read state
             x011 = relative +3 pin's read state
             x100 = this pin's OUT bit from cogs
             x101 = relative -3 pin's read state
             x110 = relative -2 pin's read state
             x111 = relative -1 pin's read state

 %BBBB:  'B' input selector
             0xxx = true (default)
             1xxx = inverted
             x000 = this pin's read state (default)
             x001 = relative +1 pin's read state
             x010 = relative +2 pin's read state
             x011 = relative +3 pin's read state
             x100 = this pin's OUT bit from cogs
             x101 = relative -3 pin's read state
             x110 = relative -2 pin's read state
             x111 = relative -1 pin's read state

  %FFF:  'A' and 'B' input logic/filtering (after 'A' and 'B' input selectors)
             000 = A, B (default)
             001 = A AND B, B
             010 = A OR  B, B
             011 = A XOR B, B
             100 = A, B, both filtered using global filt0 settings
             101 = A, B, both filtered using global filt1 settings
             110 = A, B, both filtered using global filt2 settings
             111 = A, B, both filtered using global filt3 settings

             The resultant 'A' will drive the IN signal in non-smart-pin modes.

 %P..P:  low-level pin control (needs final silicon to fully operate)
```

```
            %0000CIOHHHLLL = digital mode (default = %0000000000000)
                     %C: 1 = clocked I/O (extra clock for IN and OUT)
                     %I: 1 = invert IN output
                     %O: 1 = invert OUT input
                  %HHH: 000 = drive high, other = float when driven high
                  %LLL: 000 = drive low,  other = float when driven low
            %101xxDDDDDDDD = DAC mode, %DDDDDDDD: DAC output level

   %TT:  pin DIR/OUT control (default = %00)

         for odd pins,  'OTHER' = NOT lower pin's output state (diff source)
         for even pins, 'OTHER' = unique pseudo-random bit (noise source)
         for all pins,  'SMART' = smart pin output which overrides OUT/OTHER
         'DAC_MODE' is enabled when P[12:10] = %101
         'BIT_DAC' overrides P[7:0] with $00 during 'low' output in DAC_MODE

         for smart pin mode off (%MMMMM = %00000):

             DIR enables output

             for non-DAC_MODE:
                 0x = OUT drives output
                 1x = OTHER drives output
             for DAC_MODE:
                 00 = OUT enables ADC, P[7:0] sets DAC level
                 01 = OUT enables ADC, P[3:0] selects cog DAC channel
                 10 = OUT drives BIT_DAC
                 11 = OTHER drives BIT_DAC

         for all smart pin modes (%MMMMM > %00000):
             x0 = output disabled, regardless of DIR
             x1 = output enabled, regardless of DIR

         for DAC smart pin modes (%MMMMM = %00001..%00011):
             0x = OUT enables ADC in DAC_MODE, P[7:0] overriden
             1x = OTHER enables ADC in DAC_MODE, P[7:0] overriden

         for non-DAC smart pin modes (%MMMMM = %00100..%11111):
             0x = SMART/OUT drives output or BIT_DAC if DAC_MODE
             1x = SMART/OTHER drives output or BIT_DAC if DAC_MODE

%MMMMM:  00000   = smart pin off (default)
         00001   = long repository              (P[12:10] != %101)
         00010   = long repository              (P[12:10] != %101)
         00011   = long repository              (P[12:10] != %101)
```

```
00001   = DAC noise                     (P[12:10]  = %101)
00010   = DAC 16-bit dither, noise      (P[12:10]  = %101)
00011   = DAC 16-bit dither, PWM        (P[12:10]  = %101)
00100*  = pulse/cycle output
00101*  = transition output
00110*  = NCO frequency
00111*  = NCO duty
01000*  = PWM triangle
01001*  = PWM sawtooth
01010*  = PWM switch-mode power supply, V and I feedback
01011   = periodic/continuous: A-B quadrature encoder
01100   = periodic/continuous: inc on A-rise & B-high
01101   = periodic/continuous: inc on A-rise & B-high / dec on A-rise & B-low
01110   = periodic/continuous: inc on A-rise {/ dec on B-rise}
01111   = periodic/continuous: inc on A-high {/ dec on B-high}
10000   = time A-states
10001   = time A-highs
10010   = time X A-highs/rises/edges -or- timeout a-/high/rise/edge
10011   = for X periods, count time
10100   = for X periods, count states
10101   = for periods in X+ clocks, count time
10110   = for periods in X+ clocks, count states
10111   = for periods in X+ clocks, count periods
11000*  = USB host, low-speed          (even/odd pin pair = DM/DP)
11001*  = USB host, high-speed         (even/odd pin pair = DM/DP)
11010*  = USB device, low-speed        (even/odd pin pair = DM/DP)
11011*  = USB device, high-speed       (even/odd pin pair = DM/DP)
11100*  = sync serial transmit         (A-data, B-clock)
11101   = sync serial receive          (A-data, B-clock)
11110*  = async serial transmit        (baudrate)
11111   = async serial receive         (baudrate)

* OUT signal overridden
```

When a mode-related event occurs in a smart pin, it raises its IN signal to alert the cog(s) that new data is ready, new data can be loaded, or some process has finished. A cog acknowledges a smart pin whenever it does a WRPIN, WXPIN, WYPIN, RDPIN or AKPIN on it. This causes the smart pin to lower its IN signal so that it can be raised again on the next event. Note that since the RQPIN instruction (read quiet) does not do an acknowledge, it can be used by any number of cogs, concurrently, to read a pin without bus conflict.

After WRPIN/WXPIN/WYPIN/RDPIN/AKPIN, it will take two clocks for IN to drop, before it can be polled again:

```
WRPIN/WXPIN/WYPIN/RDPIN/AKPIN   'acknowledge smart pin, releases IN from high
NOP                             'elapse 2 clocks (or more)
TESTP   pin     WC              'IN can now be polled again
```

Smart pins should be configured while their DIR signal is low, holding them in reset. During that time, WRPIN/WXPIN/WYPIN can be used to establish the mode and related parameters. Once configured, DIR can be raised high and the smart pin will begin operating. After that, depending on the mode, you may feed it new data via WXPIN/WYPIN or retrieve results using RDPIN/RQPIN. These activities are usually coordinated with the IN signal going high.

To return a pin to normal mode, do a 'WRPIN #0,pin'.

SMART PIN MODES

%00001..%00011 and not DAC_MODE = long repository

This mode turns the smart pin into a long repository, where WXPIN writes the long and RDPIN/RQPIN can read the long.

Upon each WXPIN, IN is raised.

%00001 and DAC_MODE = DAC noise

This mode overrides P[7:0] to feed the pin's 8-bit DAC unique pseudo-random data on every clock. P[12:10] must be set to %101 to configure the low-level pin for DAC output.

X[15:0] can be set to a sample period, in clock cycles, in case you want to mark time with IN raising at each period completion. If a sample period is not wanted, set X[15:0] to zero (65,536 clocks), in order to maximize the unused sample period, thereby reducing switching power.

RDPIN/RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period.

During reset (DIR=0), IN is low.

%00010 and DAC_MODE = DAC 16-bit with pseudo-random dither

This mode overrides P[7:0] to feed the pin's 8-bit DAC with pseudo-randomly-dithered data on every clock. P[12:10] must be set to %101 to configure the low-level pin for DAC output.

X[15:0] establishes the sample period in clock cycles.

Y[15:0] establishes the DAC output value which gets captured at each sample period and used for its duration.

On completion of each sample period, Y[15:0] is captured for the next output value and IN is raised. Therefore, you would coordinate updating Y[15:0] with IN going high.

Pseudo-random dithering does not require any kind of fixed period, as it randomly dithers the 8-bit DAC between adjacent levels, in order to achieve 16-bit DAC output, averaged over time. So, if you would like to be able to update the output value at any time and have it take immediate effect, set X[15:0] to one (IN will stay high).

If OUT is high, the ADC will be enabled and RDPIN/RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period. This can be used to measure loading on the DAC pin.

During reset (DIR=0), IN is low and Y[15:0] is captured.

%00011 and DAC_MODE = DAC 16-bit with PWM dither

This mode overrides P[7:0] to feed the pin's 8-bit DAC with PWM-dithered data on every clock. P[12:10] must be set to %101 to configure the low-level pin for DAC output.

X[15:0] establishes the sample period in clock cycles. The sample period must be a multiple of 256 (X[7:0]=0), so that an integral number of 256 steps are afforded the PWM, which dithers the DAC between adjacent 8-bit levels.

Y[15:0] establishes the DAC output value which gets captured at each sample period and used for its duration.

On completion of each sample period, Y[15:0] is captured for the next output value and IN is raised. Therefore, you would coordinate updating Y[15:0] with IN going high.

PWM dithering will give better dynamic range than pseudo-random dithering, since a maximum of only two transitions occur for every 256 clocks. This means, though, that a frequency of Fclock/256 will be present in the output at -48dB.

If OUT is high, the ADC will be enabled and RDPIN/RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period. This can be used to measure loading on the DAC pin.

During reset (DIR=0), IN is low and Y[15:0] is captured.

%00100 = pulse/cycle output

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

X[31:16] establishes a value to which the base period counter will be compared to on each clock cycle, as it counts from X[15:0] down to 1, before starting over at X[15:0] if decremented Y > 0. On each clock, if the base period counter > X[31:16] and Y > 0, the output will be high (else low).

Whenever Y[31:0] is written with a non-zero value, the pin will begin outputting a high pulse or cycles, starting at the next base period.

Some examples:

If X[31:16] is set to 0, the output will be high for the duration of Y > 0.

If X[15:0] is set to 3 and X[31:16] is set to 2, the output will be 0-0-1 (repeat) for the duration of Y > 0.

IN will be raised when the pulse or cycles complete, with the pin reverting to low output.

During reset (DIR=0), IN is low, the output is low, and Y is set to zero.

%00101 = transition output

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

Whenever Y[31:0] is written with a non-zero value, the pin will begin toggling for Y transitions at each base period, starting at the next base period.

IN will be raised when the transitions complete, with the pin remaining in its current output state.

During reset (DIR=0), IN is low, the output is low, and Y is set to zero.

%00110 = NCO frequency

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

Y[31:0] will be added into Z[31:0] at each base period.

The pin output will reflect Z[31].

IN will be raised whenever Z overflows.

During reset (DIR=0), IN is low, the output is low, and Z is set to zero.

%00111 = NCO duty

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

Y[31:0] will be added into Z[31:0] at each base period.

The pin output will reflect Z overflow.

IN will be raised whenever Z overflows.

During reset (DIR=0), IN is low, the output is low, and Z is set to zero.

%01000 = PWM triangle

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

X[31:16] establishes a PWM frame period in terms of base periods.

Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.

A counter, updating at each base period, counts from the frame period down to one, then from one back up to the frame period. Then, Y[15:0] is captured, IN is raised, and the process repeats.

At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. Therefore, a zero will always output a low and the frame period value will always output a high.

During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

%01001 = PWM sawtooth

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

X[31:16] establishes a PWM frame period in terms of base periods.

Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.

A counter, updating at each base period, counts from one up to the frame period. Then, Y[15:0] is captured, IN is raised, and the process repeats.

At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. Therefore, a zero will always output a low and the frame period value will always output a high.

During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

%01010 = PWM switch-mode power supply with voltage and current feedback

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

X[31:16] establishes a PWM frame period in terms of base periods.

Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.

A counter, updating at each base period, counts from one up to the frame period. Then, the ‘A’ input is sampled at each base period until it reads low. After ‘A’ reads low, Y[15:0] is captured, IN is raised, and the process repeats.

At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. If, at any time during the cycle, the ‘B’ input goes high, the output will be low for the rest of that cycle.

Due to the nature of switch-mode power supplies, it may be appropriate to just set Y[15:0] once and let it repeat indefinitely.

During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

%01011 = A/B-input quadrature encoder

X[31:0] establishes a measurement period in clock cycles.

If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit quadrature step count can always be read via RDPIN/RQPIN.

If a non-zero value is used for the period, quadrature steps will be counted for that many clock cycles and then the result will be placed in Z while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it. This way, all quadrature steps get counted across measurements. At the end of each period, IN will be raised and RDPIN/RQPIN can be used to retrieve the last 32-bit measurement.

It may be useful to configure both ‘A’ and ‘B’ smart pins to quadrature mode, with one being continuous (X=0) for absolute position tracking and the other being periodic (x<>0) for velocity measurement.

The quadrature encoder can be “zeroed” by pulsing DIR low at any time. There is no need to do another WXPIN.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

%01100 = Count A-input positive edges when B-input is high

X[31:0] establishes a measurement period in clock cycles.

If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN/RQPIN.

If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN/RQPIN can be used to retrieve the 32-bit measurement.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1).

%01101 = Accumulate A-input positive edges with B-input supplying increment (B=1) or decrement (B=0)

X[31:0] establishes a measurement period in clock cycles.

If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN/RQPIN.

If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN/RQPIN can be used to retrieve the 32-bit measurement.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

%01110 AND !Y[0] = Count A-input positive edges
%01110 AND Y[0] = Increment on A-input positive edge and decrement on B-input positive edge

X[31:0] establishes a measurement period in clock cycles. Y[0] establishes whether to just count A-input positive edges (=0), or to increment on A-input positive edge and decrement on B-input positive edge (=1).

If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN/RQPIN.

If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN/RQPIN can be used to retrieve the 32-bit measurement.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

%01111 AND !Y[0] = Count A-input highs
%01111 AND Y[0] = Increment on A-input high and decrement on B-input high

X[31:0] establishes a measurement period in clock cycles. Y[0] establishes whether to just count A-input highs (=0), or to increment on A-input high and decrement on B-input high (=1).

If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN/RQPIN.

If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN/RQPIN can be used to retrieve the 32-bit measurement.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

%10000 = Time A-input states

Continuous states are counted in clock cycles.

Upon each state change, the prior state is placed in the C-flag buffer, the prior state's duration count is placed in Z, and IN is raised. RDPIN/RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.

If states change faster than the cog is able to retrieve measurements, the measurements will effectively be lost, as old ones will be overwritten with new ones. This may be gotten around by using two smart pins to time highs, with one pin inverting its 'A' input. Then, you could capture both states, as long as the sum of the states' durations didn't exceed the cog's ability retrieve both results. This would help in cases where one of the states was very short in duration, but the other wasn't.

During reset (DIR=0), IN is low and Z is set to $00000001.

%10001 = Time A-input high states

Continuous high states are counted in clock cycles.

Upon each high-to-low transition, the previous high duration count is placed in Z, and IN is raised. RDPIN/RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.

During reset (DIR=0), IN is low and Z is set to $00000001.

%10010 AND !Y[2] = Time X A-input highs/rises/edges

Time is measured until X A-input highs/rises/edges are accumulated.

X[31:0] establishes how many A-input highs/rises/edges are to be accumulated.

Y[1:0] establishes A-input high/rise/edge sensitivity:

%00 = A-input high
%01 = A-input rise
%1x = A-input edge

Time is measured in clock cycles until X highs/rises/edges are accumulated from the A-input. The measurement is then placed in Z, and IN is raised. RDPIN/RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.

During reset (DIR=0), IN is low and Z is set to $00000001.

%10010 AND Y[2] = Timeout on X clocks of missing A-input high/rise/edge

If no A-input high/rise/edge occurs within X clocks, IN is raised, a new timeout period of X clocks begins, and Z maintains a running count of how many clocks have elapsed since the last A-input high/rise/edge. Z will be limited to $80000000 and can

be read any time via RDPIN/RQPIN.

If an A-input high/rise/edge does occur within X clocks, a new timeout period of X clocks begins and Z is reset to $00000001.

X[31:0] establishes how many clocks before a timeout due to no A-input high/rise/edge occurring.

Y[1:0] establishes A-input high/rise/edge sensitivity:

%00 = A-input high
%01 = A-input rise
%1x = A-input edge

During reset (DIR=0), IN is low and Z is set to $00000001.

%10011 = For X periods, count time
%10100 = For X periods, count states

X[31:0] establishes how many A-input rise/edge to B-input rise/edge periods are to be measured.

Y[1:0] establishes A-input and B-input rise/edge sensitivity:

%00 = A-input rise to B-input rise
%01 = A-input rise to B-input edge
%10 = A-input edge to B-input rise
%11 = A-input edge to B-input edge

Note: The B-input can be set to the same pin as the A-input for single-pin cycle measurement.

Clock cycles or A-input trigger states are counted from each A-input rise/edge to each B-input rise/edge for X periods. If the A-input rise/edge is ever coincident with the B-input rise/edge at the end of the period, the start of the next period is registered. Upon completion of X periods, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN/RQPIN can then be used to retrieve the completed measurement. Z will be limited to $80000000.

The first mode is intended to be used as an oversampling period measurement, while the second mode is a complementary duty measurement.

During reset (DIR=0), IN is low and Z is set to $00000000.

%10101 = For periods in X+ clock cycles, count time
%10110 = For periods in X+ clock cycles, count states
%10111 = For periods in X+ clock cycles, count periods

X[31:0] establishes the minimum number of clock cycles to track periods for. Periods are A-input rise/edge to B-input rise/edge.

Y[1:0] establishes A-input and B-input rise/edge sensitivity:

%00 = A-input rise to B-input rise
%01 = A-input rise to B-input edge

%10 = A-input edge to B-input rise
%11 = A-input edge to B-input edge

Note: The B-input can be set to the same pin as the A-input for single-pin cycle measurement.

A measurement is taken across some number of A-input rise/edge to B-input rise/edge periods, until X clock cycles elapse and then any period in progress completes. If the A-input rise/edge is ever coincident with the B-input rise/edge at the end of the period, the start of the next period is registered. Upon completion, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN/RQPIN can then be used to retrieve the completed measurement. Z will be limited to $80000000.

The first mode accumulates time within each period, for an oversampled period measurement.

The second mode accumulates A-input trigger states within each period, for an oversampled duty measurement.

The third mode counts the periods.

Knowing how many clock cycles some number of complete periods took, and what the duty was, affords a very time-efficient and precise means of determining frequency and duty cycle. At least two of these measurements must be made concurrently to get useful results.

During reset (DIR=0), IN is low and Z is set to $00000000.

%11000 = USB host, low-speed
%11001 = USB host, full-speed
%11010 = USB device, low-speed
%11011 = USB device, full-speed

This mode requires that two adjacent pins be configured together to form a USB pair, whose OUTs will be overridden to control their output states. These pins must be an even/odd pair, having only the LSB of their pin numbers different. For example: pins 0 and 1, pins 2 and 3, pins 4 and 5, etc., can form USB pairs. They can be configured via WRPIN with identical D data of %1_110xx_0. Using D data of %0_110xx_0 will disable output drive and effectively create a USB ‘sniffer’. A new WRPIN can be done to effect such a change without resetting the smart pin. **NOTE: in the current FPGA, there are no built-in 1.5k and 15k resistors, which the final silicon smart pins will contain, so it is up to you to insert these yourself on the DP and DM lines.**

The upper (odd) pin is the DP pin. This pin’s IN is raised whenever the output buffer empties, signalling that a new output byte can be written via WYPIN to the lower (even) pin. No WXPIN/WYPIN instructions are used for this pin.

The lower (even) pin is the DM pin. This pin’s IN is raised whenever a change of status occurs in the receiver, at which point a RDPIN/RQPIN can be used on this pin to read the 16-bit status word. WXPIN is used on this pin to set the NCO baud rate.

These DP/DM electrical designations can actually be switched by swapping low-speed and full-speed modes, due to USB’s mirrored line signalling.

To start USB, clear the DIR bits of the intended two pins and configure them each via WRPIN. Use WXPIN on the lower pin to

set the baud rate, which is a 16-bit fraction of the system clock. For example, if the main clock is 80MHz and you want a 12MHz baud rate (full-speed), use 12,000,000 / 80,000,000 * $10000 = 9830. Then, set the pins' DIR bits. You are now ready to read the receiver status via RDPIN/RQPIN and set output states and send packets via WYPIN, both on the lower pin.

To affect the line states or send a packet, use WYPIN on the lower pin. Here are its D values:

0 = output IDLE	- default state, float pins, except possible resistor(s) to 3.3V or GND
1 = output SE0	- drive both DP and DM low
2 = output K	- drive K state onto DP and DM (opposite)
3 = output J	- drive J state onto DP and DM (opposite), like IDLE, but driven
4 = output EOP	- output end-of-packet: SE0, SE0, J, then IDLE
$80 = SOP	- output start-of-packet, then bytes, automatic EOP when buffer runs out

To send a packet, first do a WYPIN #$80,lowerpin'. Then, after each IN rise on the upper pin, do a 'WYPIN byte,lowerpin' to buffer the next byte. The transmitter will automatically send an EOP when you stop giving it bytes. To keep the output buffer from overflowing, you should always verify that the upper pin's IN was raised after each WYPIN, before issuing another WYPIN, even if you are just setting a state. The reason for this is that all output activity is timed to the baud generator and even state changes must wait for the next bit period before being implemented, at which time the output buffer empties.

There are separate state machines for transmitting and receiving. Only the baud generator is common between them. The transmitter was just described above. Below, the receiver is detailed. Note that the receiver receives not just input from another host/device, but all local output, as well.

At any time, a RDPIN/RQPIN can be executed on the lower pin to read the current 16-bit status of the receiver, with the error flag going into C. The lower pin's IN will be raised whenever a change occurs in the receiver's status. This will necessitate A WRPIN/WXPIN/WYPIN/RDPIN/AKPIN before IN can be raised again, to alert of the next change in status. The receiver's status bits are as follows:

[31:16]	<unused>		- $0000
[15:8]	byte		- last byte received
[7]	byte toggle		- cleared on SOP, toggled on each byte received
[6]	error		- cleared on SOP, set on bit-unstuff error, EOP SE0 > 3 bits, or SE1
[5]	EOP in		- cleared on SOP or 7+ bits of J or K, set on EOP
[4]	SOP in		- cleared on EOP or 7+ bits of J or K, set on SOP
[3]	SE1 in	(illegal)	- cleared on !SE1, set on 1+ bits of SE1
[2]	SE0 in	(RESET)	- cleared on !SE0, set on 1+ bits of SE0
[1]	K in	(RESUME)	- cleared on !K, set on 7+ bits of K
[0]	J in	(IDLE)	- cleared on !J, set on 7+ bits of J

The result of a RDPIN/RQPIN can be bit-tested for events of interest. It can also be shifted right by 8 bits to LSB-justify the last byte received and get the byte toggle bit into C, in order to determine if you have a new byte. Assume that 'flag' is initially zero:

```
SHR     D,#8    WC      'get byte into D, get toggle bit into C
CMPX    flag,#1 WZ      'compare toggle bit to flag, new byte if Z
```

```
IF_Z   XOR     flag,#1          'if new byte, toggle flag
IF_Z   <use byte>               'if new byte, do something with it
```

%11100 = synchronous serial transmit

This mode overrides OUT to control the pin output state.

Words of 1 to 32 bits are shifted out on the pin, LSB first, with each new bit being output two internal clock cycles after registering a positive edge on the B input. For negative-edge clocking, the B input may be inverted by setting B[3] in WRPIN's D value.

WXPIN is used to configure the update mode and word length.

X[5] selects the update mode:

> X[5] = 0 sets continuous mode, where a first word is written via WYPIN during reset (DIR=0) to prime the shifter. Then, after reset (DIR=1), the second word is buffered via WYPIN and continuous clocking is started. Upon shifting each word, the buffered data written via WYPIN is advanced into the shifter and IN is raised, indicating that a new output word can be buffered via WYPIN. This mode allows steady data transmission with a continuous clock, as long as the WYPIN's after each IN-rise occur before the current word transmission is complete.
>
> X[5] = 1 sets start-stop mode, where the current output word can always be updated via WYPIN before the first clock, flowing right through the buffer into the shifter. Any WYPIN issued after the first clock will be buffered and loaded into the shifter after the last clock of the current output word, at which time it could be changed again via WYPIN. This mode is useful for setting up the output word before a stream of clocks are issued to shift it out.

X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.

WYPIN is used to load the output words. The words first go into a single-stage buffer before being advanced to the shifter for output. Each time the buffer is advanced into the shifter, IN is raised, indicating that a new output word can be written via WYPIN. During reset, the buffer flows straight into the shifter.

If you intend to send MSB-first data, you must first shift and then reverse it. For example, if you had a byte in D that you wanted to send MSB-first, you would do a 'SHL D,#32-8' and then a 'REV D'.

During reset (DIR=0) the output is held low. Upon release of reset, the output will reflect the LSB of the output word written by any WYPIN during reset.

%11101 = synchronous serial receive

Words of 1 to 32 bits are shifted in by sampling the A input around the positive edge of the B input. For negative-edge clocking, the B input may be inverted by setting B[3] in WRPIN's D value.

WXPIN is used to configure the sampling and word length.

X[5] selects the A input sample position relative to the B input edge:

X[5] = 0 selects the A input sample just before the B input edge was registered. This requires no hold time on the part of the sender.

X[5] = 1 selects the sample coincident with the B edge being registered. This is useful where transmitted data remains steady after the B edge for a brief time. In the synchronous serial transmit mode, the data is steady for two internal clocks after the B edge was registered, so employing this complementary feature would enable the fastest data transmission when receiving from another smart pin in synchronous serial transmit mode.

X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.

When a word is received, IN is raised and the data can then be read via RDPIN/RQPIN. The data read will be MSB-justified.

If you received LSB-first data, it will require right-shifting, unless the word size was 32 bits. For a word size of 8 bits, you would need to do a 'SHR D,#32-8' to get the data LSB-justified.

If you received MSB-first data, it will need to be reversed and possibly masked, unless the word size was 32 bits. For example, if you received a 9-bit word, you would do 'REV D' + 'TRIML D,#8' to get the data LSB-justified.

%11110 = asynchronous serial transmit

This mode overrides OUT to control the pin output state.

Words from 1 to 32 bits are serially transmitted on the pin at a programmable baud rate, beginning with a low "start" bit and ending with a high "stop" bit.

WXPIN is used to configure the baud rate and word length.

X[31:16] establishes the number of clocks in a bit period, and in case X[31:26] is zero, X[15:10] establishes the number of fractional clocks in a bit period. The X bit period value can be simply computed as: (clocks * $1_0000) & $FFFFFC00. For example, 7.5 clocks would be $00078000, and 33.33 clocks would be $00215400.

X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.

WYPIN is used to load the output words. The words first go into a single-stage buffer before being advanced to a shifter for output. This buffering mechanism makes it possible to keep the shifter constantly busy, so that gapless transmissions can be achieved. Any time a word is advanced from the buffer to the shifter, IN is raised, indicating that a new word can be loaded.

Here is the internal state sequence:

1. Wait for an output word to be buffered via WYPIN, then set the 'buffer-full' and 'busy' flags.
2. Move the word into the shifter, clear the 'buffer-full' flag, and raise IN.
3. Output a low for one bit period (the START bit).
4. Output the LSB of the shifter for one bit period, shift right, and repeat until all data bits are sent.
5. Output a high for one bit period (the STOP bit).
6. If the 'buffer-full' flag is set due to an intervening WYPIN, loop to (2). Otherwise, clear the 'busy' flag and loop to (1).

RDPIN/RQPIN with WC always returns the ‘busy’ flag into C. This is useful for knowing when a transmission has completed.

During reset (DIR=0) the output is held high.

%11111 = asynchronous serial receive

Words from 1 to 32 bits are serially received on the A input at a programmable baud rate.

WXPIN is used to configure the baud rate and word length.

X[31:16] establishes the number of clocks in a bit period, and in case X[31:26] is zero, X[15:10] establishes the number of fractional clocks in a bit period. The X bit period value can be simply computed as: (clocks * $1_0000) & $FFFFFC00. For example, 7.5 clocks would be $00078000, and 33.33 clocks would be $00215400.

X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.

Here is the internal state sequence:

1. Wait for the A input to go high (idle state).
2. Wait for the A input to go low (START bit edge).
3. Delay for half a bit period.
4. If the A input is no longer low, loop to (2).
5. Delay for one bit period.
6. Right-shift the A input into the shifter and delay for one bit period, repeat until all data bits are received.
7. Capture the shifter into the Z register and raise IN.
8. Loop to (1).

RDPIN/RQPIN is used to read the received word. The word must be shifted right by 32 minus the word size. For example, to LSB-justify an 8-bit word received, you would do a ‘SHR D,#32-8’.

BOOT PROCESS (needs more editing)

Boot Pattern Set By Resistors	P61	P60	P59
Serial window of 60s, default.	none	none	none
Serial window of 60s, overrides SPI and SD.	ignored	ignored	pull-up
Serial window of 100ms, then SPI flash. If SPI flash fails then serial window of 60s.	pull-up	ignored	none
SPI flash only (fast boot), no serial window. If SPI flash fails then shutdown.	pull-up	ignored	pull-down
SD card with serial window on failure. If SD card fails then serial window of 60s.	no pull-up	pull-up (built into SD card)	none
SD card only, no serial window.	no pull-up	pull-up	pull-down

If SD card fails then shutdown.		(built into SD card)	

Boot Serial	**P63 (input)**	**P62 (output)**
Serial	RX	TX

Boot Memory	**P61 (output)**	**P60 (output)**	**P59 (output)**	**P58 (input)**
SPI flash	CSn	CLK	DI	DO
SD card	CLK	CSn	DI	DO

After a hardware reset, cog 0 loads and executes a booter program from an internal ROM. The booter program (ROM_Booter.spin2) performs the following steps:

1) If an external pull-up resistor is sensed on P61 (SPI_CS), then attempt to boot from SPI:
    a) Load the first 1024 bytes (256 longs) from SPI into the hub starting at $00000.
    b) Compute the 32-bit sum of the 256 longs.
    c) If the sum is "Prop" ($706F7250):
        i) Copy the first 256 longs from hub into cog registers $000..$0FF.
        ii) If an external pull-up resistor is sensed on P60 (SPI_CK):
            (1) Execute ‘JMP #$000’ to run the SPI program. Done.
        iii) Begin waiting for serial command(s) on P63 (RX_PIN).
        iv) If 100ms elapsed and no command begun:
            (1) Execute ‘JMP #$000’ to run the SPI program. Done.
        v) If a program successfully loads serially within 60 seconds:
            (1) Execute ‘COGINIT #0,#0’ to relaunch cog 0 from $00000. Done.
        vi) Execute ‘JMP #$000’ to run the SPI program. Done.
2) Wait for serial command(s) on P63 (RX_PIN):
    a) If a program successfully loads serially within 60 seconds:
        i) Execute ‘COGINIT #0,#0’ to relaunch cog 0 from $00000. Done.
    b) Slow clock to 20KHz and stop cog 0. Done.

SERIAL LOADING PROTOCOL

The built-in serial loader allows Propeller 2 chips to be loaded via 8-N-1 asynchronous serial into P63, where START=low and STOP=high, at any rate the sender uses, between 9,600 baud and 2,000,000 baud.

The loader automatically adapts to the sender’s baud rate from every “>” character ($3E) it receives. It is necessary to initially send “> “ ($3E, $20) before the first command, and then use “>” characters periodically throughout your data to keep the baud rate tightly calibrated to the internal RC oscillator that the loader uses during boot ROM execution. Received “>” characters are not passed to the command parser, so they can be placed anywhere.

The loader’s response messages are sent back serially over P62 at the same baud rate that the sender is using. P62 is

normally driven continuously during the serial protocol, but will go into open-drain mode when either the INA or INB mask of a command is non-0 (masking is explained below).

Unless preempted by a program in a SPI memory chip with a pull-up resistor on P60 (SPI_CK), the serial loader becomes active within 15ms of reset being released.

Between command keywords and data, whitespace is required. The following characters, in any contiguous combination, constitute a single whitespace:

```
$09    TAB
$0A    LF
$0D    CR
$20    SP
$3D    "="    (may be present in Base64 data)
```

There are four commands which the sender can issue:

1) Request Propeller type:

```
Prop_Chk <INAmask> <INAdata> <INBmask> <INBdata>
```

2) Change clock setting:

```
Prop_Clk <INAmask> <INAdata> <INBmask> <INBdata> <HUBSETclocksetting>
```

3) Load and execute hex data, with and without sum checking:

```
Prop_Hex <INAmask> <INAdata> <INBmask> <INBdata> <hexdatabytes> ?
Prop_Hex <INAmask> <INAdata> <INBmask> <INBdata> <hexdatabytes> ~
```

4) Load and execute Base64 data, with and without sum checking:

```
Prop_Txt <INAmask> <INAdata> <INBmask> <INBdata> <base64chrs> ?
Prop_Txt <INAmask> <INAdata> <INBmask> <INBdata> <base64chrs> ~
```

Each command keyword is followed by four 32-bit hex values which allow selection of certain chips by their INA and INB states. If you wanted to talk to any and all chips that are connected, you would use zeroes for these values. In case multiple chips are being loaded from the same serial line, you would probably want to differentiate each download by unique INA and INB mask and data values. When the serial loader receives data and mask values which do not match its own INA and INB ports, it waits for another command. Because the command keywords all contain an underscore ("_"), they cannot be mistaken by intervening data belonging to a command destined for another chip, while a new command is being waited for.

If, at any time, a character is received which does not comport with expectations (i.e. an "x" is received when hex digits are expected), the loader aborts the current command and waits for a new command.

Prop_Chk

The Prop_Chk command returns CR+LF+"Prop_Ver"+SP+VerChr+CR+LF. VerChr is "A.."Z" and indicates the version of Propeller chip. A version "A" chip would respond as follows:

Sender: **"> Prop_Chk 0 0 0 0"+CR**

Loader: **CR+LF+"Prop_Ver A"+CR+LF**

Prop_Clk

The Prop_Clk command is used to switch the chip's clock source, as if a SETCLK instruction were being executed. Upon receiving the command, the loader immediately echos a "." character and then switches the clock over a 5ms period. The sender should allow 10ms after receipt of the "." before sending a new "> " ($3E, $20) followed by another command. To switch the clock to 80MHz:

Sender: **"> Prop_Clk 0 0 0 0 FF"+CR**

Loader: **"."**

Sender: Waits ~10 ms, then sends new command preceded by **"> "**

Prop_Hex

The Prop_Hex command is used to load byte data into the hub, starting at $00000, and then execute them. Hex bytes must be separated by whitespaces. Only the bottom 8 bits of hex values are used as data.

If the command is terminated with a "~" character, the loader will do a 'COGINIT #0,#0' to relaunch cog 0 (currently running the booter program) with the new program starting at $00000.

If the command is terminated with a "?" character, the loader will send either a "." character to signify that the embedded checksum was correct, in which case it will run the program as "~" would have. Or, it will send a "!" character to signify that the checksum was incorrect, after which it will wait for a new command.

To demonstrate hex loading, consider this small program:

```
DAT     ORG
        not     dirb                'all outputs
.lp     not     outb                'toggle states (blinks leds on Prop123 boards)
        waitx   ##20_000_000/4      'wait ¼ second
        jmp     #.lp                'loop
```

It assembles to:

```
00000- FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD
```

Here is how you would run this program from the serial loader:

Sender: **"> Prop_Hex 0 0 0 0 FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD ~"**

In the case of our assembled program, there are 5 little-endian longs which sum to $E6CE9A2C. To generate an embedded checksum long, you would compute $706F7250 ("Prop") minus the sum $E6CE9A2C, which results in $89A0D824. Those four bytes could be appended to the data as follows. Note that it doesn't matter where your embedded checksum long is placed,

only that it be long-aligned within your data:

Sender: **"> Prop_Hex 0 0 0 0 FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD 24 D8 A0 89 ?"**

Loader: **"."**

It's a good idea to start each hex data line with a ">" character, to keep the baud rate tightly calibrated.

Prop_Txt

The Prop_Txt command is like Prop_Hex, but with one difference: Instead of hex bytes separated by whitespaces, it takes in Base64 data, which are text characters that convey six bits, each, and get assembled into bytes as they are received. This format is 2.25x denser than hex, and so minimizes transmission size and time.

These are the characters that make up the Base64 alphabet:

```
"A".."Z"      = $00..$19
"A".."z"      = $1A..$33
"0".."9"      = $34..$3D
"+"           = $3E
"/"           = $3F
```

Whitespaces are ignored among Base64 characters.

To load and run the program used in the Prop_Hex example:

Sender: **"> Prop_Txt 0 0 0 0 +/cj9v37I/YlJoD/H4Bm/fD/n/0 ~"**

To add the embedded checksum:

Sender: **"> Prop_Txt 0 0 0 0 +/cj9v37I/YlJoD/H4Bm/fD/n/0k2KCJ ?"**

Loader: **"."**

It's a good idea to start each Base64 data line with a ">" character, to keep the baud rate tightly calibrated.

SUMMARY

It is possible to uniquely load many Propeller chips from the same serial signal by giving them each a different INA/INB signature and not connecting SPI memory chips or SD cards to P61..P58.

To try out the serial loader, just open a terminal program on your PC with the Propeller 2 connected and type: "> Prop_Chk 0 0 0 0"+CR. You can also cut and paste those Prop_Hex and Prop_Txt example lines to load the blinker program. A simple Propeller 2 development tool needs no special serial signalling, just simple text output that needn't worry about PC/Mac/Unix new-line differences, whitespace conventions, or generating non-standard characters.

<END>

```
wire [15:0][31:0] booti = {

            // cold boot rom - only cog0 on startup
            32'b1111_1100100_010_000000000_111111000,           // 0 =  wrfast  #0,ptra            (begin write at $FC000)
            32'b1111_1100110_110_000000001_111111001,           // 1 =  rep     #1,ptrb            (write $4000 bytes)
            32'b1111_1101011_000_000000000_000010101,           // 2 =  wfbyte  0                  (write rom byte to hub ram)
            32'b1111_1100100_010_000000000_111111000,           // 3 =  wrfast  #0,ptra            (finish write)
            32'b1111_1100111_010_000000000_111111000,           // 4 =  coginit #0,ptra            (restart at $FC000)
            32'b1111_1100111_010_000000000_111111000,           // 5 =  (unused, same as 4)
            32'b1111_1100111_010_000000000_111111000,           // 6 =  (unused, same as 4)
            32'b1111_1100111_010_000000000_111111000,           // 7 =  (unused, same as 4)

            // warm boot rom - via coginit
            32'b1111_0110000_001_111111100_000000000,           // 0 =  mov     outa,#0            (clear port shadow registers)
            32'b1111_0110000_001_111111101_000000000,           // 1 =  mov     outb,#0
            32'b1111_0110000_001_111111110_111111000,           // 2 =  mov     ina,#$1F8          (point ina/ijmp0 to cog's initial int0 handler)
            3'b111, !hubs, 28'b1101011_001_111110111_000101000, // 3 =  setq    #$1F7              (if !hubs, load $1F8 longs from ptrb)
            3'b111, !hubs, 28'b1011000_001_000000000_110000000, // 4 =  rdlong  0,ptrb
            21'b1111_1101011_000_1111110, !hubs, 10'b1_000101100, // 5 =  jmp     dirb/ptrb          (if !hubs, jump to $000 (dirb=0), else ptrb)
            21'b1111_1101011_000_1111110, !hubs, 10'b1_000101100, // 6 =  (unused, same as 5)
            21'b1111_1101011_000_1111110, !hubs, 10'b1_000101100  // 7 =  (unused, same as 5)
            };

wire [7:0][31:0] debugi = {

            // debug rom - executes in $001F8..$001FF
            32'b1111_1101011_001_000001111_000101000,           // 0 =  setq    #$0F               (ready to save registers $000..$00F)
            32'b1111_1100011_000_000000000_111111000,           // 1 =  wrlong  0,ptra             (ptra reads %1111_1111_1ccc_c000_0000)
            32'b1111_1101011_001_000001111_000101000,           // 2 =  setq    #$0F               (ready to load program into $000..$00F)
            32'b1111_1011000_000_000000000_111111001,           // 3 =  rdlong  0,ptrb             (ptrb reads %1111_1111_1ccc_c100_0000)
            32'b1111_1101100_000_000000000_000000000,           // 4 =  jmp     #0                 (jump to loaded program)
            32'b1111_1101011_001_000001111_000101000,           // 5 =  setq    #$0F               (ready to restore registers $000..$00F)
            32'b1111_1011000_000_000000000_111111000,           // 6 =  rdlong  0,ptra             (ptra reads %1111_1111_1ccc_c000_0000)
            32'b1111_1011001_110_111111111_111111111            // 7 =  reti0                      ('calld inb,inb wcz')
            };
```

Standard Device Specification

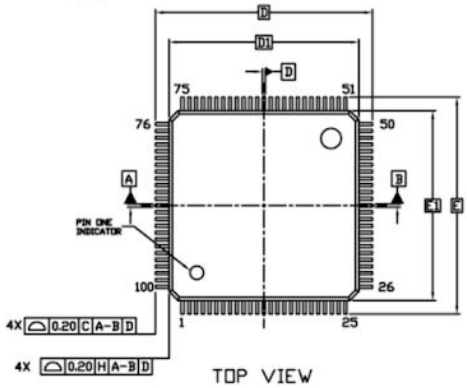
Package Drawing

MECHANICAL CASE OUTLINE
PACKAGE DIMENSIONS

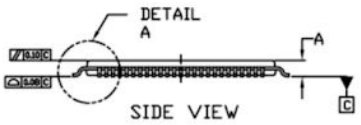
ON Semiconductor®

TQFP100 14x14, 0.5P
CASE 932BR
ISSUE O

DATE 03 JUL 2018



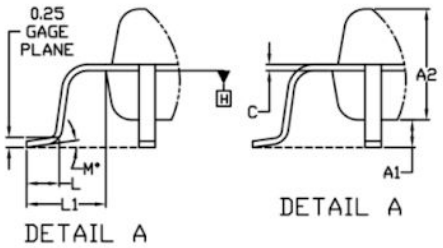
TOP VIEW



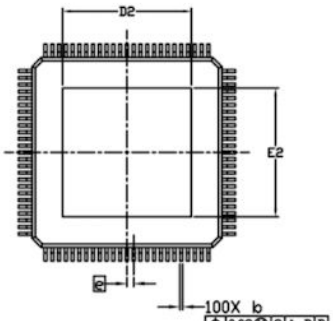
SIDE VIEW

NOTES:
1. DIMENSIONING AND TOLERANCING PER ASME Y14.5M, 1994.
2. CONTROLLING DIMENSION: MILLIMETERS
3. DIMENSION b DOES NOT INCLUDE DAMBAR PROTRUSION. DAMBAR PROTRUSION SHALL BE 0.08 MAX. AT MMC. DAMBAR CANNOT BE LOCATED ON THE LOWER RADIUS OF THE FOOT.
4. DIMENSIONS D1 AND E1 DO NOT INCLUDE MOLD FLASH, PROTRUSIONS, OR GATE BURRS. MOLD FLASH, PROTRUSIONS, OR GATE BURRS SHALL NOT EXCEED 0.25 PER SIDE. DIMENSIONS D1 AND E1 ARE MAXIMUM PLASTIC BODY SIZE INCLUDING MOLD MISMATCH.
5. THE TOP PACKAGE BODY SIZE IS SMALLER THAN THE BOTTOM PACKAGE SIZE AND TOP PACKAGE WILL NOT OVERHANG THE BOTTOM.
6. DATUM PLANE H IS LOCATED AT THE BOTTOM MOLD PARTING LINE COINCIDENT WITH WHERE THE LEAD EXITS THE BODY.
7. DIMENSIONS D1 AND E1 TO BE DETERMINED AT DATUM PLANE H.
8. DATUMS A-B AND D ARE DETERMINED AT DATUM PLANE H.
9. A1 IS DEFINED AS THE VERTICAL DISTANCE FROM THE SEATING PLANE TO THE LOWEST POINT ON THE PACKAGE BODY.
10. DIMENSIONS D AND E TO BE DETERMINED AT DATUM PLANE C.
11. EXPOSED PAD SIZE IS AFFECTED BY MOLD FLASH AND MOLD FLASH IS CONTROLLED BY FVIOFINAL VISUAL INSPECTION SPEC.

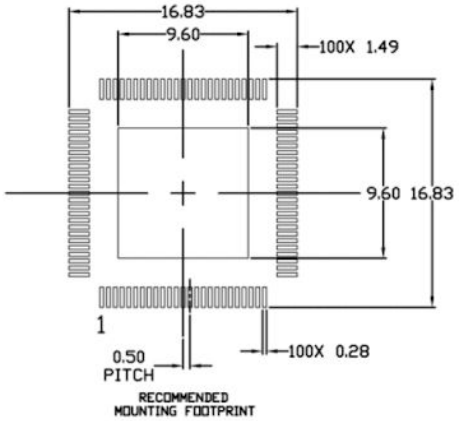
DIM	MILLIMETERS		
	MIN.	NOM.	MAX.
A	----	----	1.20
A1	0.05	----	0.15
A2	0.95	1.00	1.05
b	0.17	0.22	0.27
c		0.20 REF	
D	15.80	16.00	16.20
D1	13.80	14.00	14.20
D2		9.50 REF	
E	15.80	16.00	16.20
E1	13.80	14.00	14.20
E2		9.50 REF	
e		0.50 BSC	
L	0.45	0.60	0.75
L1		1.00 REF	
M	0°	----	7°



DETAIL A

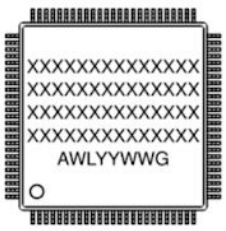


BOTTOM VIEW



RECOMMENDED
MOUNTING FOOTPRINT

GENERIC MARKING DIAGRAM*



XXX = Specific Device Code
A = Assembly Location
WL = Wafer Lot
YY = Year
WW = Work Week
G = Pb−Free Package

*This information is generic. Please refer to device data sheet for actual part marking. Pb−Free indicator, "G" or microdot " ▪", may or may not be present. Some products may not follow the Generic Marking.

DOCUMENT NUMBER:	98AON94348G	Electronic versions are uncontrolled except when accessed directly from the Document Repository. Printed versions are uncontrolled except when stamped "CONTROLLED COPY" in red.
DESCRIPTION:	TQFP100 14X14, 0.5P	PAGE 1 OF 1