



HELLO Propeller P2 EXAMPLES

User Notes

[Abstract](#)

Spin2 Example Programs

WRD william robert drury Rev08

bob_drury@hotmail.com

Foreward

The notes are a summation of various documents produced either by Parallax or Forum Topic submissions (thanks) while the Propeller II was being developed.

Sections 1.0-11.0 are Hardware product related. Sections 12.0-17.0 are programming related. The Appendix are either details of Propeller function or general support information.

The "Hello Propeller P2 Examples.zip" document is to be used in conjunction with the Propeller examples that are independent Spin2\PASM files provided in the "Hello Propeller II Examples.zip" file.

If there are any suggestions for clarification and improvements (or outright mistakes). Please forward this too: bob_drury@hotmail.com Your input would be appreciated.

Propeller II has smart pins allowing any pin to be either digital In, digital Out, analog In or analog Out. This feature sets the propeller apart from most microprocessors.

The smart pins also have built in logic allowing functions to run independent of the processor(s) and to request servicing.

The 8 independent cores with the same clock, can run separately or co-operatively with "OR" digital busing makes a unique hardware configuration.

Propeller firmware has custom "debug" routine which facilitates learning and debugging programs which is another unique feature from other microprocessors.

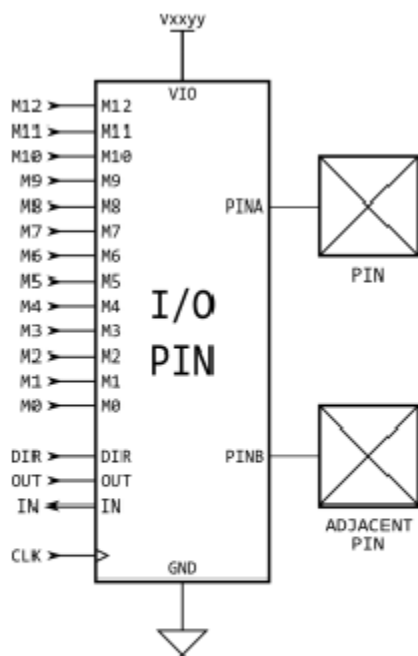
Propeller assembly language allows self modifying operands, again this unique feature allows for implementing pointers. Normally self modifying code is considered risky in the programming world.

The Propeller 2 features a pseudo-random number generator (PRNG) based on the Xoroshiro128** algorithm.

The Propeller 2 (Hub) contains a 54-stage pipelined CORDIC solver (Coordinate Rotation Digital Computer) useful for 32 bit arithmetic and Cartesian to Polar conversions.

1.1) Smart Pin Schematic

Every I/O pin features versatile digital and analog capabilities as well as autonomous state machine functions that would otherwise require processor time to perform. The combination provides adept functionality for application design, increasing the Propeller 2 potential beyond what multi-core architecture alone provides. There are 24 low-level 'pin' modes and 34 high-level 'smart' modes. Pin Modes Each I/O pin has 13 low-level pin mode configuration bits which determine the operation of its 3.3 V circuit.



P0..P63

(64 Instances)

The pin mode is set using the WRPIN instruction, where the 13 %MMMMMMMMMMMMMM bits within the instruction's D operand go directly to these bits. Note though that in some smart pin modes, these bits are partially overwritten to set things like DAC values.

Note: Maybe should read 32 instances

Note: Upon startup or reset, all I/O pins default to input (high impedance), meaning each cog's direction registers are initialized to zero. Each cog's output registers are initialized to zero as well, but this low (ground) state is not reflected on the pin until the pin is set to the output direction (via the direction register).

Pins to avoid for standard configuration:

- SP58: MISO (connection to SPI Flash Data Out pin or Micro SD MISO pin)
- P59: MOSI (connection to SPI Flash Data In pin or Micro SD MOSI pin)
- P60: CLK / CS (connection to SPI Flash CLK pin or Micro SD CS pin)
- P61: CS / CLK (connection to SPI Flash CS pin or Micro SD CLK pin)
- P62: Serial Tx (connection to host's Serial Rx)
- P63: Serial Rx (connection to host's Serial Tx)

1.2) Smart Pin WRPIN, WXPIN and WYPIN

Smart Modes Each I/O pin has built-in 'smart pin' circuitry which, when enabled, performs an autonomous function on the pin. Smart pins free the cogs from the need to micromanage many I/O operations by providing high-bandwidth concurrent hardware functions that cogs could otherwise not perform as well through I/O pin manipulating instructions.

In normal operation, an I/O pin's output enable is controlled by its DIR bit, its output state is controlled by its OUT bit, and its IN bit returns the pin's read state.

With smart pin mode enabled, its 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 circuit takes over driving the output state, in which case the OUT bit gets ignored. Its 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.

To configure a smart pin, first set its DIR bit to low (holding it in reset) then use WRPIN, WXPIN, and WYPIN 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; explained later. Note that while a smart pin is configured, the %TT bits (of the WRPIN instruction's D operand) will govern the pin's output enable, regardless of the DIR state.

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

Smart Pin Registers	
32-bit Register	Purpose
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 **PASM 2-clock instructions**, in which S/# is used to select the pin number (0..63) and D/# is the 32-bit data conduit:

Note: **S/#** indicates a literal **9-bit pin number (0..63)** or a symbol such as LED_pin, you defined earlier.
D/# is the 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 pin

AKPIN S/# - Acknowledge pin S/#

The format of the D (pin setup) operand value is:

D = %AAAA_BBBB_FFF_MMMMMMMMMMMMMM_TT_SSSS_0

- A = PINA input selector
- B = PINB input selector
- F = PINA and PINB input logic/filtering (after PINA and PINB input selectors)
- M = pin mode
- T = pin DIR/OUT control (default = %00)
- S = smart mode

Each smart pin has a 34-bit input bus and a 33-bit output bus that connect it to the cogs. To configure and control smart pins, each cog writes data and acknowledgement signals to the smart pin input bus. Each smart pin **OR's** all incoming **34-bit** buses from the collective of 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.

Reading a smart pin with RDPIN can cause the same conflict; however, any number of cogs can read a smart pin simultaneously without bus conflict by using RQPIN ('read quiet'), since it does not utilize the smart pin input 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.

Any number of cogs can read a smart pin simultaneously, without bus conflict, by using RQPIN ('read quiet'), since it does not utilize the 34-bit cog-to-smart-pin bus for acknowledge signalling, like RDPIN does.

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 can test for this signal via the TESTP instruction and can acknowledge a smart pin by executing a WRPIN, WXPIN, WYPIN, RDPIN, or AKPIN instruction for it. This acknowledgement causes the smart pin to lower its IN signal so that it can be raised again on the next event. After a WRPIN/WXPIN/WYPIN/RDPIN/AKPIN, it takes two clocks for IN to drop, before it can be polled again. A smart pin can be reset at any time, without the need to reconfigure it, by clearing and then setting its DIR bit. To return a pin to normal mode, do a 'WRPIN #0,pin'

WRPIN instruction writes 32-bit data, D/#, to the Mode register for I/O pin identified by the S/# value or symbol. Note: The WRPIN instruction sets two logic modes for each Smart Pin. The following tables describe the data fields in the WRPIN instruction. Most likely you will refer often to this table as you study the Smart-Pin modes. Each Smart-Pin mode requires 32 bits that define how pins and internal circuits will function. To make operations easier to understand, we break the 32-bit value into six sections. The LSB always equals 0.

D/# = %AAAA_BBBB_FFF_PPPPPPPPPPPP_TT_MMMMMM_0

You might ask, Why would a Smart Pin need to get information from a nearby pin?

This capability comes in handy when you want to monitor an input stream to calculate a serial-input bit rate, or to test an input with a different cog to obtain debug or diagnostic information. Some mode examples that follow use A and B signals for data and a clock, two encoder inputs, an input and a logic control, and so on.

1.2.1) PinA or PinB Input Selector

PINA or PINB Input Selector	
%AAAA %BBBB	Selection
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

1.2.2) PinA or PinB Logic\Filtering

PINA and PINB Logic/Filtering	
%FFF	Logic/Filter
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.

1.2.3) M Pin Modes

13-bit M PAD_IO Mode field are described by this table

PAD_IO Modes

M[12:0]	Legend	Input	PinA Output	CIOHHHLLL	OE	DAC	ADC	ADC Mode	Comp
0000_CIOHHHLLL		PinA Logic	OUT	CIOHHHLLL	DIR	0	0		0
0001_CIOHHHLLL	<u>C</u> IN/OUT	PinA Logic	Input	CIOHHHLLL	DIR	0	0		0
0010_CIOHHHLLL	0 Live	PinB Logic	Input	CIOHHHLLL	DIR	0	0		0
0011_CIOHHHLLL	1 Clocked	PinA Schmitt	OUT	CIOHHHLLL	DIR	0	0		0
0100_CIOHHHLLL		PinA Schmitt	Input	CIOHHHLLL	DIR	0	0		0
0101_CIOHHHLLL	<u>I</u> IN	PinB Schmitt	Input	CIOHHHLLL	DIR	0	0		0
0110_CIOHHHLLL	0 True	PinA > PinB	OUT	CIOHHHLLL	DIR	0	0		A>B
0111_CIOHHHLLL	1 Not	PinA > PinB	Input	CIOHHHLLL	DIR	0	0		A>B
100000_OHHHLLL	<u>O</u> Output	ADC, GIO 1x	OUT	10OHHHLLL	DIR	0	1	000	0
100001_OHHHLLL	0 True	ADC, VIO 1x	OUT	10OHHHLLL	DIR	0	1	001	0
100010_OHHHLLL	1 Not	ADC, float	OUT	10OHHHLLL	DIR	0	1	010	0
100011_OHHHLLL	<u>IOH</u> Drive	ADC, PinA 1x	OUT	10OHHHLLL	DIR	0	1	011	0
100100_OHHHLLL	000 Fast	ADC, PinA 3.16x	OUT	10OHHHLLL	DIR	0	1	100	0
100101_OHHHLLL	001 1.5kΩ	ADC, PinA 10x	OUT	10OHHHLLL	DIR	0	1	101	0
100110_OHHHLLL	010 15kΩ	ADC, PinA 31.6x	OUT	10OHHHLLL	DIR	0	1	110	0
100111_OHHHLLL	011 150kΩ	ADC, PinA 100x	OUT	10OHHHLLL	DIR	0	1	111	0
10100_DDDDDDDD	100 1mA	ADC, PinA 1x	It OUT = 1	10xxxxxxx	0	DIR	OUT	011	0
10101_DDDDDDDD	101 100μA	ADC, PinA 1x	DAC 990Ω, 3.3V	10xxxxxxx	0	DIR	OUT	011	0
10110_DDDDDDDD	110 10μA	ADC, PinA 1x	DAC 600Ω, 2.0V	10xxxxxxx	0	DIR	OUT	011	0
10111_DDDDDDDD	111 Float	ADC, PinA 1x	DAC 123.75Ω, 3.3V	10xxxxxxx	0	DIR	OUT	011	0
1100_CDDDDDDDD	<u>DDDDDDDD</u> DAC Level	ADC, PinA 1x	DAC 75Ω, 2.0V	C00001001	DIR	0	0		A>D
1101_CDDDDDDDD	<u>DIR</u> Pin	PinA > D	OUT, 1.5kΩ	C01001001	DIR	0	0		A>D
1110_CDDDDDDDD	0 Float	PinA > D	Input, 1.5kΩ	C00001001	DIR	0	0		B>D
1111_CDDDDDDDD	1 Drive	PinB > D	Input, 1.5kΩ	C01001001	DIR	0	0		B>D

1.2.4) TT DIR\Out Control

Pin DIR/OUT Control (%TT)		
Default = %00		
for odd pins	'OTHER' = even pin's NOT 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' outputs (2{P[7:4]}) for 'high' or (2{P[3:0]}) for 'low' 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 DAC, 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 DAC in DAC_MODE, P[7:0] overridden
	1x	OTHER enables DAC in DAC_MODE, P[7:0] overridden
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

1.2.5) SSSSS Smart Pin Mode Setting

Smart Pin Modes		
%SSSSS	Mode	Note
00000	smart pin off (default)	
00001	long repository	M[12:10] != %101
00010	long repository	M[12:10] != %101
00011	long repository	M[12:10] != %101
00001	DAC noise	M[12:10] = %101
00010	DAC 16-bit dither, noise	M[12:10] = %101
00011	DAC 16-bit dither, PWM	M[12:10] = %101
00100 ¹	pulse/cycle output	
00101 ¹	transition output	
00110 ¹	NCO frequency	
00111 ¹	NCO duty	
01000 ¹	PWM triangle	
01001 ¹	PWM sawtooth	
01010 ¹	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 on X 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	ADC sample/filter/capture, internally clocked	
11001	ADC sample/filter/capture, externally clocked	
11010	ADC scope with trigger	
11011 ¹	USB host/device	even/odd pin pair = DM/DP
11100 ¹	sync serial transmit	A-data, B-clock
11101	sync serial receive	A-data, B-clock
11110 ¹	async serial transmit	baud rate
11111	async serial receive	baud rate

¹ OUT signal overridden

1.3) Smart Pin Symbol Names

Smart Pin Symbol Value	Symbol Name	Details
A Input Polarity	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_A (default)	True A input
%1000_0000_000_00000000000000_00_00000_0	P_INVERT_A	Invert A input
A Input Selection	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_LOCAL_A (default)	Select local pin for A input
%0001_0000_000_00000000000000_00_00000_0	P_PLUS1_A	Select pin+1 for A input
%0010_0000_000_00000000000000_00_00000_0	P_PLUS2_A	Select pin+2 for A input
%0011_0000_000_00000000000000_00_00000_0	P_PLUS3_A	Select pin+3 for A input
%0100_0000_000_00000000000000_00_00000_0	P_OUTBIT_A	Select OUT bit for A input
%0101_0000_000_00000000000000_00_00000_0	P_MINUS3_A	Select pin-3 for A input
%0110_0000_000_00000000000000_00_00000_0	P_MINUS2_A	Select pin-2 for A input
%0111_0000_000_00000000000000_00_00000_0	P_MINUS1_A	Select pin-1 for A input
B Input Polarity	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_B (default)	True B input
%0000_1000_000_00000000000000_00_00000_0	P_INVERT_B	Invert B input
B Input Selection	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_LOCAL_B (default)	Select local pin for B input
%0000_0001_000_00000000000000_00_00000_0	P_PLUS1_B	Select pin+1 for B input

%0000_0010_000_00000000000000_00_00000_0	P_PLUS2_B	Select pin+2 for B input
%0000_0011_000_00000000000000_00_00000_0	P_PLUS3_B	Select pin+3 for B input
%0000_0100_000_00000000000000_00_00000_0	P_OUTBIT_B	Select OUT bit for B input
%0000_0101_000_00000000000000_00_00000_0	P_MINUS3_B	Select pin-3 for B input
%0000_0110_000_00000000000000_00_00000_0	P_MINUS2_B	Select pin-2 for B input
%0000_0111_000_00000000000000_00_00000_0	P_MINUS1_B	Select pin-1 for B input
A, B Input Logic	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_PASS_AB (default)	Select A, B
%0000_0000_001_00000000000000_00_00000_0	P_AND_AB	Select A & B, B
%0000_0000_010_00000000000000_00_00000_0	P_OR_AB	Select A B, B
%0000_0000_011_00000000000000_00_00000_0	P_XOR_AB	Select A ^ B, B
%0000_0000_100_00000000000000_00_00000_0	P_FILT0_AB	Select FILT0 settings for A, B
%0000_0000_101_00000000000000_00_00000_0	P_FILT1_AB	Select FILT1 settings for A, B
%0000_0000_110_00000000000000_00_00000_0	P_FILT2_AB	Select FILT2 settings for A, B
%0000_0000_111_00000000000000_00_00000_0	P_FILT3_AB	Select FILT3 settings for A, B
Low-Level Pin Modes	(pick one)	
Logic/Schmitt/Comparator Input Modes		
%0000_0000_000_00000000000000_00_00000_0	P_LOGIC_A (default)	Logic level A → IN, output OUT

%0000_0000_000_00010000000000_00_00000_0	P_LOGIC_A_FB	Logic level A → IN, output feedback
%0000_0000_000_00100000000000_00_00000_0	P_LOGIC_B_FB	Logic level B → IN, output feedback
%0000_0000_000_00110000000000_00_00000_0	P_SCHMITT_A	Schmitt trigger A → IN, output OUT
%0000_0000_000_01000000000000_00_00000_0	P_SCHMITT_A_FB	Schmitt trigger A → IN, output feedback
%0000_0000_000_01010000000000_00_00000_0	P_SCHMITT_B_FB	Schmitt trigger B → IN, output feedback
%0000_0000_000_01100000000000_00_00000_0	P_COMPARE_AB	A > B → IN, output OUT
%0000_0000_000_01110000000000_00_00000_0	P_COMPARE_AB_FB	A > B → IN, output feedback
%xxxx_xxxx_xxx_xxxxSIOHHHLLL_xx_xxxxx_x		Sync mode, IN/output polarity, high/low drive
ADC Input Modes		
%0000_0000_000_10000000000000_00_00000_0	P_ADC_GIO	ADC GIO → IN, output OUT
%0000_0000_000_10000100000000_00_00000_0	P_ADC_VIO	ADC VIO → IN, output OUT
%0000_0000_000_10001000000000_00_00000_0	P_ADC_FLOAT	ADC FLOAT → IN, output OUT
%0000_0000_000_10001100000000_00_00000_0	P_ADC_1X	ADC 1x → IN, output OUT
%0000_0000_000_10010000000000_00_00000_0	P_ADC_3X	ADC 3.16x → IN, output OUT

%0000_0000_000_10010100000000_00_00000_0	P_ADC_10X	ADC 10x → IN, output OUT
%0000_0000_000_10011000000000_00_00000_0	P_ADC_30X	ADC 31.6x → IN, output OUT
%0000_0000_000_10011100000000_00_00000_0	P_ADC_100X	ADC 100x → IN, output OUT
%xxxx_xxxx_xxx_xxxxxxOHHHLLL_xx_xxxxx_x		O = output polarity, HHH/LLL = high/low drive
DAC Output Modes		DIR enables output, OUT enables ADC
%0000_0000_000_10100000000000_00_00000_0	P_DAC_990R_3V	DAC 990Ω, 3.3V peak, ADC 1x → IN
%0000_0000_000_10101000000000_00_00000_0	P_DAC_600R_2V	DAC 600Ω, 2.0V peak, ADC 1x → IN
%0000_0000_000_10110000000000_00_00000_0	P_DAC_124R_3V	DAC 123.75Ω, 3.3V peak, ADC 1x → IN
%0000_0000_000_10111000000000_00_00000_0	P_DAC_75R_2V	DAC 75Ω, 2.0V peak, ADC 1x → IN
%xxxx_xxxx_xxx_xxxxxxDDDDDDDD_xx_xxxxx_x		DDDDDDDD = 8-bit DAC value
Level-Comparison Modes		DIR enables output (1.5kΩ drive)
%0000_0000_000_11000000000000_00_00000_0	P_LEVEL_A	A > Level → IN, output OUT
%0000_0000_000_11010000000000_00_00000_0	P_LEVEL_A_FBN	A > Level → IN, output negative feedback

%0000_0000_000_11100000000000_00_00000_0	P_LEVEL_B_FBP	B > Level → IN, output positive feedback
%0000_0000_000_11110000000000_00_00000_0	P_LEVEL_B_FBN	B > Level → IN, output negative feedback
%xxxx_xxxx_xxx_xxxxSLLLLLLLLL_xx_xxxxx_x		S = Synchronous, LLLLLLLLL = 8- bit Level
Low-Level Pin Sub-Modes		
Sync Mode	(pick one)	(for Logic/Schmitt/Co mparator/Level modes)
%xxxx_xxxx_xxx_xxxxSxxxxxxxx_xx_xxxxx_x		Sync mode bit
%0000_0000_000_00000000000000_00_00000_0	P_ASYNC_IO (default)	Select asynchronous I/O
%0000_0000_000_00001000000000_00_00000_0	P_SYNC_IO	Select synchronous I/O
IN Polarity	(pick one)	(for Logic/Schmitt/Co mparator modes)
%xxxx_xxxx_xxx_xxxxIxxxxxxxx_xx_xxxxx_x		IN polarity bit
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_IN (default)	True IN bit
%0000_0000_000_00000100000000_00_00000_0	P_INVERT_IN	Invert IN bit
Output Polarity	(pick one)	(for Logic/Schmitt/Co mparator/ADC modes)
%xxxx_xxxx_xxx_xxxxxOxxxxxx_xx_xxxxx_x		Output polarity bit
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_OUTPUT (default)	Select true output

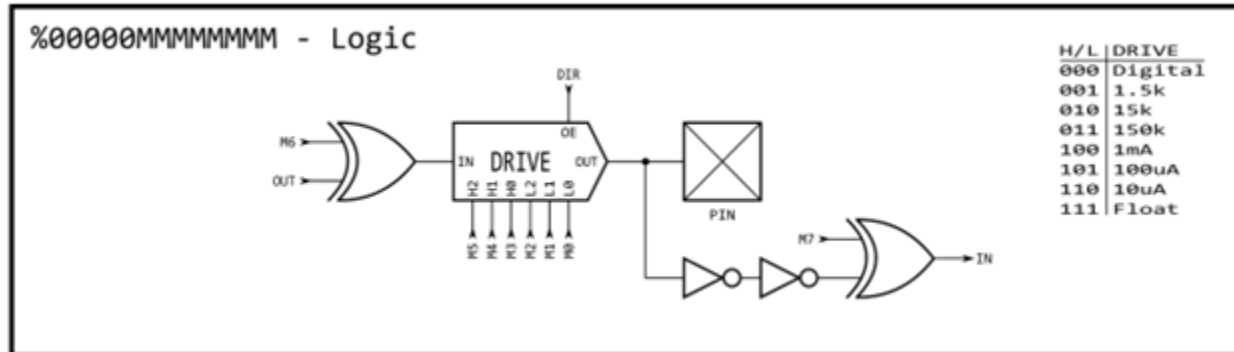
%0000_0000_000_00000001000000_00_00000_0	P_INVERT_OUTPUT	Select inverted output
Drive-High Strength	(pick one)	(for Logic/Schmitt/Comparator/ADC modes)
%xxxx_xxxx_xxx_00000000HHHxxx_xx_00000_x		Drive-high selector bits
%0000_0000_000_00000000000000_00_00000_0	P_HIGH_FAST (default)	Drive high fast (30mA)
%0000_0000_000_00000000001000_00_00000_0	P_HIGH_1K5	Drive high 1.5kΩ
%0000_0000_000_00000000010000_00_00000_0	P_HIGH_15K	Drive high 15kΩ
%0000_0000_000_00000000011000_00_00000_0	P_HIGH_150K	Drive high 150kΩ
%0000_0000_000_00000000100000_00_00000_0	P_HIGH_1MA	Drive high 1mA
%0000_0000_000_00000000101000_00_00000_0	P_HIGH_100UA	Drive high 100μA
%0000_0000_000_00000000110000_00_00000_0	P_HIGH_10UA	Drive high 10μA
%0000_0000_000_00000000111000_00_00000_0	P_HIGH_FLOAT	Float high
Drive-Low Strength	(pick one)	(for Logic/Schmitt/Comparator/ADC modes)
%xxxx_xxxx_xxx_00000000000000LLL_xx_00000_x		Drive-low selector bits
%0000_0000_000_00000000000000_00_00000_0	P_LOW_FAST (default)	Drive low fast (30mA)
%0000_0000_000_000000000000001_00_00000_0	P_LOW_1K5	Drive low 1.5kΩ
%0000_0000_000_000000000000010_00_00000_0	P_LOW_15K	Drive low 15kΩ
%0000_0000_000_000000000000011_00_00000_0	P_LOW_150K	Drive low 150kΩ
%0000_0000_000_000000000000100_00_00000_0	P_LOW_1MA	Drive low 1mA
%0000_0000_000_000000000000101_00_00000_0	P_LOW_100UA	Drive low 100μA

%0000_0000_000_000000000000110_00_00000_0	P_LOW_10UA	Drive low 10μA
%0000_0000_000_000000000000111_00_00000_0	P_LOW_FLOAT	Float low
DIR/OUT Control	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_TT_00 (default)	TT = %00
%0000_0000_000_00000000000001_01_00000_0	P_TT_01	TT = %01
%0000_0000_000_00000000000010_10_00000_0	P_TT_10	TT = %10
%0000_0000_000_00000000000011_11_00000_0	P_TT_11	TT = %11
%0000_0000_000_00000000000001_01_00000_0	P_OE	Enable output in smart pin mode
%0000_0000_000_00000000000001_01_00000_0	P_CHANNEL	Enable DAC channel in non-smart pin DAC mode
%0000_0000_000_00000000000010_10_00000_0	P_BITDAC	Enable BITDAC for non-smart pin DAC mode
Smart Pin Modes	(pick one)	
%0000_0000_000_00000000000000_00_00000_0	P_NORMAL (default)	Normal mode (not smart pin mode)
%0000_0000_000_00000000000000_00_00001_0	P_REPOSITORY	Long repository (non-DAC mode)
%0000_0000_000_00000000000000_00_00001_0	P_DAC_NOISE	DAC Noise (DAC mode)
%0000_0000_000_00000000000000_00_00010_0	P_DAC_DITHER_RND	DAC 16-bit random dither (DAC mode)
%0000_0000_000_00000000000000_00_00011_0	P_DAC_DITHER_PWM	DAC 16-bit PWM dither (DAC mode)
%0000_0000_000_00000000000000_00_00100_0	P_PULSE	Pulse/cycle output
%0000_0000_000_00000000000000_00_00101_0	P_TRANSITION	Transition output

%0000_0000_000_0000000000000000_00_00110_0	P_NCO_FREQ	NCO frequency output
%0000_0000_000_0000000000000000_00_00111_0	P_NCO_DUTY	NCO duty output
%0000_0000_000_0000000000000000_00_01000_0	P_PWM_TRIANGLE	PWM triangle output
%0000_0000_000_0000000000000000_00_01001_0	P_PWM_SAWTOOTH	PWM sawtooth output
%0000_0000_000_0000000000000000_00_01010_0	P_PWM_SMPS	PWM switch-mode power supply I/O
%0000_0000_000_0000000000000000_00_01011_0	P_QUADRATURE	A-B quadrature encoder input
%0000_0000_000_0000000000000000_00_01100_0	P_REG_UP	Inc on A-rise when B-high
%0000_0000_000_0000000000000000_00_01101_0	P_REG_UP_DOWN	Inc on A-rise when B-high, dec on A-rise when B-low
%0000_0000_000_0000000000000000_00_01110_0	P_COUNT_RISES	Inc on A-rise, optionally dec on B-rise
%0000_0000_000_0000000000000000_00_01111_0	P_COUNT_HIGHS	Inc on A-high, optionally dec on B-high
%0000_0000_000_0000000000000000_00_10000_0	P_STATE_TICKS	For A-low and A-high states, count ticks
%0000_0000_000_0000000000000000_00_10001_0	P_HIGH_TICKS	For A-high states, count ticks
%0000_0000_000_0000000000000000_00_10010_0	P_EVENTS_TICKS	For X A-highs/rises/edges, count ticks / Timeout on X ticks of no A-high/rise/edge

%0000_0000_000_00000000000000_00_10011_0	P_PERIODS_TICKS	For X periods of A, count ticks
%0000_0000_000_00000000000000_00_10100_0	P_PERIODS_HIGHS	For X periods of A, count highs
%0000_0000_000_00000000000000_00_10101_0	P_COUNTER_TICKS	For periods of A in X+ ticks, count ticks
%0000_0000_000_00000000000000_00_10110_0	P_COUNTER_HIGHS	For periods of A in X+ ticks, count highs
%0000_0000_000_00000000000000_00_10111_0	P_COUNTER_PERIODS	For periods of A in X+ ticks, count periods
%0000_0000_000_00000000000000_00_11000_0	P_ADC	ADC sample/filter/capture, internally clocked
%0000_0000_000_00000000000000_00_11001_0	P_ADC_EXT	ADC sample/filter/capture, externally clocked
%0000_0000_000_00000000000000_00_11010_0	P_ADC_SCOPE	ADC scope with trigger
%0000_0000_000_00000000000000_00_11011_0	P_USB_PAIR	USB pin pair
%0000_0000_000_00000000000000_00_11100_0	P_SYNC_TX	Synchronous serial transmit
%0000_0000_000_00000000000000_00_11101_0	P_SYNC_RX	Synchronous serial receive
%0000_0000_000_00000000000000_00_11110_0	P_ASYNC_TX	Asynchronous serial transmit
%0000_0000_000_00000000000000_00_11111_0	P_ASYNC_RX	Asynchronous serial receive

2.0) Digital Pin Operation Logic Output (Smart Pin Off)



00000M7M6M5M4M3M2M1M0 = 00000 I O HHH LLL = Logic (output)

Even though Pin mode is being set as an Output the sinking and sourcing resistors allow the Pin to be a Input. To use Pin as a traditional Output the Low and High drive currents should be set to Fast.

LLL = M2M1M0 = 000 H2H1H0 = M5M4M3 = 000

For Sinking Input the Direction bit DIR must be set High (1) for Output and the Output bit must be set Low (0) with the appropriate Low drive bits LLL(M2M1M0) set for sink value (Resistor Load or Current Load). The High drive bits do not matter(don't care)

For Sourcing Input the Direction bit DIR must be set High (1) for Output and the Output bit must be set High (1) with the appropriate High drive bits HHH(M5M4M3) set for source value(Resistor Load or Current Load). The Low drive bits do not matter(don't care).

The Logic operation for Input and Output can be inverted using mode bits M7 and M6

M6 [O(out)] = 1 invert pin (OUTx = !PinStatus) M6[O(out)] = 0 Non invert pin (OUTx = PinStatus)

M7[I(in)] = 1 invert pin (PinStatus = !Inx) M7[I(in)] = 0 Non invert pin (PinStatus = Inx)

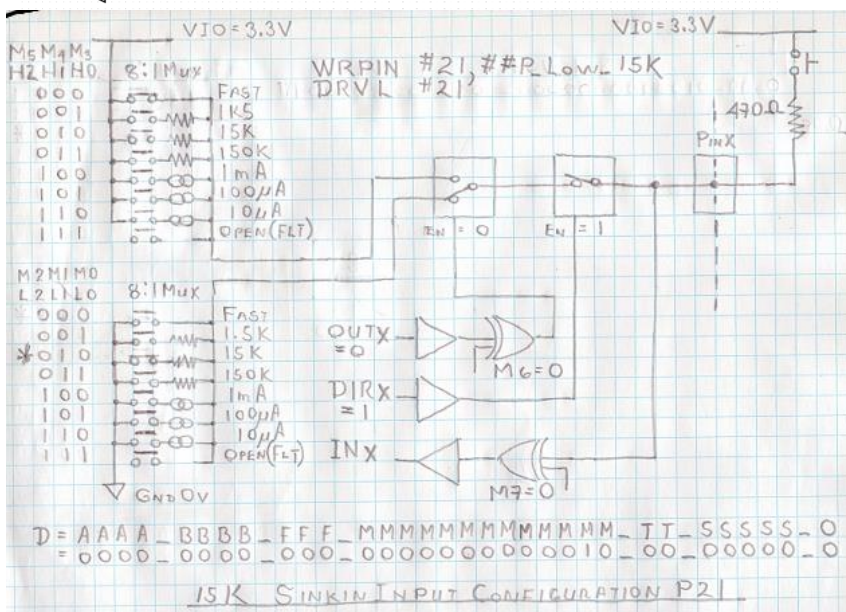
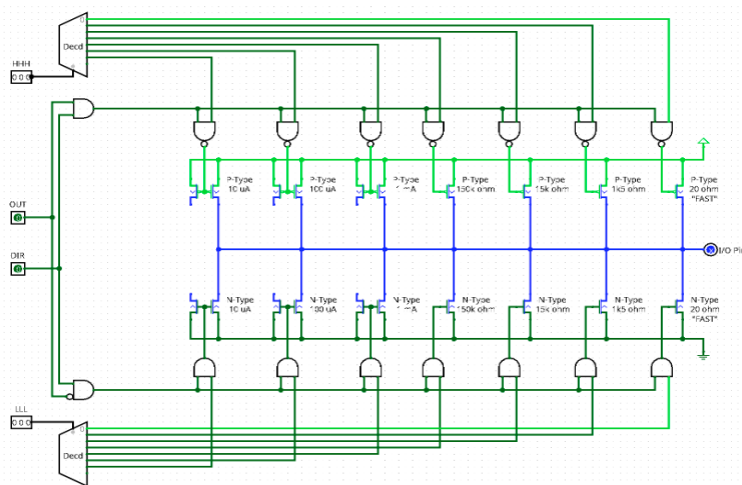
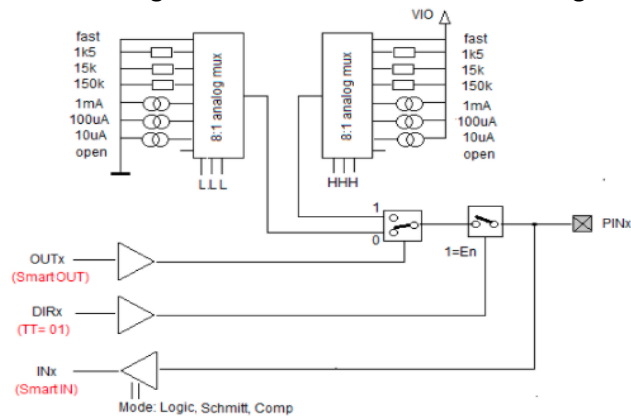
Example

1.5K source Non invert out and Non Invert in M7M6 =00 M5M4M3M2M1M0 =001000

1.5k sink Non invert out and Non Invert in M7M6 =00 M5M4M3M2M1M0 =000001

2.0.1 Sketch of 15K Sinking Source Input Push Button

The "Drive" Logic block is broken into the following components:



The resistors/current sources are a kind of drive strength. To get a pullup, you need to set the pin to output with a drive strength of 1.5k or 15k or 150k Ohm for HHH and you need to output a High (OUTx=1). Also if the pin is set to output, you still can read the input.

If the pin is in smartmode, the red labels are valid, DIRx and OUTx do no longer control the IO signals.

D = %AAAA_BBBB_FFF_MMMMMMMMMMMMMM_TT_SSSS_0

AAAA	= 0000	"x000 this pin read state"
BBBB	= 0000	"x000 this pin read state"
FFF	= 000	"A,B default filter"
MMMMMMMMMMMMMMMM	= 000000000000	"Fast"
TT	= 00	"can't figure out what table says"
SSSS	= 00000	"smart pin off"

By default Propeller II starts with ('WRPIN #0,pin') Fast Logic Mode

D = %0000_0000_000_000000000000_00_00000_0

If you're familiar with the assembly-language input-output instructions for the Propeller-1 microcontroller you will recognize the following six instructions a Propeller-2 program also may use these registers. These registers give you direct access to I/O pins:

DIRA direction register pins P0..P31, 1= output, 0 = disable output
 DIRB direction register pins P63-P32, 1= output, 0 = disable output
 OUTA output register bits for pins P0..P31
 OUTB output register bits for pins P32..P63
 INA input register bits for pins P0..P31
 INB input register bits for pins P32..P63

Propeller II does not have instructions to above registers as propeller 1 DIRA[0]~~ but the registers can be accessed with:

```
mov dira,#0
mov reg1,ina
mov outa,#1
```

Definition of PinField

PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

The above direction and output registers can be affected using special instructions which operate on 1 to 32 bits within each register.

In the following lists, {#}D denotes an 11-bit value, (PinField) with the 6 lower bits pointing to a base pin and the next upper 5 bits expressing an additional number of pins within the same I/O register.

{#}D = PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

11 bits for PinField LLLLL 5bits for number of additional pins PPPPPP 6 bits for base pin number

%00011_000101 base pin 5 plus 3 pins P3 P4 P5 P6 total 4 pins

%11111_0010000 base pin 8 plus 31 pins wrapping occurs P8-P31 plus P0-P7 (P0-P31)

PinField := BasePin addpins Add_pins eg. PinField := 0 addpins 7 ' P0-P7 assigned

In these instructions, bit 5 of {#}D selects between DIRA/DIRB or OUTA/OUTB.

10-09-08-07-06__05-04-03-02-01-00

16 08 04 02 01 __32 16 08 04 02 01

The **ADDPINS operator** can be used to set the additional-bits field in {#}D as follows:

DIRH #8 'Drive P8 high

DIRH #10 ADDPINS 7 'Drive P10..P17 high {#}D = 00111_001010 = LLLLL_PPPPPP

Each cog has its own pairs of 32-bit I/O Direction Registers (DIRA & DIRB) and 32-bit I/O Output Registers (OUTA & OUTB) to influence the directions and output states of the Propeller 2's 64 I/O pins. A cog's desired I/O directions and output states are communicated through the entire cog collective to ultimately become what is applied to the I/O pins.

The result of this I/O pin wiring configuration can easily be described in the following simple rules:

- * A pin is an input only if no active cog sets it to an output.
- * A pin outputs low only if all active cogs that set it to output also set it to low.
- * A pin outputs high if any active cog sets it to an output and also sets it high.

The Propeller 2 is a CMOS device, so the I/O pin digital logic threshold is approximately 1/2 Vdd.

With the Propeller 2's I/O pins powered by 3.3 V (via the corresponding Vxxyy pins), the I/O pin digital logic threshold is about 1.65 V.

An input pin will interpret a voltage below 1.65 V as a digital logic level low, and will interpret a voltage above 1.65 V as a digital logic level high.

An output pin will produce 0 V for digital low and 3.3 V for digital high.

2.0.1_Example_WRD_Set Pin for Sinking Input Using Spin2

00000MMMMMMMMM = Pin Logic

M5M4M3M2M1M0 = H2H1H0L2L1L0 = 000010

M6 = 0 Non invert pin (OUTx = PinStatus)

M7 = 0 Non invert pin (PinStatus = Inx)

P_LOW_15K = %0000_0000_000_0000000000010_00_00000_0

D = %AAAA_BBBB_FFF_MMMMMMMMMMMMMM_TT_SSSSS_0

AAAA = 0000 "x000 this pin read state"

BBBB = 0000 "x000 this pin read state"

FFF = 000 "A,B default filter"

00000MMMMMMMMM = 00000000000010 "15k Sink"

TT = 00 "can't figure out what table says"

SSSSS = 00000 "smart pin off"

2.0.2_Example_WRD_Set pin 21 for sinking input using PASM2

2.0.3_Example_WRD_Set pin 21 for sinking using PASM2 and read using PASM2

2.1) PASM Pin Digital Commands

2.1.1) DIR Bit Instruction

{#}D = PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

These instruction bits affect the associated DIR direction bit:

DIRL {#}D	Set direction bit(s) to logic 0 (input)
DIRH {#}D	Set direction bit(s) to logic 1 (output)
DIRC {#}D	Set direction bit(s) to Carry flag
DIRNC {#}D	Set direction bit(s) to inverse of Carry flag
DIRZ {#}D	Set direction bit(s) to Zero flag
DIRNZ {#}D	Set direction bit(s) to inverse of Zero flag
DIRRND {#}D	Set direction bit(s) to random state(s)
DIRNOT {#}D	Invert direction bit(s)

Example: DIRL #20 'Set P20 as an input pin

2.1.2) Pin-Output Instructions

{#}D = PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

These Instructions change the associated OUT bit(s)

OUTL {#}D	Set output bit(s) to logic 0
OUTH {#}D	Set output bit(s) to logic 1
OUTC {#}D	Set output bit(s) to Carry flag
OUTNC {#}D	Set output bit(s) to inverse of Carry flag
OUTZ {#}D	Set output bit(s) to Zero flag
OUTNZ {#}D	Set output bit(s) to inverse of Zero flag
OUTRND {#}D	Set output bit(s) to random state(s)
OUTNOT {#}D	Invert output bit(s)

Example: OUTNOT \$20 'Invert the logic state of the P20 output

2.1.3) Pin-Float Instructions

{#}D = PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

These instructions change the associated DIR bit(s) to logic-0(input float)

FLTL {#}D	Set output bit(s) to logic 0
FLTH {#}D	Set output bit(s) to logic 1
FLTC {#}D	Set output bit(s) to Carry flag
FLTNC {#}D	Set output bit(s) to inverse of Carry flag
FLTZ {#}D	Set output bit(s) to Zero flag
FLTNZ {#}D	Set output bit(s) to inverse of Zero flag
FLTRND {#}D	Set output bit(s) to random state(s)
FLTNOT {#}D	Invert output bit(s)

Example: FLTC #20 'Make P20 input with its output bit set to C.

2.1.4) Pin-Drive Instructions

{#}D = PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

These instructions change the associated DIR bit(s) to logic-1 (output).

DRV L	{#}D	Set output bit(s) to logic-0
DRV H	{#}D	Set output bit(s) to logic-1
DRV C	{#}D	Set output bit(s) to Carry flag value
DRV NC	{#}D	Set output bit(s) to inverse of Carry flag
DRV Z	{#}D	Set output bit(s) to Zero flag
DRV NZ	{#}D	Set output bit(s) to inverse of Zero flag
DRV RND	{#}D	Set output bit(s) to random state(s)
DRV NOT	{#}D	Invert output bit(s)

Example: DRVZ #20 'Make P20 output the Z-flag state.

2.1.5) Input-Pin Instructions

{#}D (pinfield) represents a pin number.

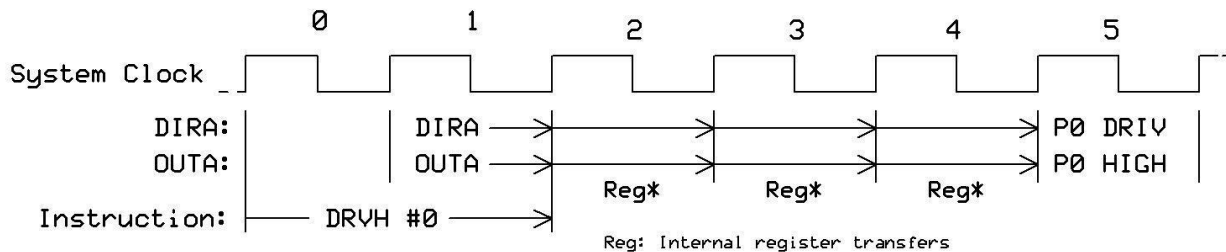
Two instructions, TESTP and TESTPN can read the state of a single bit within an INA/INB register and either write that bit to the Carry (C) or Zero (Z) flag, or perform a logic operation on the flag.

TESTP	{#}D	WC/WZ	Get a pin's state and write it into the C or Z flag.
TESTP	{#}D	ANDC/ANDZ	Get a pin's state and AND it into the C or Z flag.
TESTP	{#}D	ORC/ORZ	Get a pin's state and OR it into the C or Z flag.
TESTP	{#}D	XORC/XORZ	Get a pin's state and XOR it into the C or Z flag.
TESTPN	{#}D	WC/WZ	Get a pin's NOT-state and write it into the C or Z flag.
TESTPN	{#}D	ANDC/ANDZ	Get a pin's NOT-state and AND it into the C or Z flag.
TESTPN	{#}D	ORC/ORZ	Get a pin's NOT-state and OR it into the C or Z flag.
TESTPN	{#}D	XORC/XORZ	Get a pin's NOT-state and XOR it into the C or Z flag.

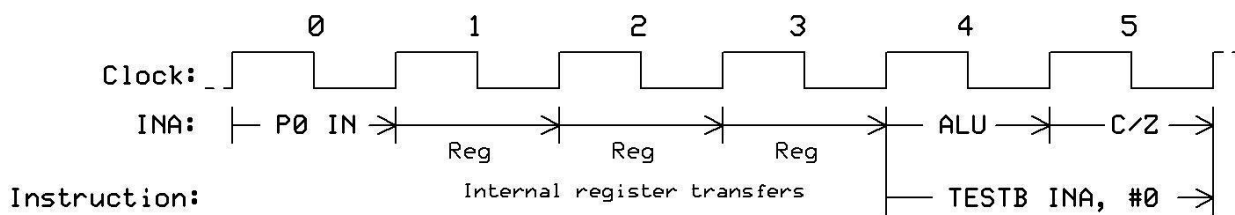
Example: TESTP #10 ORZ 'Read P10 and or its state into Z.

2.1.5) Input-Output-Bit Timing

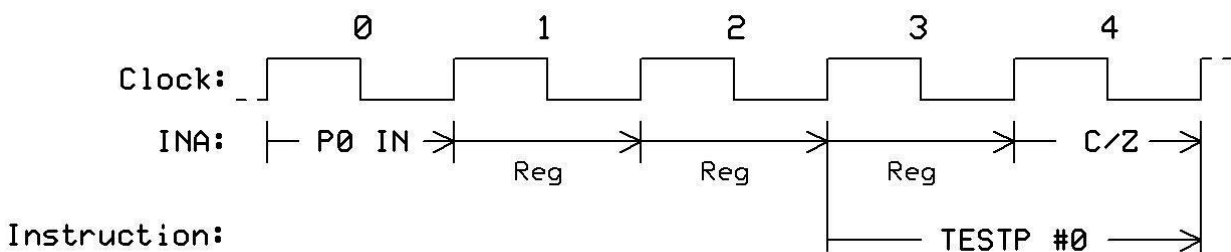
When an instruction changes a DIRx or OUTx bit, the processor needs three (3) additional system-clock cycles *after the instruction* before the pin starts to transition to its new state. The figure below shows the delay for a DRVH instruction:



When an instruction reads the contents of the IN register associated with a pin, the processor receives the state of the pins as they existed three (3) system-clock cycles *before* the start of the instruction. The figure below shows the timing for a the TESTB INA,#0 operation:



When a program uses a TESTP or TESTPN instruction to read the state of a pin, the processor receives the state of the pins as they existed two (2) system-clock cycles *before* the start of the instruction. So, the TESTP and TESTPN gather "fresher" INx data than is available via the INx registers. The figure below shows the timing for a TESTP instruction:



[2.1.1_Example_WRD_PASM_DIRH_OUTH](#)

DIRH {#}D Set direction bit(s) to logic 1 (output)

OUTH {#}D Set output bit(s) to logic 1

P0 used with 1k resistor to LED

[2.1.2_Example_Wrd_PASM_DRVH_DRVL.spin2](#)

P0-P7 used with 1k resistor to LED

[2.1.3_Example_WRD_PASM_Cog_Assembly](#)

Demonstrates Calling an assembly program

[2.1.4_Example_WRD_PASM_Cog_Assembly](#)

Demonstrates running two cog programs simultaneously

2.2) SPIN I/O Digital Methods

PinField = 11 bits %LLLLL_PPPPPP %extrapins_basepins

PinField = %00011_000101 base pin 5 plus 3 pins P3 P4 P5 P6 total 4 pins

%11111_0010000 base pin 8 plus 31 pins wrapping occurs P8-P31 plus P0-P7 (P0-P31)

PinField := BasePin **addpins** Add_pins eg. PinField := 0 addpins 7 ' P0-P7 assigned

addpins is a Propeller Tool directive to create PinField

The following are spin 2 commands:

Pin Methods	Details
PINW PINWRITE(PinField, Data)	Drive PinField pin(s) with Data
PINL PINLOW(PinField)	Drive PinField pin(s) low
PINH PINHIGH(PinField)	Drive PinField pin(s) high
PINT PINTOGGLE(PinField)	Drive and toggle PinField pin(s)
PINF PINFLOAT(PinField)	Float PinField pin(s)
PINR PINREAD(PinField) : PinStates	Read PinField pin(s)
PINSTART(PinField, Mode, Xval, Yval)	Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1
PINCLEAR(PinField)	Clear PinField smart pin(s): DIR=0, then WRPIN=0
WRPIN(PinField, Data)	Write 'mode' register(s) of PinField smart pin(s) with Data
WXPIN(PinField, Data)	Write 'X' register(s) of PinField smart pin(s) with Data
WYPIN(PinField, Data)	Write 'Y' register(s) of PinField smart pin(s) with Data
AKPIN(PinField)	Acknowledge PinField smart pin(s)
RDPIN(Pin) : Zval	Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN data
RQPIN(Pin) : Zval	Read Pin smart pin without acknowledge, Zval[31] = C flag from RQPIN, other bits are RQPIN data

2.2.1_Example_WRD_SPIN_PINT_PINREAD_PINF_PINCLEAR

Demonstrates following Spin commands LED for P0-P7 1k resistor

PINT PINTOGGLE(PinField)	Drive and toggle PinField pin(s)
PINR PINREAD(PinField) : PinStates	Read PinField pin(s)
PINCLEAR(PinField)	Clear PinField smart pin(s): DIR=0, then WRPIN=0
PINF PINFLOAT(PinField)	Float PinField pin(s)

2.2.2_Example_WRD_SPIN_PINW_PINL_PINH

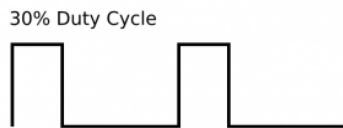
Demonstrates following commands LED for P0-P7 1k resistor

PINW PINWRITE(PinField, Data)	Drive PinField pin(s) with Data
PINL PINLOW(PinField)	Drive PinField pin(s) low
PINH PINHIGH(PinField)	Drive PinField pin(s) high
PINR PINREAD(PinField) : PinStates	Read PinField pin(s)
PINCLEAR(PinField)	Clear PinField smart pin(s): DIR=0, then WRPIN=0
PINF PINFLOAT(PinField)	Float PinField pin(s)

3.0) PWM Pulse Width Modulation with Smart Pin

Smart Pin Registers	
32-bit Register	Purpose
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)

The mechanism typically used to control the brightness of an LED is called PWM (Pulse Width Modulation). In our blink example the LED was either always on or always off. If we want an intermediate brightness, we need to have it partially on; this is the purpose of PWM. In this figure the on-time portion of the waveform is 30% of the entire cycle (on-time plus off-time). The ratio of on-time to cycle-time is called the duty cycle.

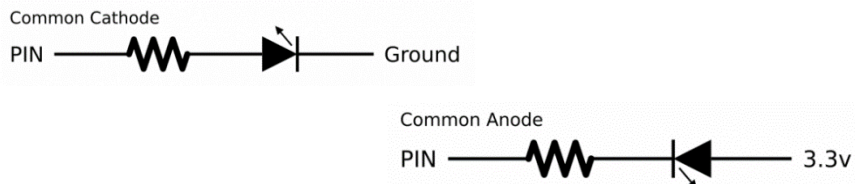


```
pinstart(led, m, x, y)
```

```
m := P_PWM_SAWTOOTH | P_OE
```

%0000_0000_000_00000000000000_00_01001_0	P_PWM_SAWTOOTH	PWM sawtooth output
%0000_0000_000_00000000000000_01_0000_0	P_OE	Enable output in smart pin mode

The first step is to select the PWM mode (sawtooth is the easiest to implement) and to make the smart pin an output with the P_OE constant. The output enable flag is required because in smart pin mode, the pin direction bit is used to enable or disable the smart pin.



If the Common Cathode connection is used the output being high will turn on the LED if the Common Anode connection is used the output being high will turn off the LED.

The output can be inverted by setting the P_INVERT_OUTPUT bit in the mode register:

m |= P_INVERT_OUTPUT

%0000_0000_000_00000001000000_00_0 0000_0	P_INVERT_OUTPUT	Select inverted output
--	-----------------	------------------------

x.word[1] := 255

The high word of the smart pin X register holds the value that will set the output to 100% duty cycle. As discussed, we will use 255.

x.word[0] := 1 #> ((clkfreq / hz) / 255) <# \$FFFF

Finally, the low word of the smart pin X register holds the number of system ticks in one unit for the desired PWM frequency. This takes a little bit of math, but, again, is fairly straightforward. It works out like this: the system clock frequency (clkfreq) is divided by the desired PWM frequency (hz); this gives us the number of system ticks in one PWM period. That is divided by the number of units in 100% (255) to get the number of system ticks in one unit. The #> and <# operators constrain the value to a legal 16-bit number for the low word of X.

Y register which holds the current level; in our setup this will be 0 (0%) to 255 (100%). To change the LED brightness at any time we can write to the smart pin Y register like this:

wypin(LED, 128) this sets 50% duty cycle

3.1_Example_WRD_PWM_Demo.spin2

Demonstrate Square Wave Scope

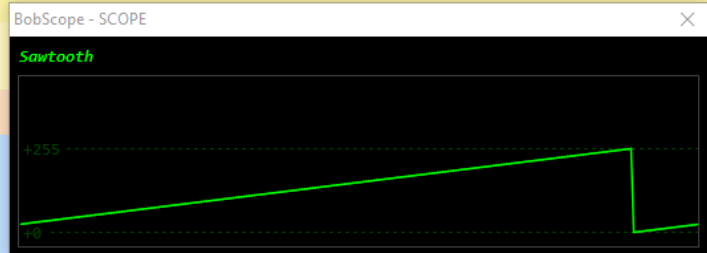
```
{3.1_Example_WRD_PWM_Demo}
con { timing }

CLK_FREQ = 200_000_000
MS_001   = CLK_FREQ / 1_000
US_001   = CLK_FREQ / 1_000_000

_clkfreq = CLK_FREQ

con
    #0, C_CATHODE, C_ANODE

var
    led
pub main() | C_Type, Pin, HZ, Index
    C_Type := C_CATHODE
    Index := 0
    PIN := 0
    Hz := 100
    debug(·SCOPE BobScope SIZE 512 128 SAMPLES 256)
    debug(·BobScope 'Sawtooth' 0 255 64 10 %1111)
    startx(PIN, C_Type, HZ)
    repeat
        debug(udec(Index))
        debug(·BobScope ·(Index))
        wupin(led, Index)
        waitms(100)
        Index += 1
        if Index == 256
            Index := 0
    repeat
```



```
DEBUG Output
Cog0 Index = 22
`BobScope 22
Cog0 Index = 23
`BobScope 23
Cog0 Index = 24
`BobScope 24
```

4.0) Analog Out Smart Pin(DAC)

4.1) DAC Digital to Analog Conversion

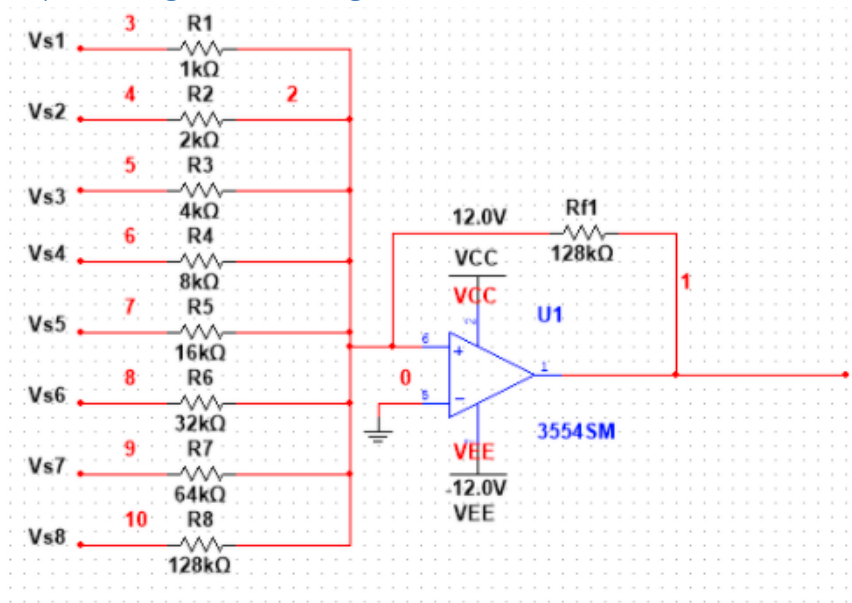


Figure 4.1.0

The above Schematic is the standard Method for Dac P2 Uses a Voltage Divider approach:

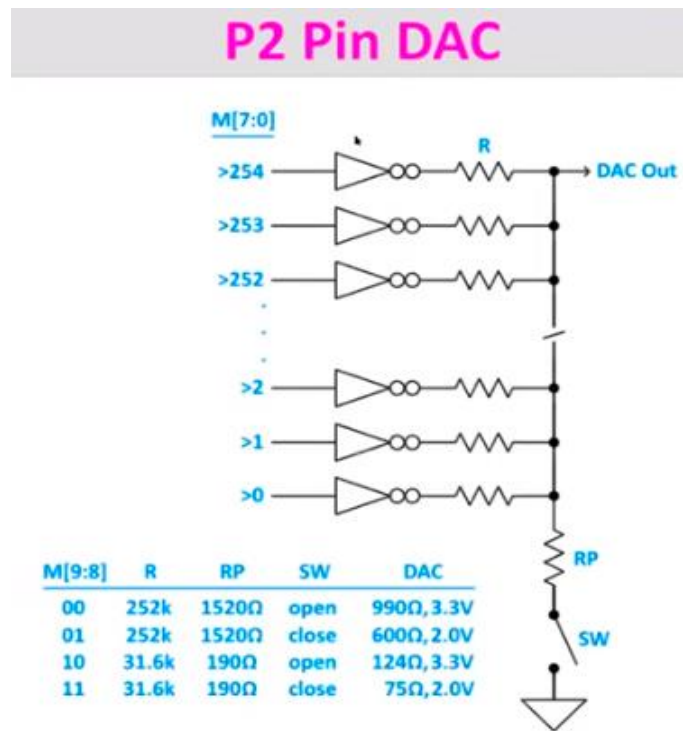


Figure 4.1.1

The above schematic is how the propeller II implements a Dac using a voltage divider

The main difference between a standard voltage divider and the Prop2 the Pro2 uses a more robust and faster acting arrangement that doesn't have the buffering op-amp. Therefore the output is pure resistive, hence why the output resistance is specified. The side effect of better linearity may be a result.

Note: The first Figure 4.1.0 is called a "binary weighted resistor" DAC. Linearity of that type becomes a major issue at higher word sizes. Figure 4.1.1 The "R-2R ladder" DAC is effectively the same but greatly improves on the linearity. Prop2 uses what's called a "Thermometer coded" or "Unary coded" DAC. It's pretty heavy on real-estate but does deliver high precision at speed.

Propeller smart pins each have 2 DAC's, one DAC has 255 resistors of 252K and the other DAC has 255 resistors of 31.6k. The resistors are pulled high or low dependant on the magnitude of the value essentially an 8 bit dac is created. If all resistors are pulled high the Voltage is 100% or value 255 (3.3v) if the value is 128 (\$80) the voltage is 50% half resistors high half low. If 0 is used all resistors pulled low. 255 resistors in parallel would be 252K/255 Plus the driver circuit impedance approx 990 ohm. For video on DAC see: <https://www.youtube.com/c/ParallaxInc/playlists> The essence of this dac is a voltage divider network that is set by a clocked Flip Flops.

1) Variable "pin" may be a PinField need to make sure only 1 pin is configured.

```
PinField = llll_pppppp  llll = 5 bits for addpins  pppppp = 6 bits for P0-P63 0-63
let pin = llll_pppppp
0000_0000_0000_0000_0000_0lll_lpp_pppp = pin
0000_0000_0000_0000_0000_0000_0011_1111 = $3F
pin &= $3F 'include this instruction to clear llll upper addpins
0000_0000_0000_0000_0000_0000_00pp_pppp = pin
```

2) Disable analog smart pin if previously configured

pinclear(Pin) ' disable smart pin

3) Following Spin method sets Pin for Digital output

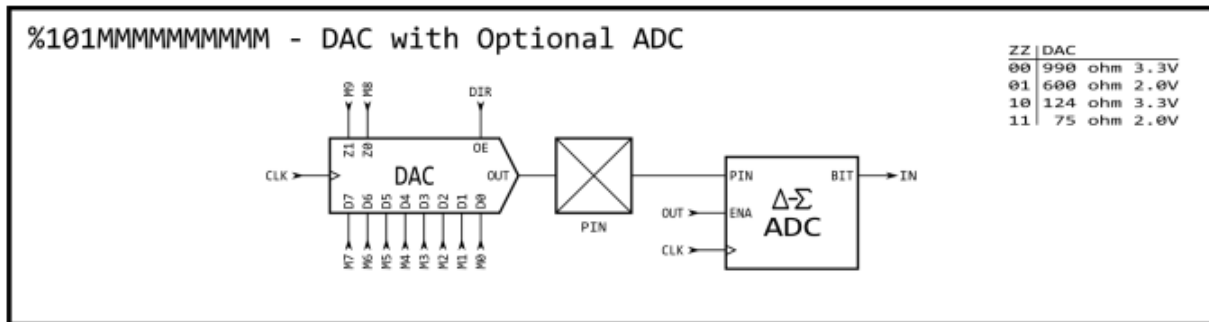
pinstart(pin, P_DAC_DITHER_PWM | P_DAC_990R_3V | P_OE, 256, 0)' 16-bit dac

-- https://docs.google.com/document/d/16qVkmA6Co5fUNKJHF6pBfGfDupuRwDtf-wyieh_fbqw/edit#heading=h.1h0sz9w9bl25

Built-In Symbols for Smart Pin Configuration; use Ctrl+F for searching strings

%0000_0000_000_000000000000_00_00011_0	P_DAC_DITHER_PWM	DAC 16-bit PWM dither (DAC mode)
%0000_0000_000_101000000000_00_00000_0	P_DAC_990R_3V	DAC 990Ω, 3.3V peak, ADC 1x → IN
%0000_0000_000_000000000000_01_00000_0	P_OE	Enable output in smart pin mode

4.1_Example_WRD_Analog_Out_Demo



PINSTART(PinField, Mode, Xval, Yval)

Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1

P_DAC_DITHER_PWM = %0000_0000_000_00000000000000_00_00011_0 DAC 16-bit PWM dither (DAC)

P_DAC_990R_3V = %0000_0000_000_10100000000000_00_00000_0 DAC 990Ω, 3.3V peak

POE = %0000_0000_000_00000000000000_01_00000_0 Enable smart pin mode

MODE = %0000_0000_000_10100000000000_01_00011_0 above values or'ed

PINFIELD (Pin) = 25 from calling 4.1_Example_WRD_Analog_Output_Demo

MODE = %0000_0000_000_10100000000000_01_00011_0 above values or'ed

WXPIN = 256 max unsigned 16 bit \$FFFF

WYPIN = 0 min unsigned

Pinstart(pin,MODE,WXPIN,WYPIN)

pinstart(pin, P_DAC_DITHER_PWM | P_DAC_990R_3V | P_OE, 256, 0) ' 16-bit dac selection

The pinstart method writes to the 3 registers WRPIN,WXPIN,WYPIN (below are the PASM instructions that can be used to write directly to these registers:

D = %AAAA_BBBB_FFF_MMMMMMMMMMMMMM_TT_SSSSS_0

- A = PINA input selector
- B = PINB input selector
- F = PINA and PINB input logic/filtering (after PINA and PINB input selectors)
- M = pin mode
- T = pin DIR/OUT control (default = %00)
- S = smart mode

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

4.1_Example_WRD_Analog_Output

File..... jm_analog_out.spin2

Purpose.... Simple P2 analog output using smart pin

Author..... Jon "JonnyMac" McPhalen Copyright (c) 2020-2021 Jon McPhalen MIT Licenc

PUB start(Pin,lo,hi) pin for analog out lo range 0 and high range 3300 for 3.3v

PINSTART(PinField, Mode, Xval, Yval)

Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1

WXPIN = 256 max unsigned 16 bit \$FFFF

WYPIN = 0 min unsigned

PINFIELD (Pin) = 25 from calling 4.1_Example_WRD_Analog_Output_Demo

P_DAC_DITHER_PWM = %0000_0000_000_00000000000000_00_00011_0 DAC 16-bit PWM dither

P_DAC_990R_3V = %0000_0000_000_10100000000000_00_00000_0 DAC 990Ω, 3.3V peak, ADC

POE = %0000_0000_000_00000000000000_01_00000_0 Enable output smart mode

MODE = %0000_0000_000_10100000000000_01_00011_0 above values or'ed

5.0) Analog Input Smart Pin (ADC)

5.1) ADC Analog Digital Conversion

Comparator Circuit

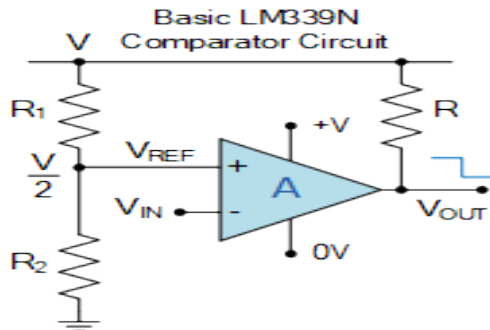


Figure 5.1.0

An analogue comparator such as the LM339N which has two analogue inputs, one positive and one negative, and which can be used to compare the magnitudes of two different voltage levels. A voltage input, (V_{IN}) signal is applied to one input of the comparator, while a reference voltage, (V_{REF}) to the other. A comparison of the two voltage levels at the comparator's input is made to determine the comparators digital logic output state, either a "1" or a "0".

The reference voltage, V_{REF} is compared against the input voltage, V_{IN} applied to the other input. For an LM339 comparator, if the input voltage is less than the reference voltage, ($V_{IN} < V_{REF}$) the output is "OFF", and if it is greater than the reference voltage, ($V_{IN} > V_{REF}$) the output will be "ON". Thus a comparator compares two voltage levels and determines which one of the two is higher.

2-bit ADC Using Diodes

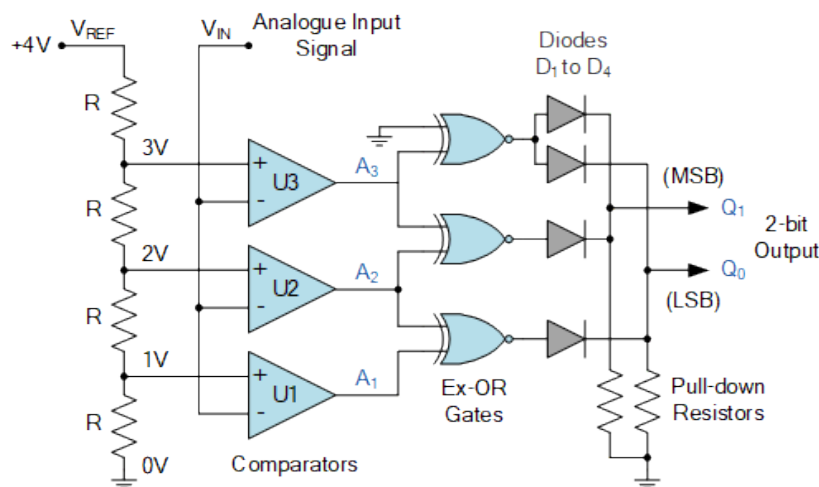
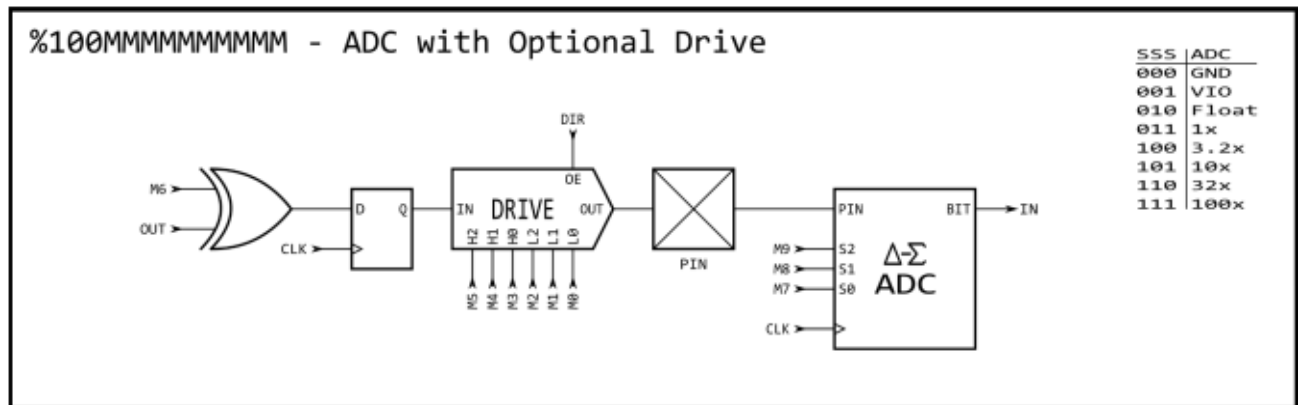


Figure 5.1.1

Prop2's ADCs are what's called a "sigma-delta modulator" or "delta-sigma modulator". Just a single fast comparator with feedback, which oscillates in the MHz region producing a natural bit-stream known as pulse-density-modulation (PDM). This is subsequently vacuumed up by a counter configured as a "Sinc filter/decimator" from which you then get your typical PCM samples.

5.1_Example_WRD_Analog_Input_Demo



D = %AAAA_BBBB_FFF_MMMMMMMMMMMMMM_TT_SSSSS_0

- A = PINA input selector
- B = PINB input selector
- F = PINA and PINB input logic/filtering (after PINA and PINB input selectors)
- M = pin mode
- T = pin DIR/OUT control (default = %00)
- S = smart mode

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

5.1_Example_WRD_Analog_Input

Demonstrate Analog out call object "5.1_Example_WRD_Analog_Input"

Uses following spin functions:

PINCLEAR (PinField) Clear PinField smart pin(s): DIR=0, then WRPIN=0

PINFLOAT (PinField) Float PinField pin(s)

LONGMOVE (Dest, Source, Count) Move Count longs from Source to Dest

LONGFILL (Dest, Value, Count) Fill Count longs at Dest with Value

RD PIN(Pin) :Zval Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN Data

6.0) AnalogIn and AnalogOut Demo

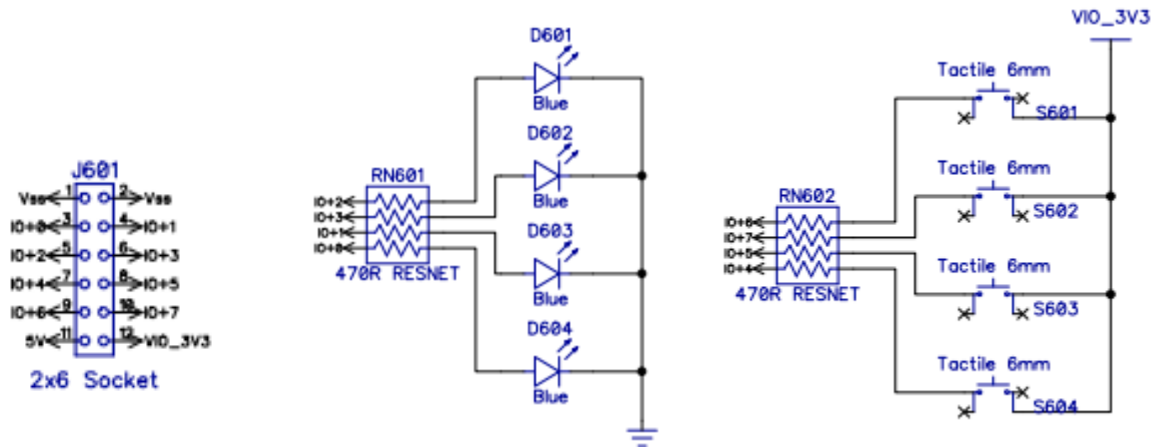
Using P24 as an analog Input fed from P25 as an analog output a 25 K load resistor is tied to Pins and ground. Two objects are included “4.1_Example_WRD_analog_output.spin2” and “5.1_Example_WRD_analog_input.spin2”. These programs were previously used in section 4.0 and 5.0.

6.1_Example_WRD_AnalogIn_AnalogOut_Demo

Analog out is fed to Analog In and displayed

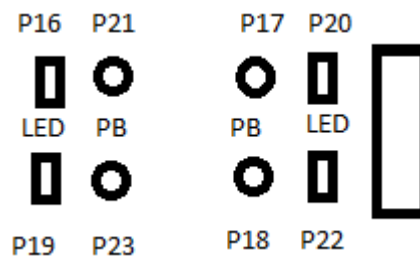
```
analogInP24 = 24      'P24 analog pin input
analogInP24_LO = 0     'Lo range analog out
analogInP24_HI = 3300  'Hi range analog out
analogOutPinP25 = 25   'P25 20k Load Resistor
analogOutP25_LO = 0    'Lo range analog out
analogOutP25_HI = 3300 'Hi range analog out
```

7.0) P2 Eval PB/LED Control Add-on Board (64006-ES)



Led_P16 Led_P17 Led_P18 Led_P19

Pb_P20 Pb_21 Pb_22 Pb_23 Pb-24



```
wrpin (20 , P_LOW_15K)
result0:=pinread(20)
pinhigh(17)
```

```
'select P20 pull-down enable sets pin as a sink
'read status of P20 button press
'set P17 led High = 1 (out high enable on)
```

“7.1_Example_WRD_Control_Board_01” and “7.2_Example_WRD_Control_Board_02” illustrates how to press input button to turn on the corresponding LED using two different methods.

[7.1_Example_WRD_Control_Board_01](#)

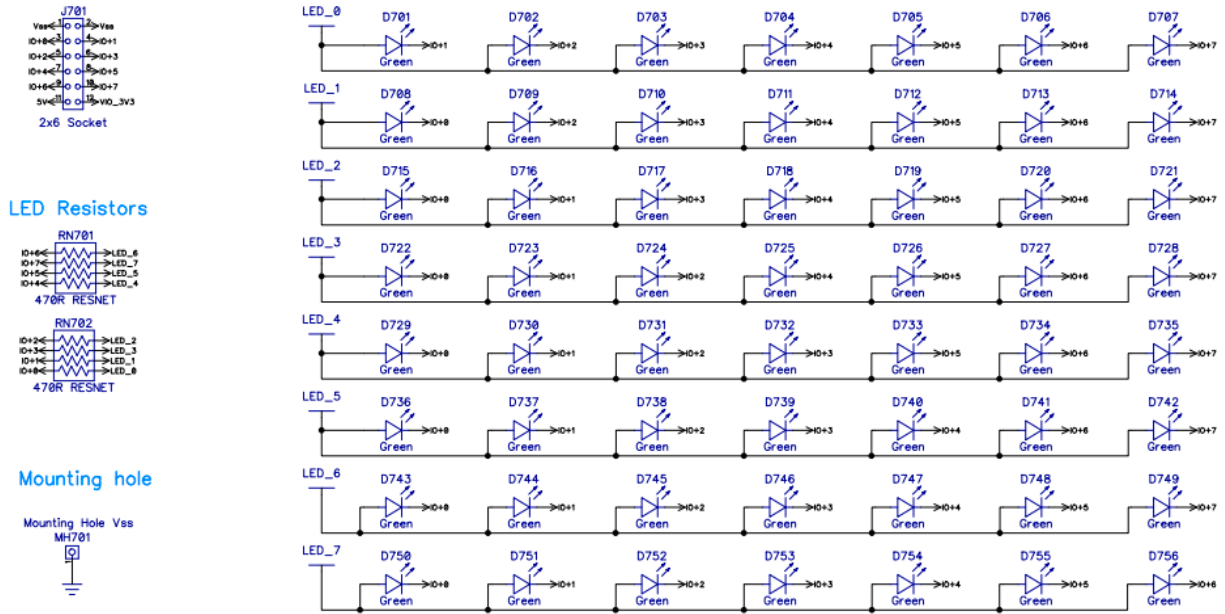
Demonstrate PB\LED Control Board64006-FS

```
result0:=pinread(Base_Pin + 4)  'read status of P20 button press  set P17 LED
result1:=pinread(Base_Pin + 5)  'read status of P21 button press  set P16 LED
result2:=pinread(Base_Pin + 6)  'read status of P22 button press  set P18 LED
result3:=pinread(Base_Pin + 7)  'read status of P23 button press  set P19 LED
```

[7.2_Example_WRD_Control_Board_02](#)

Demonstrate PinField and PINWRITE command

8.0) P2 Eval LED Matrix Add-on Board (#64006C)

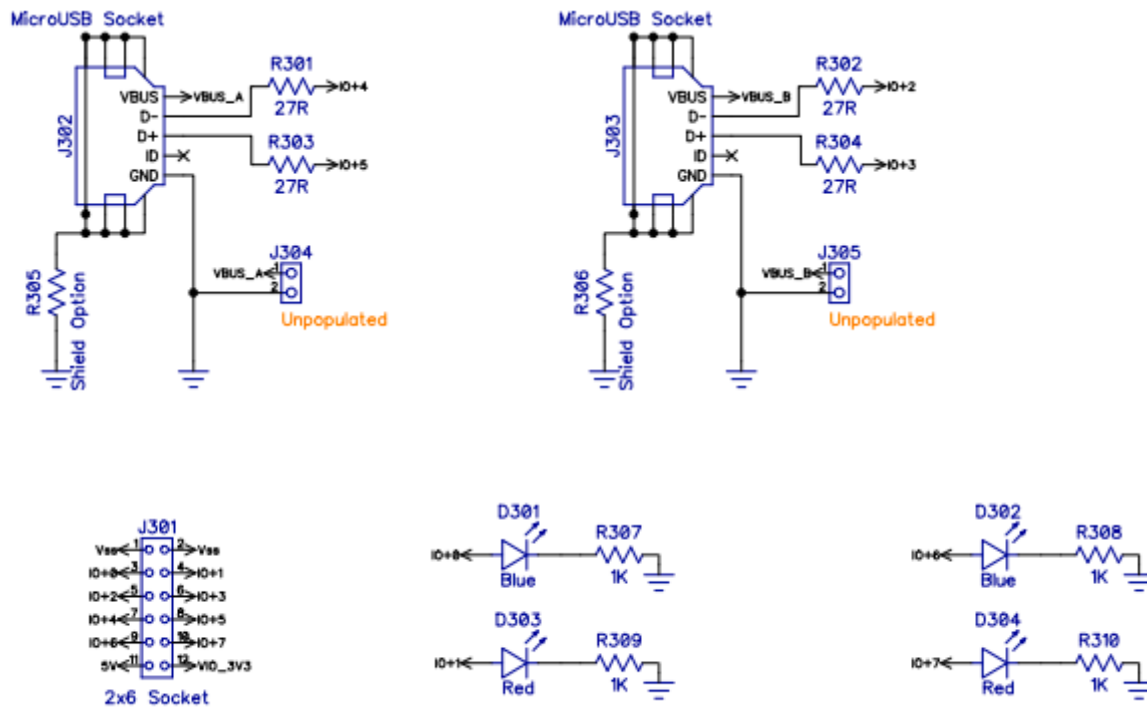


8.1_Example_WRD_P2_LED_Matrix_Digits

Demonstrate LED MATRIX by displaying digits 0-9

9.0) P2 Eval Serial Device Add-on Board (SKU 64006F)

Serial Device (slave) sockets



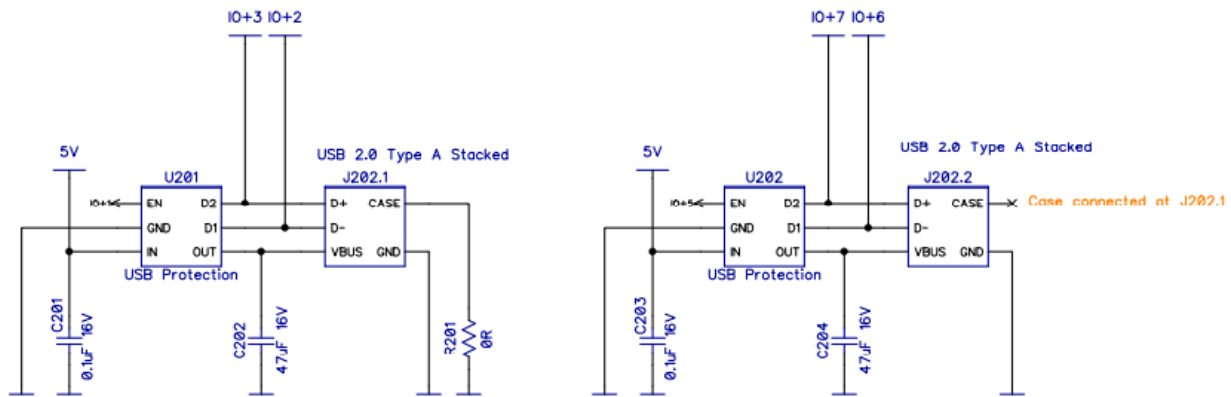
Two user controlled activity LEDs (red and blue) are located beside each microUSB-type socket.
Copyright © Parallax Inc. P2 Eval Add-on Boards (#64006 Series)

Function

- 0 Blue LED with 1 kΩ series resistor. Assert high to light.
- 1 Red LED with 1 kΩ series resistor. Assert high to light.
- 2 Serial channel 1 : Data D-
- 3 Serial channel 1 : Data D+
- 4 Serial channel 2 : Data D-
- 5 Serial channel 2 : Data D+
- 6 Blue LED with 1 kΩ series resistor. Assert high to light.
- 7 Red LED with 1 kΩ series resistor. Assert high to light

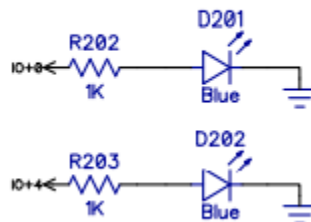
10.0) P2 Eval Serial Host Add-on Board (SKU 64006B)

Serial Host (master) - dual USB type socket



Edge expansion socket

Activity LEDs



10.1) USB host overview

The Universal **Serial** Bus, or **USB**, is an external **port** that interfaces between external **devices** and a computer.

When a port is in USB host mode, it powers the bus, and enumerates connected USB devices.

- **Host:** The host is the computer or item that acts as the main element or controller for the USB system. The host has a hub contained within it and this is called the Root Hub.
- **Hub:** The hub is a device that effectively expands the number of ports available - it will have one connection to the upstream connection, and several downstream. It is possible to plug one hub into another to expand the capability and connectivity further.
- **Port:** This is the socket through which access to the USB network is gained. It can be on a host, or a hub.
- **Function:** These are the peripherals or items to which the USB link is connected. Mice, keyboards, Flash memories, etc, etc.
- **Device:** This term is collectively used for hubs and functions.

10.2) Selecting USB function destination

With data for all devices being sent along the bus, it is necessary for the USB operation that the data is only accepted by the required function.

To achieve this, when a device is attached to the bus it is assigned a unique number or address by the host for the time it is connected.

In addition to the address, the device also contains endpoints. These are the actual sources and destinations for communications between the host and the device.

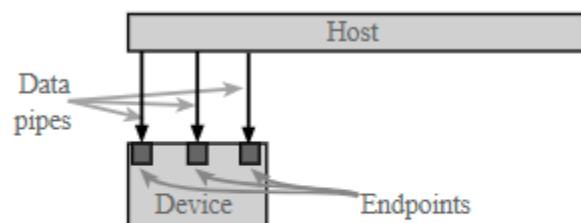
Endpoints can only operate in one direction, i.e. input or output, but not both, and devices can have up to 16, of which one each for the input and output must be reserved as the 'Zero Endpoint' for that direction. Although each device can have sixteen input and sixteen output endpoints, it is very rare for them all to be used.

The zero endpoints are used for a variety of activities including auto-detection and configuration of the device on the bus and the two zero endpoints are the only ones accessible until the device is properly connected on the bus.

10.3) USB data pipes

The communication within USB is based around the concept of using data pipes. These can be considered as being logical channels within the data flow on the bus.

In reality, a USB data pipe is a connection from the host controller to a logical entity within a device, i.e. the endpoint. Because pipes correspond to endpoints, the terms are sometimes used interchangeably.



The host then uses the concept of a data pipe to ensure the data to and from a device is correctly directed or the source is known. The data pipe uses a combination of the address, endpoint and also the direction to define it.

To communicate with the zero endpoints a special form of data pipe is needed because it needs to be used to establish the initial communication. It is called the Default Control Pipe and it can be used when the initial physical connection is made.

There are two types of USB pipe:

- **Message Pipe :** This type is a bi-directional USB pipe and it is used for control data. Message pipes are typically used for short, simple commands to the device, and for status responses from the device. They can be used by the bus control pipe number 0.
- **Stream Pipe:** This form of USB pipe is uni-directional and it is connected to a uni-directional endpoint that transfers data using an isochronous, interrupt, or bulk transfers (see below).
- h

10.4) USB signalling and data transfer basics

For USB 1 and 2 a four wire system is employed. As detailed elsewhere, the cables carry: power, ground and then there is a twisted pair for the differential data transfer.

The lines are designated Data+, D+ and Data-, D- for USB 1 and USB 2. For USB 3, new lines were introduced. For each port there are TX1+ & TX1- and TX2+ & TX2- to cover the transmitted data and then for the received data the lines are RX1+ & RX1- and RX2+ & RX2-.

The use of twisted pairs and differential signaling reduces the effects of external interference that may be picked up. It also reduces the effect of any hum loops, etc that could cause issues. As it is not related to ground, but the difference between the two lines, the effects of hum are significantly reduced.

The data uses an NRZI system, i.e. non-return to zero. In terms of operation, when the USB host powers up, it polls each of the slave devices in turn.

The USB host has address 0, and then assigns addresses to each device as well as discovering the slave device capabilities in a process called enumeration. [Enumeration takes place when a new device is connected].

Transactions between the host and device comprise a number of packets. As there are several different types of data that can be sent, a token indicating the type is required, and sometimes an acknowledgement is also returned.

Each packet that is sent is preceded by a sync field and followed by an end of packet marker. This defines the start and end of the packet and also enables the receiving node to synchronize properly so that the various data elements fall into place.

There are four basic types of data transaction that can be made within USB.

Control: This type of data transaction within the overall USB protocol is used by the host to send commands or query parameters. The packet lengths are defined within the protocol as 8 bytes for Low speed, 8-64 bytes for Full, and 64 bytes for High Speed devices.

Interrupt: The USB protocol defines an interrupt message. This is often used by devices sending small amounts of data, e.g. mice or keyboards. It is a polled message from the host which has to request specific data of the remote device.

Bulk: This USB protocol message is used by devices like printers for which much larger amounts of data are required. In this form of data transfer, variable length blocks of data are sent or requested by the Host. The maximum length is 64-byte for full speed Devices or 512 bytes for high speed ones. The data

integrity is verified using cyclic redundancy checking, CRC and an acknowledgement is sent. This USB data transfer mechanism is not used by time critical peripherals because it utilises bandwidth not used by the other mechanisms.

Isynchronous: This form of data transfer is used to stream real time data and is used for applications like live audio channels, etc. It does not use and data checking, as there is not time to resend any data packets with errors - lost data can be accommodated better than the delays incurred by resending data. Packet sizes can be up to 1024 bytes.

The data transfer methodology and protocol for USB provides an effective method of transferring the data across the interface in an effective and reliable manner.

10.5) USB data packets

Within the USB system, there are four different types of data packets each used for different types of data transfer.

Token Packets: Essentially a Token USB data packet indicates the type of transaction is to follow.

Data Packets: The USB data packets carry the payload data, carrying the data as required.

Handshake Packets: The handshake packets are used acknowledging data packets received or for reporting errors, etc.

Start of Frame Packets: The Start of Frame packets used to indicate the start of a new frame of data.

Although USB has developed from USB 1 through USB 2 to USB 3 and now USB 4, it still utilizes the same basic approach to data transfer. There are many USB connectors and leads available, and these leads now have many more wires for higher rate data transfer. Accordingly the data transfer speeds have increased many fold over the first USB specification that was released and the devices that were available.

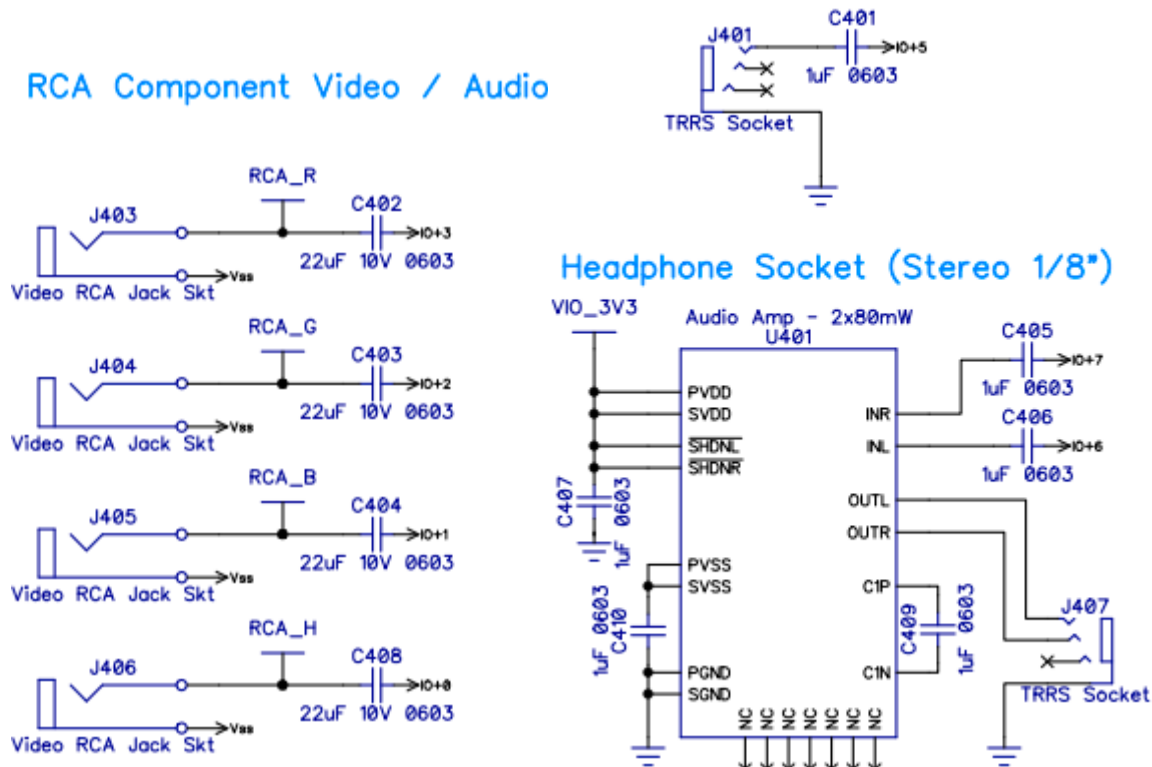
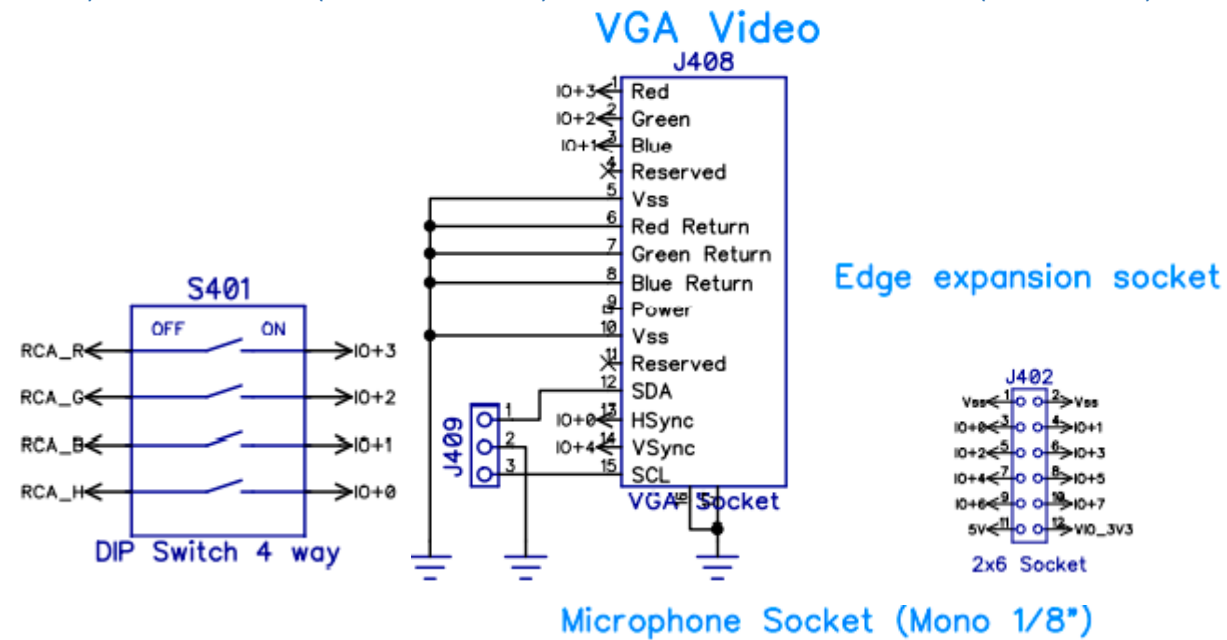
10.6) USB 3 capabilities

The USB 3.0, Superspeed and 3.1, Superspeed+ specifications enable much higher rates of data transfer. This is in keeping with requirements for downloading video and many other applications.

PERFORMANCE FIGURES FOR USB 3
I.E. USB3.0 AND USB 3.1

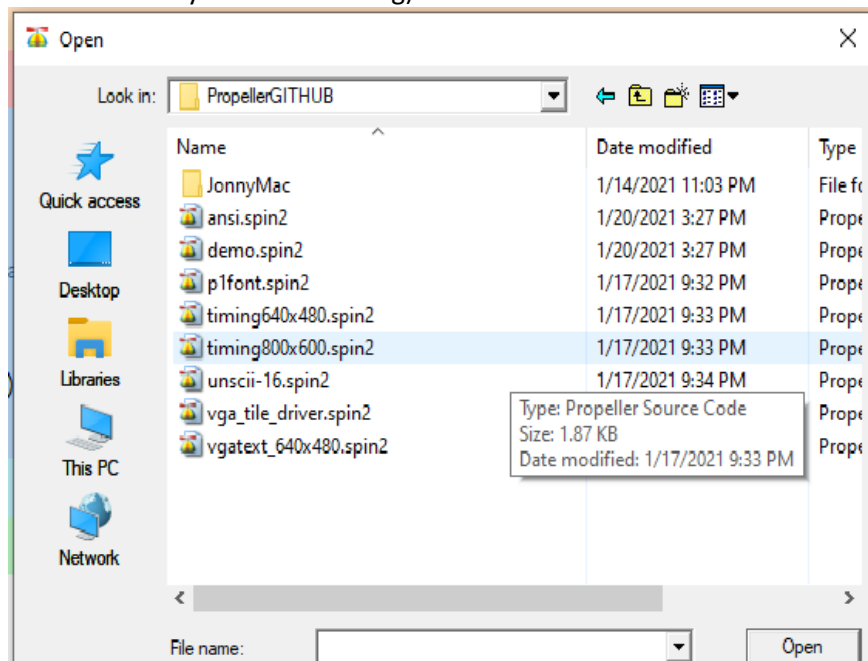
USB VERSION	DUPLEX STATUS	TRANSFER SPEED	INCREASE OVER USB 2.0
USB 2.0	Half Duplex	480 Mbps	---
USB 3.0 - Superspeed	Full Duplex	5 Gbps	10 x
USB 3.1 - Superspeed+	Full Duplex	10 Gbps	20 x

11.0) P2 Eval A/V (Audio/Video) Breakout Add-on Board (#64006H)



11.1) VGA Object (ansi_vgatest_demo.spin2)

Using the library file from the “Propeller Tool”(ansi_vgatest_demo.spin2) for VGA output (No resistor wire Pins directly to VGA Test Plug)



Set clock to 200_000_000 screen size to 600x800 set “CELL_SIZE = 4” call demo.spin2.

VGA Connector (No dropping resistors uses DAC to generate signal)

Pin- Function- VGA Pin - Wirecolour (VGA Test Plug)

P48	H	13	Torquise-Yel
P49	B	3	RED-yel
P50	G	2	BROWN-yel
P51	R	1	BLACK-yel
P52	V	14	BLUE/White-yel
GND		5,6,7,8,10	purple blue green yellow grey –blk

12.0 COG Overview

The Propeller 2 contains eight (8) processors, called cogs, numbered 0 to 7. Each cog contains the same components, including a Processor block, Cog RAM, Event Tracker, Cog Attention strobes, Streamer, Colorspace Converter, Pixel Mixer, DAC Channels, an I/O Output Register, and an I/O Direction Register. Each cog is designed exactly the same and can run tasks independently from the others.

All eight cogs are driven from the same clock source, the [System Clock](#), so they each maintain the same time reference and all active cogs execute instructions simultaneously. They also all have access to the same [shared resources](#), like I/O pins, Hub RAM, the System Counter, and CORDIC math solver.

Cogs can be started and stopped at-will, performing independent or cooperative tasks simultaneously. Regardless of the nature of their use, the Propeller application developer has full control over how and when each cog is employed; there is no compiler-driven or operating system-driven splitting of tasks between multiple cogs. This empowers the developer to deliver absolutely deterministic timing, power consumption, and response to the embedded application.

Any cog can start or stop any other cog, or restart or stop itself. Each of the eight cogs has a unique three-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.

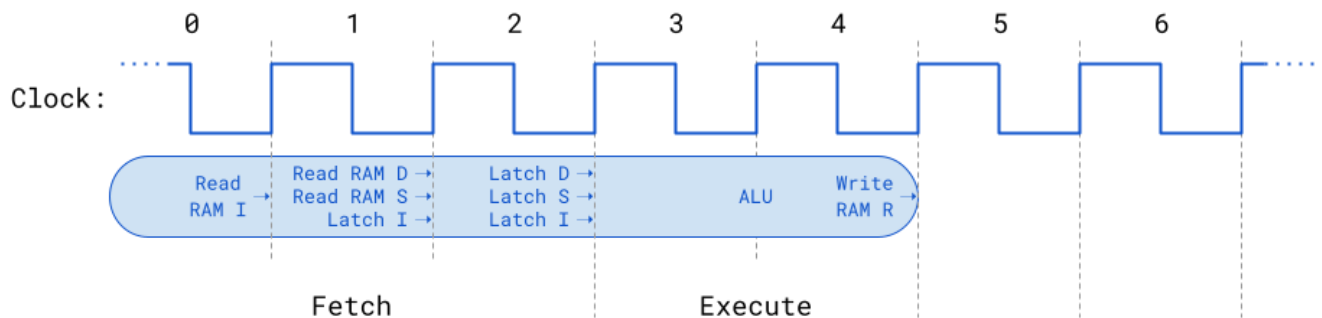
Instruction Pipeline

To optimize execution speed, cogs employ a pipelined execution architecture for PASM2. The nature of the pipeline is summarized by these attributes:

- There are five stages of processing per instruction, performed in a minimum of five clock cycles.
- Instructions are overlapped to effectively execute in as little as two clock cycles when the pipeline is full.
- Branch instructions cause the pipeline to be flushed; the first instruction following the branch will take at least five clock cycles (13 or 14 if branching to a hub address) since the pipeline is refilling.
- Any instruction that is conditionally cancelled will not execute but will still take effectively two clocks (or at least five clocks, if following a branch) to pass through the pipeline.
- If an instruction stalls for additional clock cycles, all following instructions in the pipeline are also stalled.

An instruction's five stages of processing are illustrated below.

Isolated Instruction Processing



I = Instruction opcode

D = Destination operand

S = Source operand

ALU = Arithmetic Logic Unit, i.e. Adder

R = Result of instruction execution; ALU output value including C and Z flags

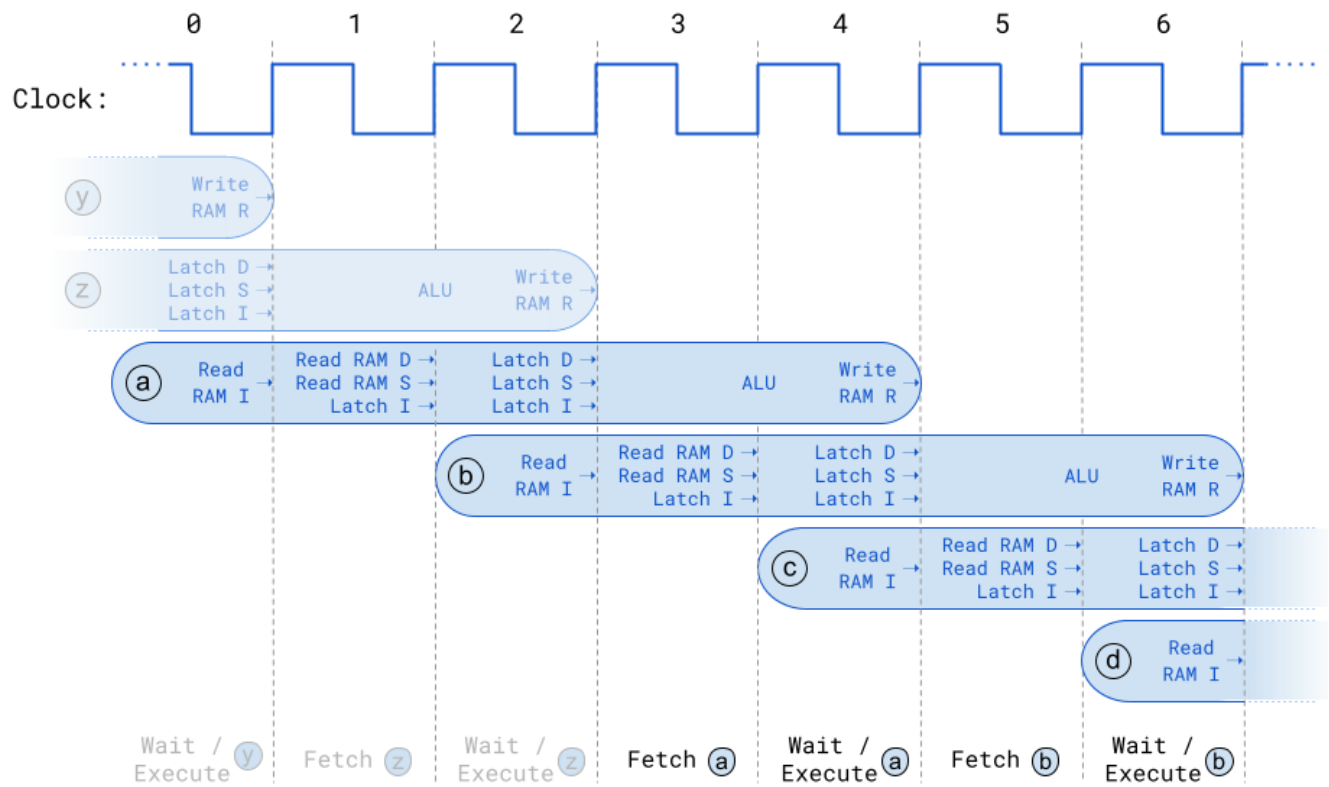
Note that most stages are performed (completed) upon the rising edge of the following clock signal, indicated by the arrows (→).

The first three stages ("fetch" phase) involve reading the 32-bit PASM2 instruction (I) opcode from RAM, latching (saving) the instruction opcode for decoding, and reading/latching the instruction's source (S) and destination (D) values (32-bits each). The final two stages ("execute" phase) perform the instruction's intended operation via the arithmetic logic unit (ALU) and writes the resulting 32-bit value as well as the carry and zero flags if required. At that point (five clock cycles in this case) the instruction is fully executed.

- Each individual result value, carry, and zero flag outcome is either written to RAM or is discarded, depending on the specific instruction and given effects ($WC / WZ / WCZ$)
- As needed for proper processing, an instruction may stall (i.e. one or more extra clock cycles may be inserted, without any stage advancement) at any point in the instruction's five-stage journey

In the pipeline, instructions are overlapped by three stages, resulting in an effective two-cycle execution per instruction when the pipeline is full; known as Fetch and Execute (or Wait). Compare the instruction from the illustration above with the multi-instruction pipeline flow below – this seven-cycle slice of time is processing six contiguous instructions. Each stage of the instruction above appears in the next illustration with the prefix "a" and other instructions use prefixes "b", "c", etc.

Instruction Pipeline Flow



Ix = Instruction opcode

Dx = Destination operand

Sx = Source operand

ALUx = Arithmetic Logic Unit, i.e. Adder

Rx = Result of instruction execution; ALU output value including C and Z flags

a, b, c, etc. = [Suffixes] Process path of instruction a, instruction b, instruction c, etc.

In typical operation, each cycle of the instruction pipeline simultaneously processes different stages of 2 or 3 contiguous instructions. In this example, instruction "a" is read in the first cycle which is the same moment its two previous instructions ("y" and "z") write results and latch for later ALU operation, respectively.

When an instruction **requires a resource** that is not yet available (such as Hub RAM), the whole pipeline waits (halts temporarily) before the instruction's Execute phase, for **as many clock cycles** as it takes for that resource. Afterwards, processing continues again for all instructions in the pipeline. For example, if instruction "a" needed to wait 2 extra cycles to execute properly, the pipeline flow (above) would be stretched starting at cycle 4— instead of "Execute a," two "Wait" cycles would occur, delaying the ALU (and subsequent Write) for instruction 'a' as well as the latching and reading stages of instructions "b," "c," and "d."

Locks (Semaphores)

For application-defined cog coordination, the hub provides a pool of 16 semaphore bits, called locks. Cogs may use locks, for example, to manage exclusive access of a resource or to represent an exclusive state, shared among multiple cogs. What a lock represents is completely up to the application using it; they are a means of allowing one cog at a time the exclusive status of 'owner' of a particular lock ID. In order to be useful, all participant cogs must agree on a lock's ID and what purpose it serves.

The LOCK instructions are:

```
LOCKNEW    D {WC}
LOCKRET    {#}D
LOCKTRY    {#}D {WC}
LOCKREL    {#}D {WC}
```

Lock Usage

In order to use a lock, one cog must first allocate a lock from the lock pool with `LOCKNEW` and communicate that lock's ID with other cooperative cogs. If successful, `LOCKNEW` returns the lock ID in D and, if WC is given, will clear C (0) if a lock was available or set C (1) if all locks were already allocated. A cog may allocate more than one lock if needed.

Cooperative cogs then use `LOCKTRY` to *take* ownership of the state which that lock represents. The Hub arbitrates lock ownership in a round-robin fashion (as with all exclusive resources) so any cog waiting to take ownership of a lock will get its fair turn and only one will be awarded ownership at any given time. Here's an example of looping until ownership of a lock is successful:

```
'Keep trying to capture lock until successful
.try          LOCKTRY write_lock WC
              IF_NC    JMP #.try
```

Once lock ownership is successful, the cog should perform the task the lock was designed to protect while all other cogs in this cooperative arrangement should be busy with other tasks or waiting for lock ownership approval in a loop similar to the above. It is recommended that lock-protected steps be intentionally swift so as not to hold up other cogs waiting for ownership to perform their lock-protected counter steps.

After the designated task is performed, the cog must immediately use `LOCKREL` to *release* ownership of the lock; allowing other cogs potential ownership of the lock. Only the cog that has taken ownership of the lock can release it; however, a lock will also be implicitly released if the cog that's holding ownership is stopped (`COGSTOP`), restarted (`COGINIT`), or if `LOCKRET` is executed for that lock.

If the lock is no longer needed by the application (i.e. no cogs need it for the designed purpose), it may be returned to the unallocated lock pool by executing `LOCKRET`. Any cog can return a lock, even if it wasn't the cog that allocated it with `LOCKNEW`.

12.1) Cog Components

Shared Resources

The interaction between each cog and the Hub is vital for sharing resources in the Propeller 2. At any given time, the Hub gives a specific cog momentary exclusive access to certain shared resources such as a region of Hub RAM and system configuration settings. This happens for each cog in a “round robin” fashion— timing is consistent regardless of how many cogs are running. Cogs can choose to use or ignore those resources depending on their current needs; often processing internally (in Cog RAM) in parallel and only accessing exclusive resources in bursts.

There are two types of shared resources in the Propeller 2: 1) common, and 2) exclusive. Common resources can be accessed at any time by any number of cogs; they include Smart I/O Pins, the System Counter, and the Pseudo-Random Number Generator results. Exclusive resources can also be accessed by each cog, but only by one cog at a time; they include Hub RAM, the CORDIC solver, Lock bits and the seeder functionality for the Pseudo-Random Number Generator. The Hub helps govern access to exclusive elements by granting each cog a turn to use it, one at a time, facilitating atomic operations without any contention. For cases involving multiple elements (ex: a block of Hub RAM locations) where an atomic operation is not intrinsically possible, lock bits can be used to cooperatively share access between cogs. See the [Appendix “I” Hub Operation](#) section for more information.

Common Resources

System Clock

The System Clock is the central clock source for nearly every component of the Propeller 2. All cogs and I/O pins perform their next step upon the next System Clock's clock edge. The System Clock itself is driven from one of three selectable sources: 1) the Internal RC Oscillator, 2) the Phase-Locked Loop (PLL), or 3) the Crystal Oscillator (an internal circuit that operates an external crystal or receives an external oscillator signal). The PLL uses the Crystal Oscillator as its reference clock input. The System Clock source is selected by the CLK register setting, which is configurable both at compile time and at run time.

The System Clock speed chosen for any Propeller application is of vital importance to timing calculations in code. If coded properly via the clock setting constants (`_clkfreq`, `_xinfreq`, `_xtlffreq`, `_rcslow`, or `_rcfast`) the compiled clock mode is reflected in `clkfreq_` and `clkmode_`. When set via the `HUBSET` or `ASMCLK` instructions, the run time `CLKFREQ` and `CLKMODE` values reflect the current System Clock speed.

See [Appendix "J" System Clock](#) for more information.

System Counter

The System Counter (CT) is a 64-bit free-running counter that increments upon every clock cycle. It is a shared resource, accessible by all cogs at any time, serving as the official time reference for many instructions and events. It is often used for brief, relative time measurements; however, since it is cleared to zero upon every power-up/reset, it is also a *system up time* reference.

To read the current System Counter value:

```

GETCT    X                                'read lower 32-bits of system counter into X register
--or--
GETCT    X      WC                        'read upper 32-bits of system counter into X register
GETCT    Y                                'read lower 32-bits of system counter into Y register

```

Note: to get the full 64-bit System Counter value, it is important to read the upper 32-bits first (as shown above) and immediately read the lower 32-bits second. This sequence employs a special mechanism that avoids phase issues; CT's lower 32-bits are returned exactly as they were back at the moment in which the upper 32-bits had been read.

For event handling, there are three hidden registers dedicated to System Counter timing and events: CT1, CT2, and CT3. These represent a target moment in time (future CT value), settable via the `ADDCTx` instructions and used (read) internally by many event instructions.

To mark a moment in time to wait for, use `GETCT` with `ADDCTx` (1, 2, or 3) and `WAITCTx` (1, 2, or 3):

```

GETCT    x                                'get current CT
ADDCT1   x, #500                          'make target CT1 (500 cycles later)
WAITCT1                                     'wait for CT to pass CT1 target

```

This can easily be extended to create a 500-cycle activity-loop instead.

The event-timing instructions that utilize the System Counter are: `ADDCTx`, `POLLCTx`, `WAITCTx`, `JCTx`, and `JNCTx`. In addition, by using a `SETQ` right before any `WAITxxx` instruction, a *timeout* is created to abort the *wait* in case the target event never arrives.

Exclusive Resources

[Cordic Solver](#)

[Appendix "L"](#)

[Hub Ram](#)

[Appendix "I" Hub Operation](#)

LOCKS

Memory

The Propeller 2 has three memory regions: Register RAM, Lookup RAM, and Hub RAM. Each cog has its own Register RAM and Lookup RAM (collectively called Cog RAM), while the Hub RAM is shared by all cogs.

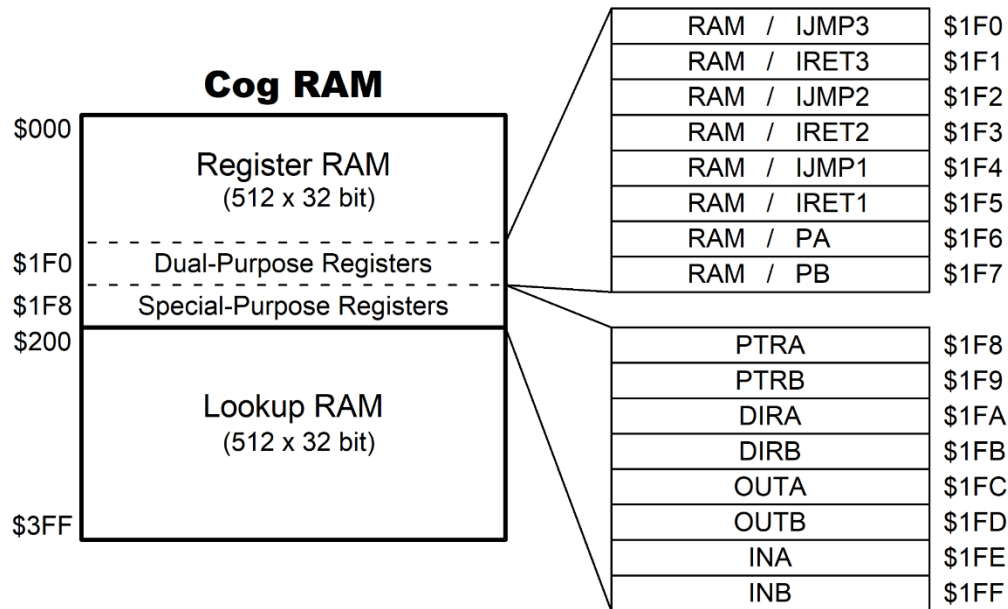
Propeller 2 (P2X8C4M64P) RAM Memory Configuration					
Region	Depth	Width	Address Range (Hex)	PASM Instruction D/S Address Range (Hex)	PC Increment ¹
Cog "Register" RAM	512	32 bits	\$00000..\$001FF	\$000..\$1FF	1
Cog "Lookup" RAM	512	32 bits	\$00200..\$003FF	\$000..\$1FF	1
Hub RAM	524,288	8 bits	\$00400..\$7FFFF	\$00000..\$7FFFF	4

¹ PC is the Program Counter for PASM execution; incrementing relative to width to retrieve 32-bit instructions.

The Spin2 interpreter fits between \$124 to \$1D7 for 172 bytes. These are registers. Also use below \$1D7.

Cog Memory

Each cog has its own internal RAM that it uses to execute code and to store and manipulate data independent of every other cog. This internal RAM is organized into two contiguous blocks of 512 longs (512 x 32), called Register RAM and Lookup RAM, each with special attributes. See [RAM Memory Configuration](#).



Note that \$1FE (*INA*) and \$1FF (*INB*) are also the debug interrupt call address and return address, respectively.

Register RAM

Each cog's primary 512 x 32-bit dual-port Register RAM (Reg RAM for short) provides for code execution, fast direct register access, and special use. It is read and written as longs (4 bytes) and contains general purpose, dual-purpose, and special-purpose registers.

General Purpose Registers

Register RAM locations \$000 through \$1EF are general-purpose registers for code and data usage. Register unnamed are from \$000-\$1D7:

Address	Name	Purpose
\$1D8	PR0	communication
\$1D9	PR1	
\$1DA	PR2	
\$1DB	PR3	
\$1DC	PR4	
\$1DD	PR5	
\$1FE	PR6	
\$1DF	PR7	

Note : It is thought at this time spin 2 uses \$1E0 to \$1EF and should be avoided could be used in PASM

Dual-Purpose Registers

Register RAM locations \$1F0 through \$1F7 may either be used as general-purpose registers, or may be used as special-purpose registers if their associated functions are enabled.

Address	Name	Purpose
\$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

Special-Purpose Registers

RAM registers \$1F8 through \$1FF give mapped access to eight special-purpose functions. In general, when specifying an address between \$1F8 and \$1FF, the PASM2 instruction accesses a special-purpose register, *not* just the underlying RAM.

Address	Name	Purpose
\$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 ¹	Input states for P31..P0
\$1FF	INB ²	Input states for P63..P32

¹Also debug interrupt call address

²Also debug interrupt return address

Lookup RAM

Each cog's secondary 512 x 32-bit dual-port Lookup RAM (LUT RAM for short) is read and written as longs (4 bytes). It is useful for:

- Scratch space
- Streamer access
- Bytecode execution lookup table
- Smart pin data source
- Paired-Cog communication mechanism
- Code execution

Scratch Space

In contrast to Register RAM, the cog cannot directly reference Lookup RAM locations in the majority of its PASM instructions. Instead, the desired location(s) must be read or written between Lookup RAM and Register RAM using the `RDLUT` and `WRLUT` instructions, respectively. This is synonymous with other hardware architecture's scratch storage using "LOAD" and "STORE" instructions. When using the `RDLUT` and `WRLUT` instructions, the Lookup RAM's locations \$200..\$3FF are addressable as \$000..\$1FF.

Paired-Cog Communication Mechanism

Adjacent cogs whose ID numbers differ by only the LSB (cogs 0 and 1, 2 and 3, 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.

Warning: Lookup RAM writes from the adjacent cog are implemented on the Lookup RAM's 2nd port. The 2nd port is also 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.

To use this feature, start two adjacent cogs using a special mechanism of the `COGINIT` instruction, enable the feature with the `SETLUTS` instruction, and if needed, facilitate handshaking between cogs using the `SETSEL..4` instructions.

Execution

Cogs use 20-bit addresses for their program counters (PC); the upper bit is a "don't care" bit - this affords an execution space of up to 512 KB. Depending on the value of a cog's PC, an instruction will be fetched from either its Register RAM, its Lookup RAM, or the Hub RAM. See [RAM Memory Configuration](#).

Register Execution

When the PC is in the range of \$00000 to \$001FF, the cog fetches instructions from Cog Register RAM. This is referred to as "cog execution." There are no special considerations when branching to a cog register address.

Lookup Execution

When the PC is in the range of \$00200 to \$003FF, the cog fetches instructions from Cog Lookup RAM. This is referred to as "lut execution." There are no special considerations when branching to a cog lookup address.

Hub Execution

When the PC is in the range of \$00400 to \$7FFFF, the cog fetches instructions from Hub RAM. This is referred to as "hub execution mode." Special considerations are involved with hub execution.

1. The PC rolling beyond \$003FF will not initiate hub execution (it will just wrap back to \$00000); a branch must occur to get from register or lookup execution to hub execution.
2. Branching to a hub address takes a minimum of 13 clock cycles. If the instruction being branched to is not long-aligned, one additional clock cycle is required.
3. When executing from Hub RAM, the cog employs the FIFO hardware to spool up instructions so that a stream of instructions will be available for continuous execution. This means the FIFO cannot be used for anything else. So, during hub execution these instructions cannot be used:

RDFAST / WRFAST / FBLOCK
 RFBYTE / RFWORD / RFLONG / RFVAR / RFVARS
 WFBYTE / WFWORD / WFLONG
 XINIT / XZERO / XCONT - when the streamer mode engages the FIFO

It is not possible to execute code from hub addresses \$00000 through \$003FF, as the cog will instead read instructions from the cog's Register RAM or Lookup RAM as indicated above.

CogInit\CogStop

In **Spin2** the Cogspin(CogNum,Spin_Method(<(parameters)>,@stack) instruction is used to start cog running a **“spin method”**

In **PASM** the Coginit(CogID,AsmAddress,Paramater) instruction is used to start cog to run a “**PASM**” program

COGSPIN(CogNum, Method({Pars}), StkAddr)	Start Spin2 method in a cog, returns cog's ID if used as an expression element, -1 = no cog free
COGINIT(CogNum, PASMaddr, PTRAvalue)	Start PASM code in a cog, returns cog's ID if used as an expression element, -1 = no cog free
COGSTOP(CogNum)	Stop cog CogNum
COGID() : CogNum	Get this cog's ID
COGCHK(CogNum) : Running	Check if cog CogNum is running, returns -1 if running or 0 if not

Starting And Stopping Cogs

Any cog can start or stop any other cog, or restart or stop itself. Each cog has a unique ID which can be used to start or stop it. It is also possible to start free (stopped or never started) cogs, without needing to know their IDs. This way, applications can 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 to become available again for restarting.

To start a free cog:

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

To (re)start a specific cog:

COGINIT #1, # \$100	'load and start cog 1 from hub address \$100
---------------------	--

To start a cog, passing in a pointer or 32-bit value:

SETQ ptr val 'ptr val will go into target cog's PTRA register

```
COGINIT #%0 1 0000, addr 'load and start a free cog at addr
```

To retrieve this cog's ID:

COGID myID 'my cog ID is written to myID

To stop this cog:

```
COGID      myID      'get my ID
```

COGSTOP	myID	'halt myself
---------	------	--------------

REGLOAD and REGEXEC calling PASM

The Spin2 instructions **REGLOAD(HubAddress)** and **REGEXEC(HubAddress)** are used to load or load-and-execute PASM code and/or data chunks from hub RAM into cog registers.

The chunk of PASM code and/or data must be preceded with two words which provide the starting register and the number of registers (longs) to load, minus 1.

```
PUB go()
  REGLOAD(@chunk) 'load self-defined chunk from hub into registers
  REPEAT
    CALL(#start) 'call program within chunk at register address
    WAITMS(100)
  DAT
  chunk WORD start,finish-start-1 'define chunk start and size-1
    ORG $120 'org can be $000..$130-size
  start DRVRND #56 ADDPINS 7 'some code
  _RET_ DRVNOT #0 'more code + return
  finish
```

REGEXEC works like REGLOAD, but it also CALLs to the start register of the chunk after loading it.

In the example below, REGEXEC launches a chunk of code in upper register memory which sets up a timer interrupt and then returns to Spin2. Meanwhile, as the Spin2 method repeatedly randomizes pins 60..63 every 100ms, the chunk of code loaded into upper register memory perpetuates the timer interrupt and toggles pins 56..59 every 500ms. Note that registers \$000..\$127 are still free for other code chunks and interrupts 2 and 3 are still unused.

```
PUB go()

  REGEXEC(@chunk) 'load self-defined chunk and execute it
                  'chunk starts timer interrupt and returns

  REPEAT
    PINWRITE(60 ADDPINS 3, GETRND()) 'randomize pins 60..63
    WAITMS(100) 'pins 56..59 toggle via interrupt
  DAT
  chunk WORD start,finish-start-1 'define chunk start and size-1
    ORG $128 'org can be $000..$130-size
  start MOV IJMP1,#isr 'set int1 vector
    SETINT1 #1 'set int1 to ct-passed-ct1 event
    GETCT PRO 'get ct
  _ret_ ADDCT1 PRO,bigwait 'set initial ct1 target, return to Spin2
  isr DRVNOT #56 ADDPINS 3 'interrupt service routine, toggle 56..59
    ADDCT1 PRO,bigwait 'set next ct1 target
    RETI1 'return from interrupt
  bigwait LONG 20_000_000 / 2 '500ms second on RCFAST
  finish
```

Cog Attention

Each cog can request the attention of other cogs by using the `COGATN` instruction. One or more of the D operand's lower 8 bits may be set high (1) to signal the corresponding cog or cogs.

```
COGATN    #00001100    'Get attention of cogs 2 and 3
```

For each high bit, the matching cog sees an *attention* event for `POLLATN` / `WAITATN` / `JATN` / `JNATN` and for interrupt use. The attention strobe outputs from all cogs are OR'd together to form a composite set of 8 strobes from which each cog receives its particular strobe.

Examples:

<code>POLLATN</code>	<code>WC</code>	'has attention been requested?
<code>WAITATN</code>		'wait for attention request
<code>JATN</code>	<code>addr</code>	'jump to addr if attention requested
<code>JNATN</code>	<code>addr</code>	'jump to addr if attention not requested

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

Pseudo-Random Number Generator

The Propeller 2 features a pseudo-random number generator (PRNG) based on the Xoroshiro128** algorithm. Note that the "**" is part of the name, indicating the exact variation of the Xoroshiro128 algorithm used.

The Xoroshiro128** PRNG iterates on every clock cycle, generating 64 fresh bits which get spread among all cogs and smart pins. From this 64-bit pool, upon every clock cycle, each cog receives a unique set of 32 different bits (in a scrambled arrangement with some bits inverted) and each smart pin receives a similarly-unique set of 8 different bits. Cogs can read their current 32-bit *pseudo-random* value 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.

After reset, the bootloader seeds 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
```

Note: using `HUBSET`, with D's MSB set, will seed the 128-bit PRNG. This will write all bits of D into 32 bits of the PRNG, affecting 1/4th of its total state. The required high MSB bit in D 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.

Note there is also another pseudo-random number feature, accessed via the `XOR032` instruction; however it doesn't use the Xoroshiro128** PRNG— instead, it iterates a register value to make a relatively good PRNG sequence under software control.

12.2) Built-In Symbols for COGINIT Usage

COGINIT Symbol Value	Symbol Name	Details
%00_0000	COGEXEC (default)	Use "COGEXEC + CogNumber" to start a cog in cogexec mode
%10_0000	HUBEXEC	Use "HUBEXEC + CogNumber" to start a cog in hubexec mode
%01_0000	COGEXEC_NEW	Starts an available cog in cogexec mode
%11_0000	HUBEXEC_NEW	Starts an available cog in hubexec mode
%01_0001	COGEXEC_NEW_PAIR	Starts an available eve/odd pair of cogs in cogexec mode, useful for LUT sharing
%11_0001	HUBEXEC_NEW_PAIR	Starts an available eve/odd pair of cogs in hubexec mode, useful for LUT sharing

12.3) Built-In Symbol for COGSPIN Usage

COGINIT Symbol Value	Symbol Name	Details
%01_0000	NEWCOG	Starts an available cog

12.4) PASM Propeller Assembly Machine Language

PASM stands for propeller assembly language program. PASM can be inline with spin2 code or called and loaded separately. There are two different languages PASM for propeller 1 (32 I/O) and PASM for propeller 2 (64 I/O). Most of the instructions are similar but are not 100% equivalent.

The boot procedure requires spin code to be initiated. The spin interpreter then can be used to launch PASM code. The propeller 1 and propeller 2 do not operate in the same manner concerning assembly code and hub access.

12.5) In Line PASM Code

Spin2 methods can execute in-line PASM code by preceding the PASM code with an '**ORG** {\$000..\$12F}' and terminating it with an **END**.

```
PUB go() | x
  repeat
    org
      getrnd wc    'rotate a random bit into x
      rcl  x,#1
    end
    pinwrite(56 addpins 7, x)  'output x to the P2 Eval board's LEDs
    waitms(100)
```

Your PASM code will be assembled with a **RET** instruction added at the end to ensure that it returns to Spin2, in case no early **_RET_** or **RET** executes.

Here's the internal Spin2 procedure for executing in-line PASM code:

- Save the current streamer address for restoration after the PASM code executes.
- Copy the method's first 16 long variables, including any parameters, return values, and local variables, from hub RAM to cog registers \$1E0..\$1EF.
- Copy the in-line PASM-code longs from hub RAM into cog registers, starting at the **ORG** address (default is \$000).

- CALL the PASM code.
- Restore the 16 longs in cog registers \$1E0..\$1EF back to hub RAM, in order to update any modified method variables.
- Restore the streamer address and resume Spin2 bytecode execution.

Within your in-line PASM code, you can do all these things:

- Read and write the following register areas:
 - \$000..\$12F, which your PASM code loads into. You can even load different PASM programs at different addresses within this range and CALL them from Spin2.
 - **\$1D8..\$1DF**, which are general-purpose registers, **named PR0..PR7**, available to both PASM and Spin2 code.
 - \$1E0..\$1EF, which temporarily contain the method's first 16 long hub RAM variables and are temporarily assigned the same symbolic names.
 - \$1F0..\$1FF, which include IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTRB, DIRA, DIRB, OUTA, OUTB, INA, and INB.
 - Avoid writing to \$130..\$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2_interpreter.spin2" to see the interpreter code.
- Use the streamer temporarily.
- Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
- Declare and reference regular and local symbols. These symbols will not be accessible outside of your PASM code.
- Declare BYTE, WORD, and LONG data.
- Use the RES, ORGF, and FIT directives. The directives ORG, ORGH, ALIGNW, ALIGNL, and FILE are not allowed within in-line PASM code.
- Establish an interrupt which executes your code remaining in cog registers \$000..\$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
- Return to Spin2, at any point, by executing an `_RET_` or RET instruction.

12.6) Calling PASM from Spin2

You can do a **CALL(address)** in Spin2 to execute PASM code in either cog register space or hub RAM.

```
PUB go() | x
  repeat
    call(@random)
    pinwrite(56 addpins 7, pr0)
    waitms(100)
DAT  orgh  'hub PASM program to rotate a random bit into pr0
random getrnd wc
_ret_ rcl  pr0,#1
```

Here's the internal Spin2 procedure for executing a CALL:

- Save the current streamer address for restoration after the PASM code executes.
- CALL the PASM code.
- Restore the streamer address and resume Spin2 bytecode execution.

Within code which you CALL, you can do all these things:

- Read and write the following register areas:
 - \$000..\$12F, which may contain PASM code and/or data which you previously loaded.
 - \$1D8..\$1DF, which are general-purpose registers, named PR0..PR7, available to both PASM and Spin2 code.
 - \$1E0..\$1EF, which are available for scratchpad use, but will likely be rewritten when Spin2 resumes.
 - \$1F0..\$1FF, which include IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTR A, PTR B, DIR A, DIR B, OUT A, OUT B, IN A, and IN B.
 - Avoid writing to \$130..\$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2_interpreter.spin2" to see the interpreter code.
- Use the streamer temporarily.
- Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
- Establish an interrupt which executes your code remaining in cog registers \$000..\$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
- Return to Spin2, at any point, by executing an _RET_ or RET instruction.
- A symbol declared under ORGH will return its hub address when referenced.

-
- A symbol declared under ORG will return its cog address when referenced,
- but can return its hub address, instead, if preceded by '@':
-
- COGINIT #0, #@newcode
-
- For immediate-branch and LOC address operands, "#" is used before the
- address. In cases where there is an option between absolute and relative
- addressing, the assembler will choose absolute addressing when the branch
- crosses between cog and hub domains, or relative addressing when the
- branch stays in the same domain. Absolute addressing can be forced by
- following "#" with "\".

12.7) Launching Cogs with SpinMethod at StackPointer

Cogspin(CogNum, Spin_Method(<(parameters)>, @stack)

returns cogID if started and running or -1 if no cog free

Note: Followin Stack Space was for P1 assignments

Stack Space should be over estimated and then "Stack Space Tool" can be used to determine actual size of stack required

2 longs for return address

1 long for return result

1 long for each method parameter

1 long for each local variable

1 long for each intermediate concurrent calculation

12.8) Launching Cogs with Assembly Program

The COGINIT instruction is used to start cogs from a PASM program:

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 overwritten 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 overwritten with the even/lower target cog's ID. Otherwise, C will be set and D will be overwritten with \$F.

S/# = address This value is either the hub address from which the target cog will load from, or it is the cog/hub address from which the target Cog will begin executing at, depending on D[5]. This 32-bit value will be written into the target cog's PTRB register.

If COGINIT is preceded by SETQ, the SETQ value will be written into the target cog's PTRB register. This is intended as a convenient means of pointing the target cog's program to some runtime data structure or passing it a 32-bit parameter. If no SETQ is used, the target cog's PTRB register will be cleared to zero.

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

COGINIT #%1_0_0101,PTRA 'start cog 5 at PTRA

SETQ ptr_val 'ptr_val will go into target cog's PTRB register

COGINIT #%0_1_0000,addr 'load and start a free cog at addr

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

COGINIT id,addr WC '(id=\$30) start a free cog at addr, C=0 and id=cog 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

[12.1_Example_WRD_COGINIT_COGEXEC_NEW](#)

Demonstrate Launching Cog with COGINIT with COEXEC_NEW(start next available cog)

[12.2_Example_WRD_COGINIT_COGEXEC_CogID](#)

Demonstrate Launching Cog With COGINIT and COGEXEC+COG_ID(specific cog value)

[12.3_Example_WRD_COGINIT_HUBEXEC](#)

Demonstrate Launching Cog in HUB using ORGH

[12.4_Example_WRD_REGLOAD](#)

Demonstrate Running Code in same Cog as Spin

[12.5_Example_WRD_8CogSpin_Demo](#)

Demonstrates 8 cogs run separately. (1k resistor with LED P0-p7)

[12.6_Example_WRD_8CogSpin_Demo_Rev](#)

Demonstrates 8 cogs run separately. (1k resistor with LED P0-p7) different order run.

13.0) Debug for Testing and Troubleshooting

The Spin2 compiler contains a stealthy debugger program that can be automatically downloaded with your application. It uses the last 16KB of RAM plus a few bytes for each Spin2 DEBUG statement and one instruction for each PASM DEBUG statement. You place DEBUG statements in your application which contain output commands that will serially transmit the state of variables and equations as your application runs. Each time a DEBUG statement is encountered during execution, the debugger is invoked and it outputs the message for that statement. Debugging is initiated by adding the **Ctrl key to the usual F10 to 'run' or F11 to 'program'**. This compiles your application with all the DEBUG statements, adds the debugger to the download, and then brings up the DEBUG Output window which begins receiving messages at the start of your application. DEBUG can be used in Spin2 or PASM.

13.1) Things to know about the DEBUG system

- To use the debugger, you must configure at least a 10 MHz clock derived from a crystal or external input. You cannot use RCFAST or RCSLOW.
- The debugger occupies the top 16 KB of hub RAM, remapped to \$FC000..\$FFFFFF and write-protected. The hub RAM at \$7C000..\$7FFFF will no longer be available.
- Data defining each DEBUG statement is stored within the debugger image in the top 16 KB of RAM, minimizing impact on your application code.
- In Spin2, each DEBUG statement adds three bytes, plus any code needed to reference variables and resolve run-time expressions used in the DEBUG statement.
- In PASM, each DEBUG statement adds one instruction (long).
- DEBUG statements are ignored by the compiler when not compiling for DEBUG mode, so you don't need to comment them out when debugging is not in use.
- If no DEBUG statements exist in your application, you will still get notification messages when cogs are started.
- Debugging is invoked by pressing the CTRL key before the usual F9..F11 keys, which compile, download, and program to flash.
- During execution, as DEBUG statements are encountered, text messages are sent out serially on P62 at 2 Mbaud in 8-N-1 format.
- DEBUG messages always start with "CogN ", where N is the cog number, followed by two spaces, and they always end with CR+LF (new line).
- Up to 255 DEBUG statements can exist within your application, since the BRK instruction is used to interrupt and select the particular DEBUG statement definition.
- You can define several symbols to modify debugger behavior: DEBUG_COG, DEBUG_DELAY, DEBUG_PIN, DEBUG_TIMESTAMP, etc. See table.
- Each time a debug-enabled cog is started, a debug message is output to indicate the cog number, code address (PTRB), parameter (PTRA), and 'load' or 'jump' mode.
- For Spin2, DEBUG statements can output expression and variable values, hub byte/word/long arrays, and register arrays.
- For PASM, DEBUG statements can output register values/arrays, hub byte/word/long arrays, and constants. PASM syntax is used: implied register or #immediate.
- DEBUG output data can be displayed in decimal, hex, or binary, signed or unsigned, and sized to byte, word, long, or auto. Hub character strings are also supported.

- DEBUG output commands show both the source and value: "DEBUG(UHEX(x))" might output "x = \$123".
- DEBUG commands which output data can have multiple sets of parameters, separated by commas: SDEC(x,y,z) and LSTR(ptr1,size1,ptr2,size2)
- Commas are automatically output between data: "DEBUG(UHEX_BYTE(d,e,f), SDEC(g))" might output "d = \$45, e = \$67, f = \$89, g = -1_024".
- All DEBUG output commands have alternate versions, ending in "_" which output only the value: DEBUG(UHEX_BYTE_(d,e,f)) might output "\$45, \$67, \$89".
- DEBUG statements can contain comma-separated strings and characters, aside from commands: DEBUG("We got here! Oh, Nooooo...", 13, 13)
- DEBUG statements may contain IF() and IFNOT() commands to gate further output within the statement. An initial IF/IFNOT will gate the entire message.
- DEBUG statements may contain a final DLY(milliseconds) command to slow down a cog's messaging, since messages may stream at the rate of ~10,000 per second.
- DEBUG serial output can be redirected to a different pin, at a different baud rate, for displaying/logging elsewhere.
- LOCK[15] is allocated by the debugger and used among all cogs during their debug interrupts to time-share the DEBUG serial-transmit pin.
- Command-line supports DEBUG-only mode: PNut -debug {CommPort if not 1} {BaudRate if not 2_000_000}

DEBUG Statement (v=100, BYTE[a]=1,2,3,4,5)	DEBUG Message Output	Note
DEBUG("`LOGIC MyDisplay SAMPLES ", SDEC_(v))	Cog0 `LOGIC MyDisplay SAMPLES 100	Regular DEBUG syntax can drive DEBUG displays, but it's not optimal.
DEBUG(`LOGIC MyDisplay SAMPLES 100)	`LOGIC MyDisplay SAMPLES 100	DEBUG-display syntax is simpler and 'CogN' is omitted in the output.
DEBUG(`LOGIC MyDisplay SAMPLES `(v))	`LOGIC MyDisplay SAMPLES 100	Decimal numbers are output using `(value) notation. Short for SDEC_.
DEBUG(`LOGIC MyDisplay SAMPLES `\${v}))	`LOGIC MyDisplay SAMPLES \$64	Hex numbers are output using `\${value) notation. Short for UHEX_.
DEBUG(`LOGIC MyDisplay SAMPLES `%(v))	`LOGIC MyDisplay SAMPLES %1100100	Binary numbers are output using `%(value) notation. Short for UBIN_.
DEBUG(`LOGIC MyDisplay TITLE `#(v))	`LOGIC MyDisplay TITLE 'd'	Characters are output using `#(value) notation.
DEBUG(`MyDisplay `UDEEC_BYTE_ARRAY_(@a,5))	`MyDisplay 1, 2, 3, 4, 5	Regular DEBUG commands can follow the backtick, as well.

13.2) Simple DEBUG example in Spin2

```
CON _clkfreq = 10_000_000    'set 10 MHz clock (assumes 20 MHz crystal)

PUB go() | i

  REPEAT i FROM 0 TO 9      'count from 0 to 9

    DEBUG(UDEC(i))          'debug, output i
```

When run with Ctrl-F10, the Debug window opens and this is what appears:

```
Cog0 INIT $0000_0000 $0000_0000 load
Cog0 INIT $0000_0D6C $0000_10BC jump
Cog0 i = 0
Cog0 i = 1
Cog0 i = 2
Cog0 i = 3
Cog0 i = 4
Cog0 i = 5
Cog0 i = 6
Cog0 i = 7
Cog0 i = 8
Cog0 i = 9
```

In the first line of the report, you see Cog0 loading the Spin2 set-up code from \$00000. In the second line, the Spin2 interpreter is launched from \$00D58 with its stack space starting at \$0101C. After that, the Spin2 program is running and you see 'i' iterating from 0 to 9.

If you change the "9" to "99" in the REPEAT, data will scroll too fast to read, but by adding a DLY command at the end of the DEBUG statement, you can slow down the output:

```
debug(udec(i), dly(250))    'debug, output i with a 250ms delay after each report
```

Let's say you want to limit the messages being output, so that only odd values of 'i' are shown. You could use an IF at the start of your DEBUG statement to check the least-significant bit of 'i'. When the IF is false, no message will be output, causing only the odd values of i to be shown:

```
debug(if(i & 1), udec(i), dly(250))    'debug, output only odd i values with a 250ms delay after each report
```

13.3) Simple DEBUG example in PASM

```

CON _clkfreq = 10_000_000    'set 10 MHz clock (assumes 20 MHz crystal)

DAT    ORG

        MOV    i,#9          'set i to 9

loop    DEBUG (UHEX_LONG(i)) 'debug, output i in hex

        DJNF   i,#loop       'decrement i and loop if not -1

        JMP    #$            'don't go wandering off, stay here

i       RES    1              'reserve one register as 'i'

```

When run with Ctrl-F10, the Debug window opens and this is what appears:

```

Cog0 INIT $0000_0000 $0000_0000 load
Cog0 i = $0000_0009
Cog0 i = $0000_0008
Cog0 i = $0000_0007
Cog0 i = $0000_0006
Cog0 i = $0000_0005
Cog0 i = $0000_0004
Cog0 i = $0000_0003

```

```
Cog0 i = $0000_0002
```

```
Cog0 i = $0000_0001
```

```
Cog0 i = $0000_0000
```

In the first line of the report, you see Cog0 loading our PASM program from \$00000. After that, the program runs and you see 'i' iterating from 9 down to 0.

If you change the "9" to "99" in the MOV instruction and you'd like to slow things down, add a DLY command to the DEBUG statement and be sure to express the milliseconds as #250, since a plain 250 would be understood as register 250:

```
debug (uhex_long(i), dly(#250)) 'debug, output i in hex and delay for 250ms after each  
report
```


There are two steps to using graphical DEBUG displays. First, they must be instantiated and, second, they must be fed:

To Use a Display:	1st	2nd	3rd	4th	Note
First, instantiate it.	`	display_type	unknown_symbol	keyword(s), number(s), string(s)	Unknown_symbol becomes instance_name.
Then, feed it.	`	instance_name(s)	keyword(s), number(s), string(s)		Multiple displays can be fed the same data.

To bring this all together, let's show a sawtooth wave on a SCOPE display:

```
CON _clkfreq = 10_000_000
```

```
PUB go() | i
```

```
  debug(`SCOPE MyScope SIZE  
254 84 SAMPLES 128)
```

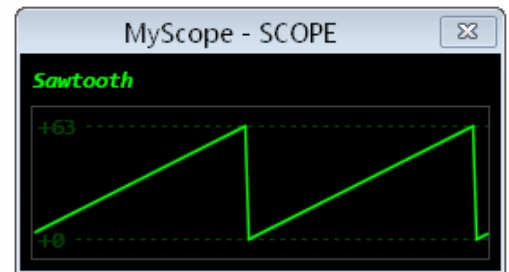
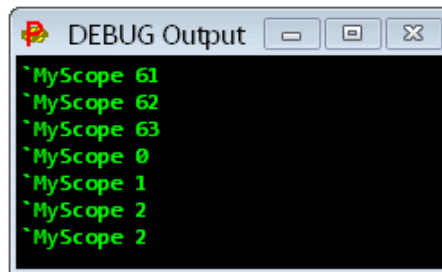
```
  debug(`MyScope 'Sawtooth' 0  
63 64 10 %1111)
```

```
  repeat
```

```
    debug(`MyScope `(i & 63))
```

```
    i++
```

```
  waitms(50)
```



Step 1 Instantiate Graphic Display

```
debug(`SCOPE Scope_Name SIZE 254 84 SAMPLES 128) 'note the backtick "`
```

```
debug(`Scope_Name 'Sawtooth' 0 63 64 10 %1111) 'note strings have single quote 'Sawtooth'
```

Step 2 Feed Data to Window (Display)

```
Debug(`Scope_Name ( i & 63))
```

```
Debug(`Scope_Name
```

In the example above, a SCOPE is instantiated called MyScope that is 254 x 84 pixels and shows 128 samples. A width of 254 was chosen since samples are numbered 0..127 and I wanted them to be spaced at a constant two-pixel pitch ($127 * 2 = 254$). A height of 84 was chosen so that there would be 10 pixels above and below the waveform, which will have a height of 64 pixels.

A channel called "Sawtooth" is defined which, for the purpose of display, has a bottom value of 0 and a top value of 63, is 64 pixels tall within that range, and is elevated 10 pixels off the bottom of the scope window. The %1111 enables top and bottom legend values and top and bottom lines. Within the REPEAT block, the SCOPE is fed a repeating pattern of 0..63 which forms the sawtooth wave. The SCOPE updates its display each time it receives a value. If there were eight channels defined, instead of just one, it would update the display on every eighth value received, drawing all eight channels.

13.4) Commands for use in DEBUG statements

Conditionals	Details
IF(condition)	If condition $\neq 0$ then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IF will gate ALL output for the statement, including the "CogN " + CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important.
IFNOT(condition)	If condition = 0 then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IFNOT will gate ALL output for the statement, including the "CogN " + CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important.

String Output *	Details	Output
ZSTR(hub_pointer)	Output zero-terminated string at hub_pointer	"Hello!"
LSTR(hub_pointer,size)	Output 'size' characters of string at hub_pointer	"Goodbye."

Decimal Output, unsigned *	Details	Min Output	Max Output
UDEC(value)	Output unsigned decimal value	0	4_294_967_295
UDEC_BYTE(value)	Output byte-size unsigned decimal value	0	255
UDEC_WORD(value)	Output word-size unsigned decimal value	0	65_535
UDEC_LONG(value)	Output long-size unsigned decimal value	0	4_294_967_295
UDEC_REG_ARRAY(reg_pointer, size)	Output register array as unsigned	0	4_294_967_295

	decimal values		
UDEC_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as unsigned decimal values	0	255
UDEC_WORD_ARRAY(hub_pointer,size)	Output hub word array as unsigned decimal values	0	65_535
UDEC_LONG_ARRAY(hub_pointer,size)	Output hub long array as unsigned decimal values	0	4_294_967_295
Decimal Output, signed *	Details	Min Output	Max Output
SDEC(value)	Output signed decimal value	- 2_147_483_648	2_147_483_647
SDEC_BYTE(value)	Output byte-size signed decimal value	-128	127
SDEC_WORD(value)	Output word-size signed decimal value	-32_768	32_767
SDEC_LONG(value)	Output long-size signed decimal value	- 2_147_483_648	2_147_483_647
SDEC_REG_ARRAY(reg_pointer,size)	Output register array as signed decimal values	- 2_147_483_648	2_147_483_647
SDEC_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as signed	-128	127

	decimal values		
SDEC_WORD_ARRAY(hub_pointer,size)	Output hub word array as signed decimal values	-32_768	32_767
SDEC_LONG_ARRAY(hub_pointer,size)	Output hub long array as signed decimal values	-2_147_483_648	2_147_483_647
Hexadecimal Output, unsigned *	Details	Min Output	Max Output
UHEX(value)	Output auto-size unsigned hex value	\$0	\$FFFF_FFFF
UHEX_BYTE(value)	Output byte-size unsigned hex value	\$00	\$FF
UHEX_WORD(value)	Output word-size unsigned hex value	\$0000	\$FFFF
UHEX_LONG(value)	Output long-size unsigned hex value	\$0000_0000	\$FFFF_FFFF
UHEX_REG_ARRAY(reg_pointer,size)	Output register array as unsigned hex values	\$0000_0000	\$FFFF_FFFF
UHEX_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as unsigned hex values	\$00	\$FF
UHEX_WORD_ARRAY(hub_pointer,size)	Output hub word array as unsigned hex values	\$0000	\$FFFF

UHEX_LONG_ARRAY(hub_pointer,size)	Output hub long array as unsigned hex values	\$0000_0000	\$FFFF_FFFF
Hexadecimal Output, signed *	Details	Min Output	Max Output
SHEX(value)	Output auto-size signed hex value	-\$8000_0000	\$7FFF_FFFF
SHEX_BYTE(value)	Output byte-size signed hex value	-\$80	\$7F
SHEX_WORD(value)	Output word-size signed hex value	-\$8000	\$7FFF
SHEX_LONG(value)	Output long-size signed hex value	-\$8000_0000	\$7FFF_FFFF
SHEX_REG_ARRAY(reg_pointer,size)	Output register array as signed hex values	-\$8000_0000	\$7FFF_FFFF
SHEX_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as signed hex values	-\$80	\$7F
SHEX_WORD_ARRAY(hub_pointer,size)	Output hub word array as signed hex values	-\$8000	\$7FFF
SHEX_LONG_ARRAY(hub_pointer,size)	Output hub long array as signed hex values	-\$8000_0000	\$7FFF_FFFF
Binary Output, unsigned *	Details	Min Output	Max Output
UBIN(value)	Output auto-size unsigned binary value	%0	%11111111_11111111_11111111_11111111

UBIN_BYTE(value)	Output byte-size unsigned binary value	%00000000	%11111111
UBIN_WORD(value)	Output word-size unsigned binary value	%00000000_00000000	%11111111_11111111
UBIN_LONG(value)	Output long-size unsigned binary value	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_REG_ARRAY(reg_pointer, size)	Output register array as unsigned binary values	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_BYTE_ARRAY(hub_pointer, size)	Output hub byte array as unsigned binary values	%00000000	%11111111
UBIN_WORD_ARRAY(hub_pointer, size)	Output hub word array as unsigned binary values	%00000000_00000000	%11111111_11111111
UBIN_LONG_ARRAY(hub_pointer, size)	Output hub long array as unsigned binary values	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
Binary Output, signed *	Details	Min Output	Max Output
SBIN(value)	Output auto-size signed binary value	- %10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_BYTE(value)	Output byte-size signed binary value	-%10000000	%01111111
SBIN_WORD(value)	Output word-size signed binary value	- %10000000_00000000	%01111111_11111111

SBIN_LONG(value)	Output long-size signed binary value	- %10000000_00 000000_000000 00_00000000	%01111111_11111111_1 1111111_11111111
SBIN_REG_ARRAY(reg_pointer, size)	Output register array as signed binary values	- %10000000_00 000000_000000 00_00000000	%01111111_11111111_1 1111111_11111111
SBIN_BYTE_ARRAY(hub_pointer, size)	Output hub byte array as signed binary values	-%10000000	%01111111
SBIN_WORD_ARRAY(hub_pointer, size)	Output hub word array as signed binary values	- %10000000_00 000000	%01111111_11111111
SBIN_LONG_ARRAY(hub_pointer, size)	Output hub long array as signed binary values	- %10000000_00 000000_000000 00_00000000	%01111111_11111111_1 1111111_11111111

Delay to Pace Messages	Details
DLY(milliseconds)	Delay for some milliseconds to slow down continuous message outputs for this cog. DLY is only allowed as the last command in a DEBUG statement, since it releases LOCK[15] before the delay, permitting other cogs to capture LOCK[15] so that they may take control of the DEBUG serial-transmit pin and output their own DEBUG messages.

* These commands accept multiple parameters, or multiple sets of parameters. Alternate commands with the same names, but ending in "_", are also available for value-only output (i.e. ZSTR_, LSTR_, UDEC_).

13.5) Symbols you can define to modify DEBUG behavior

CON Symbol	Default	Purpose
DEBUG_COGS	%11111111	Selects which cogs have debug interrupts enabled. Bits 7..0 enable debugging interrupts in cogs 7..0.
DEBUG_DELAY	0	Sets a delay in milliseconds before your application runs and DEBUG messages start appearing.
DEBUG_PIN	62	Sets the DEBUG serial output pin. For DEBUG windows to open, DEBUG_PIN must be 62.
DEBUG_BAUD	2_000_000	Sets the DEBUG baud rate.
DEBUG_TIMESTAMP	undefined	By declaring this symbol, each DEBUG message will be time-stamped with the 64-bit CT value.
DEBUG_LOG_SIZE	0	Sets the maximum size of the 'DEBUG.log' file which will collect DEBUG messages. A value of 0 will inhibit log file generation.
DEBUG_LEFT	(dynamic)	Sets the left screen coordinate where the DEBUG message window will appear.
DEBUG_TOP	(dynamic)	Sets the top screen coordinate where the DEBUG message window will appear.
DEBUG_WIDTH	(dynamic)	Sets the width of the DEBUG message window.
DEBUG_HEIGHT	(dynamic)	Sets the height of the DEBUG message window.
DEBUG_DISPLAY_LEFT	0	Sets the overall left screen offset where any DEBUG displays will appear (adds to 'POS' x coordinate in each DEBUG display).
DEBUG_DISPLAY_TOP	0	Sets the overall top screen offset where any DEBUG displays will appear (adds to 'POS' y coordinate in each DEBUG display).
DEBUG_WINDOWS_OFF	0	Disables any DEBUG windows from opening after downloading, if set to a non-zero value.

13.6) Packed-Data Modes

Packed-data modes are used to efficiently convey sub-byte data types, by having the host side unpack them from bytes, words, or longs it receives. As well, bytes can be sent within words and longs, and words can be sent within longs for some efficiency improvement.

To establish packed-data operation, you must specify one of the modes listed below, followed by optional 'ALT' and 'SIGNED' keywords:

```
packed_mode {ALT} {SIGNED}
```

The **ALT** keyword will cause bits, double-bits, or nibbles, within each byte sent, to be reordered on the host side, within each byte. This simplifies cases where the raw data you are sending has its bitfields out-of-order with respect to the DEBUG display you are using. This is most-likely to be needed for bitmap data that was composed in standard formats.

The **SIGNED** keyword will cause all unpacked data values to be sign-extended on the host side.

Packed-Data Modes	Descriptions	Final Values	Final Values if SIGNED
LONGS_1BIT	Each value received is translated into 32 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
LONGS_2BIT	Each value received is translated into 16 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
LONGS_4BIT	Each value received is translated into 8 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7
LONGS_8BIT	Each value received is translated into 4 separate 8-bit values, starting from the LSBs of the received value.	0..255	-128..127
LONGS_16BIT	Each value received is translated into 2 separate 16-bit values, starting from the LSBs of the received value.	0..65,535	-32,768..32,767

WORDS_1BIT	Each value received is translated into 16 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
WORDS_2BIT	Each value received is translated into 8 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
WORDS_4BIT	Each value received is translated into 4 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7
WORDS_8BIT	Each value received is translated into 2 separate 8-bit values, starting from the LSBs of the received value.	0..255	-128..127
BYTES_1BIT	Each value received is translated into 8 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
BYTES_2BIT	Each value received is translated into 4 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
BYTES_4BIT	Each value received is translated into 2 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7

13.7) Graphical DEBUG Displays

DEBUG messages can invoke special graphical DEBUG displays which are built into the tool. These graphical displays each take the form of a unique window. Once instantiated, displays can be continuously fed data to generate animated visualizations. These displays are very handy for development and debugging, as various data types can be viewed in their native contexts. Up to 32 graphical displays can be running simultaneously.

Currently, the following graphical DEBUG displays are implemented, but more will be added in the future:

Display Types	Descriptions
LOGIC	Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern
SCOPE	Oscilloscope with 1..8 channels, can trigger on level with hysteresis
SCOPE_XY	XY oscilloscope with 1..8 channels, persistence of 0..512 samples, polar mode, log scale mode
FFT	Fast Fourier Transform with 1..8 channels, 4..2048 points, windowed results, log scale mode
SPECTRO	Spectrograph with 4..2048-point FFT, windowed results, phase-coloring, and log scale mode
PLOT	General-purpose plotter with cartesian and polar modes
TERM	Text terminal with up to 300 x 200 characters, 6..200 point font size, 4 simultaneous color schemes
BITMAP	Bitmap, 1..2048 x 1..2048 pixels, 1/2/4/8/16/32-bit pixels with 19 color systems, 15 direction/autoscroll modes, independent X and Y pixel size of 1..256
MIDI	Piano keyboard with 1..128 keys, velocity depiction, variable screen scale

When a DEBUG message contains a backtick (`) character (ASCII \$60), a string, containing everything from the backtick to the end of the message, is sent to the graphical DEBUG display parser. The parser looks for several different element types, treating any commas as whitespace:

Element Type	Example	Description
display_type	LOGIC, SCOPE, PLOT, BITMAP	This is the formal name of the graphical DEBUG display type you wish to instantiate.
unknown_symbol	MyLogicDisplay	Each graphical DEBUG display Instance must be given a unique symbolic name.
instance_name	MyLogicDisplay	Once instantiated, a graphical DEBUG display instance is referenced by its symbolic name.
keyword	TITLE, POS, SIZE, SAMPLES	Keywords are used to configure displays. They might be followed by numbers, strings, and other keywords.
number	1024, \$FF, % 1010	Numbers can be expressed in decimal, hex (\$), and binary (%).
string	'Here is a string'	Strings are expressed within single-quotes.

Before getting into how all this fits together, we need to go over some special DEBUG-display syntax that can be used for displays. This syntax is invoked when the first character in the DEBUG statement is the backtick. This causes everything in the DEBUG statement to be viewed as a string, except when subsequent backticks act as 'escape' characters to allow normal or shorthand DEBUG commands.

DEBUG Statement (v=100, BYTE[a]=1,2,3,4,5)	DEBUG Message Output	Note
DEBUG("`LOGIC MyDisplay SAMPLES ", SDEC_(v))	Cog0 `LOGIC MyDisplay SAMPLES 100	Regular DEBUG syntax can drive DEBUG displays, but it's not optimal.
DEBUG(`LOGIC MyDisplay SAMPLES 100)	`LOGIC MyDisplay SAMPLES 100	DEBUG-display syntax is simpler and 'CogN' is omitted in the output.
DEBUG(`LOGIC MyDisplay SAMPLES `(v))	`LOGIC MyDisplay SAMPLES 100	Decimal numbers are output using `(value) notation. Short for SDEC_.
DEBUG(`LOGIC MyDisplay SAMPLES `\${v}))	`LOGIC MyDisplay SAMPLES \$64	Hex numbers are output using `\${value) notation. Short for UHEX_.
DEBUG(`LOGIC MyDisplay SAMPLES `%(v))	`LOGIC MyDisplay SAMPLES %1100100	Binary numbers are output using `%(value) notation. Short for UBIN_.
DEBUG(`LOGIC MyDisplay TITLE `#(v))	`LOGIC MyDisplay TITLE 'd'	Characters are output using #(value) notation.
DEBUG(`MyDisplay `UDECODE_BYTE_ARRAY_(@a,5))	`MyDisplay 1, 2, 3, 4, 5	Regular DEBUG commands can follow the backtick, as well.

There are two steps to using graphical DEBUG displays. First, they must be instantiated and, second, they must be fed:

To Use a Display:	1st	2nd	3rd	4th	Note
First, instantiate it.	`	display_type	unknown_symbol	keyword(s), number(s), string(s)	Unknown_symbol becomes instance_name.
Then, feed it.	`	instance_name(s)	keyword(s), number(s), string(s)		Multiple displays can be fed the same data.

Note: The backtick is critical and the variables must have single quotes

`debug(`BobTerm 0 ``udec(x)' 9 ``udec(y)')` 'gives decimal value

`debug(`BobTerm 0 ``uhex(x)' 9 ``uhex(y)')` 'gives hex value

`debug(`BobTerm 0 ``ubin(x)' 9 ``ubin(y)')` 'gives binary value

`debug(`BobTerm 0 'Var x = `(x)' 9 'Var y = `(y)')` 'gives decimal value

`debug(`BobTerm 0 ``$(x)' 9 ``$(y)')` 'gives hex value

`debug(`BobTerm 0 ``%(x)' 9 ``%(y)')` 'gives binary value

```
CON _clkfreq = 10_000_000
```

```
PUB go() | i
```

```
  debug(`SCOPE MyScope SIZE 254 84 SAMPLES 128)
```

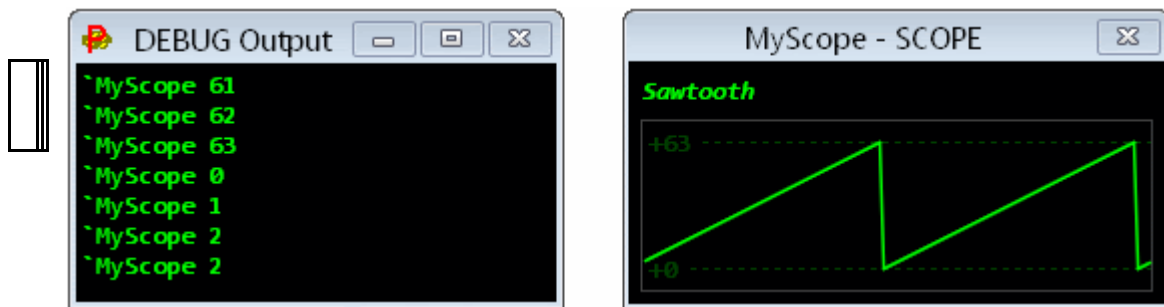
```
  debug(`MyScope 'Sawtooth' 0 63 64 10 %1111)
```

```
  repeat
```

```
    debug(`MyScope `(i & 63))
```

```
    i++
```

```
  waitms(50)
```



In the example above, a SCOPE is instantiated called MyScope that is 254 x 84 pixels and shows 128 samples. A width of 254 was chosen since samples are numbered 0..127 and I wanted them to be spaced at a constant two-pixel pitch ($127 * 2 = 254$). A height of 84 was chosen so that there would be 10 pixels above and below the waveform, which will have a height of 64 pixels. A channel called "Sawtooth" is defined which, for the purpose of display, has a bottom value of 0 and a top value of 63, is 64 pixels tall within that range, and is elevated 10 pixels off the bottom of the scope window. The %1111 enables top and bottom legend values and top and bottom lines. Within the REPEAT block, the SCOPE is fed a repeating pattern of 0..63 which forms the sawtooth wave. The SCOPE updates its display each time it receives a value. If there were eight channels defined, instead of just one, it would update the display on every eighth value received, drawing all eight channels.

13.8) Logic Analyzer Display

Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern

```
CON _clkfreq = 10_000_000

PUB go() | i

    debug(`LOGIC MyLogic SAMPLES
32 'Low' 3 'Mid' 2 'High')

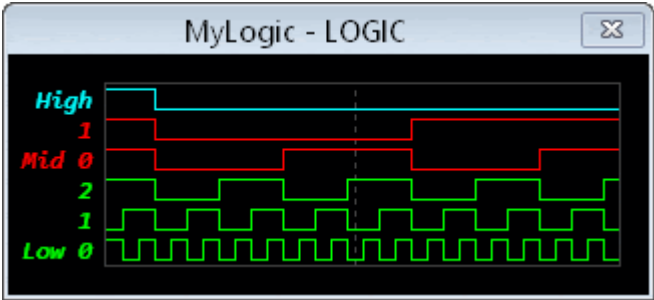
    debug(`MyLogic TRIGGER $07 $04
HOLDOFF 2)

    repeat

        debug(`MyLogic `(i & 63))

        i++

        waitms(25)
```



LOGIC Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SAMPLES 4_to_2048	Set the number of samples to track and display.	32
SPACING 2_to_32	Set the sample spacing. The width of the display will be SAMPLES * SPACING.	8
RATE 1_to_2048	Set the number of samples (or triggers, if enabled) before each display update.	1

LINESIZE 1_to_7	Set the line size.	1
TEXTSIZE 6_to_200	Set the legend text size. Height of text determines height of logic levels.	editor text size
COLOR back_color {grid_color}	Set the background and grid colors *.	BLACK, GREY 4
'name' {1_to_32 {color}}	Set the first/next channel or group name, optional bit count, optional color *.	1, default color
packed_data_mode	Enable packed-data mode. See description at end of this section.	<none>
LOGIC Feeding	Description	Default
TRIGGER mask match sample_offset	Trigger on (data & mask) = match. If mask = 0, trigger is disabled.	0, 1, SAMPLES / 2
HOLDOFF 2_to_2048	Set the minimum number of samples required from trigger to trigger.	SAMPLES
data	Numerical data is applied LSB-first to the channels.	
CLEAR	Clear the sample buffer and display, wait for new data.	
SAVE {WINDOW} 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

The LOGIC display can be used to display data that was captured at high speed. In the example below, the P2 is generating 8-N-1 serial at 333 Mbaud using a smart pin. This bit stream can be captured by the streamer. On every clock, the streamer will record the smart pin's IN signal and its output state, as read from an adjacent pin. Every time it gets four two-bit sample sets, it does an RFBYTE to save them to hub RAM, forming contiguous bytes, words, and longs. By invoking the LONGS_2BIT packed-data mode, we can have the LOGIC display unpack the two-bit sample sets from longs, yielding 16 sets per long.

13.9) Scope Display

SCOPE Display Oscilloscope with 1..8 channels, can trigger on level with hysteresis

```
CON _clkfreq = 100_000_000

PUB go() | a, af, b, bf

    debug(`SCOPE MyScope)

    debug(`MyScope 'FreqA' -1000 1000 100 136 15
MAGENTA)

    debug(`MyScope 'FreqB' -1000 1000 100 20 15
ORANGE)

    debug(`MyScope TRIGGER 0 HOLDOFF 2)

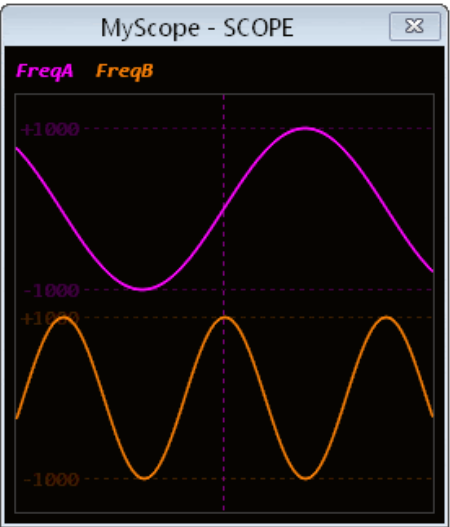
repeat

    a := qsin(1000, af++, 200)

    b := qsin(1000, bf++, 99)

    debug(`MyScope `(a,b))

    waitus(200)
```



SCOPE Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SIZE width height	Set the display size (32..2048 x 32..2048)	255, 256
SAMPLES 16_to_2048	Set the number of samples to track and display.	256

RATE 1_to_2048	Set the number of samples (or triggers, if enabled) before each display update.	1
DOTSIZE 0_to_32	Set the dot size in pixels for showing exact sample points.	0
LINESIZE 0_to_32	Set the line size in half-pixels for connecting sample points.	3
TEXTSIZE 6_to_200	Set the legend text size.	editor text size
COLOR back_color {grid_color}	Set the background and grid colors *.	BLACK, GREY 4
packed_data_mode	Enable packed-data mode. See description at end of this section.	<none>

SCOPE Feeding	Description	Default
'name' { min { max { y_size { y_base { legend { color } } } } } }	Set first/next channel name, min value, max value, y size, y base, legend, and color *. Legend is %abcd, where %a to %d enable max legend, min legend, max line, min line.	full, no legend, default color
TRIGGER channel { arm_level { trigger_level { offset } } }	Set the trigger channel, arm level, trigger level, and right offset. If channel=-1, disabled.	-1, -1, 0, width / 2
HOLDOFF 2_to_2048	Set the minimum number of samples required from trigger to trigger.	SAMPLES
data	Numerical data is applied to the channels in ascending order.	
CLEAR	Clear the sample buffer and display, wait for new data.	
SAVE { WINDOW } 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

13.10) Scope_XY Display

SCOPE_XY Display XY oscilloscope with 1..8 channels, persistence of 1..512 samples, polar mode, log scale mode

```

CON _clkfreq = 100_000_000

PUB go() | i

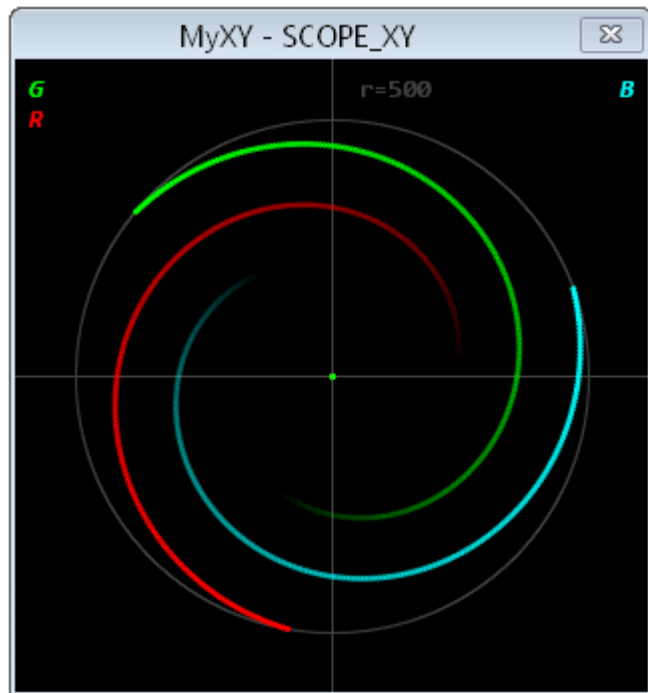
    debug(`SCOPE_XY MyXY RANGE
500 POLAR 360 'G' 'R' 'B')

    repeat

        repeat i from 0 to 500

            debug(`MyXY `(i, i, i, i+120, i,
i+240))

            waitms(5)
  
```



SCOPE_XY Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SIZE radius	Set the display radius in pixels.	128
RANGE 1_to_7FFFFFFF	Set the unit circle radius for incoming data	\$7FFFFFFF
SAMPLES 0_to_512	Set the number of samples to track and display with persistence. Use 0 for infinite persistence.	256
RATE 1_to_512	Set the number of samples before each display update.	1

DOTSIZE 2_to_20	Set the dot size in half-pixels for showing sample points.	6
TEXTSIZE 6_to_200	Set the legend text size.	editor text size
COLOR back_color {grid_color}	Set the background and grid colors *.	BLACK, GREY 4
POLAR {twopi {offset}}	Set polar mode, twopi value, and offset. For a twopi value of \$100000000 or -\$100000000, use 0 or -1.	\$100000000, 0
LOGSCALE	Set log-scale mode to magnify points within the unit circle.	<off>
'name' {color}	Set the first/next channel name and optionally assign it a color *.	default color
packed_data_mode	Enable packed-data mode. See description at end of this section.	<none>
SCOPE_XY Feeding	Description	Default
x y	X-Y data pairs are applied to the channels in ascending order. In polar mode, x=length and y=angle.	
CLEAR	Clear the sample buffer and display, wait for new data.	
SAVE {WINDOW} 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

13.11) TERM Display

Terminal for displaying text

```
CON _clkfreq = 10_000_000
```

```
PUB go() | i
```

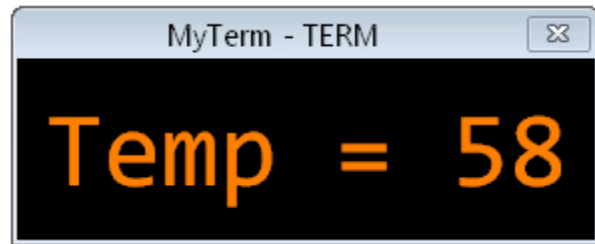
```
    debug(`TERM MyTerm SIZE 9 1  
    TEXTSIZE 40)
```

```
    repeat
```

```
        repeat i from 50 to 60
```

```
            debug(`MyTerm 1 'Temp = `(i)')
```

```
        waitms(500)
```



TERM Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SIZE columns rows	Set the number of terminal columns (1..256) and terminal rows (1..256).	40, 20
TEXTSIZE size	Set the terminal text size (6..200).	editor text size
COLOR text_color back_color ...	Set text-color and background-color combos #0..#3. *	default colors
BACKCOLOR color	Set the display background color. *	BLACK
UPDATE	Set UPDATE mode. The display will only be updated when fed an 'UPDATE' command.	automatic update
TERM Feeding	Description	Default

character	0 = Clear terminal display and home cursor. 1 = Home cursor. 2 = Set column to next character value. 3 = Set row to next character value. 4 = Select color combo #0. 5 = Select color combo #1. 6 = Select color combo #2. 7 = Select color combo #3. 8 = Backspace. 9 = Tab to next 8th column. 13+10 or 13 or 10 = New line. 32..255 = Printable character.	
'string'	Print string.	
CLEAR	Clear the display to the background color.	
UPDATE	Update the window with the current text screen. Used in UPDATE mode.	
SAVE {WINDOW} 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is a modal value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

13.1_Example_WRD_DEBUG_zstr_lstr_udec_ubin_uhex

```

{{13.1_Example_WRD_DEBUG_zstr_lstr_udec_ubin_uhex}}
{{
ZSTR(hub_pointer)    Output zero-terminated string at hub_pointer    "Hello Propeller"
LSTR(hub_pointer,size) Output 'size' characters of string at hub_pointer    "Hello"
UDEC(value)          Output unsigned decimal value                    0 .. 4_294_967_295
UHEX(value)          Output auto-size unsigned hex value              $0 .. $FFFF_FFFF
UBIN(value)          Output auto-size unsigned binary value
%0..%11111111_11111111_11111111_11111111
}}
Con
_clkfreq = 200_000_000
Var
long varMsg
long varDec
long varBin
long varHex
Pub main() |x,y
  varMsg := string("Hello Parallax") 'returns address of string
  varDec := 5432
  varBin := %1111_1111
  varHex := $F123

  debug("Output a Message Header")
  debug(zstr(@MyString)) 'output zero terminated string
  debug(lstr(@MyString,5)) 'output the 5 characters at adress @MyString
  debug(zstr(varMsg))
  debug(udec(varDec))
  debug(ubin(varBin))
  debug(uhex(varHex))
DAT
MyString Byte "Hello Propeller",0

```

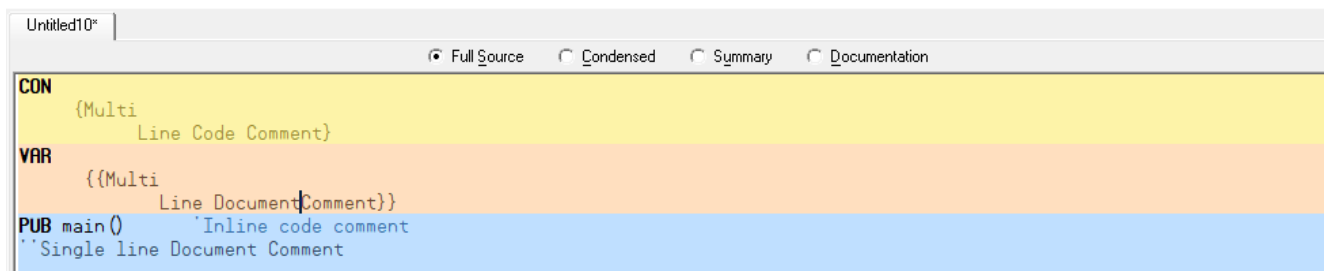
14.0) Program Structure

14.0.1) Propeller Tool (IDE)

There are several programming tools other than the “Propeller Tool” but this document will only use the “Propeller Tool” with Parallax SPIN and PASM (Propeller Assembly Machine Language).

The “Propeller Tool” is Parallax IDE (integrated development environment) allows the programs to have code and document comments to organize the documentation:

‘ single line code comment (apostrophe)
 ” single line document comment (two apostrophe not quotation)
 {...} Multi Line Code Comment
 {{...}} Multi Line document comment



Full Source	all code and comments
Condensed	code and Inline comments
Summary	Constants, Variables, Methods (no comments)
Documentation	Memory usage comments and Methods

The “Propeller Tool” Compiles the user program from “Top Object” selected. Objects are compiled from “Library” file folder and the current directory the “Top Object” is stored in (current working directory)

The default template file under “EDIT→PREFERENCES” can be used to customize User definitions for “New Projects”.

Edit → Find/Replace useful for editing files case is not used by the compiler this has advantages and disadvantages.

14.0.1.1) Spin Propeller Tool (IDE) Compiler Directives

CON VAR PUB PRI DAT OBJ

14.0.2) PASM Propeller Tool (IDE) Compiler Directives

ORG Adjust Compile-time cog Address Pointer

FIT validate that instruction/data fit in cog (511 registers)

RES reserve next long(s) for symbol

\$ current address here JMP #here = <Symbol> JMP #

\$+/- value offset to current address JMP #-\$4 or JMP #+\$4

14.0.2) P2 Memory Organization

Cogs use 20-bit addresses for program counters (PC). This affords an execution space of up to 1MB. Depending on the value 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	Memory Width	PC Increment
\$00000..\$001FF	cog register RAM	32 bits	1
\$00200..\$003FF	cog lookup RAM	32 bits	1
\$00400..\$FFFFFF	hub RAM	8 bits	4

REGISTER EXECUTION

When the PC is in the range of \$00000 and \$001FF, the cog is fetching instructions from cog register RAM. This is commonly referred to as "cog execution mode." There is no special consideration when taking branches to a cog register address.

LOOKUP EXECUTION

When the PC is in the range of \$00200 and \$003FF, the cog is fetching instructions from cog lookup RAM. This is commonly referred to as "lut execution mode." There is no special consideration when taking branches to a cog lookup address,

HUB EXECUTION

When the PC is in the range of \$00400 and \$FFFFFF, the cog is fetching instructions from hub RAM. This is commonly referred to as "hub execution mode." When executing from hub RAM, the cog employs the FIFO hardware to spool up instructions so that a stream of instructions will be available for continuous execution. Branching to a hub address takes a minimum of 13 clock cycles. If the instruction is not aligned to a slice, one additional clock cycle is required.

While in hub execution mode, the FIFO cannot be used for anything else. So, during hub execution these instructions must be avoided:

RDFAST / WRFAST / FBLOCK

RFBYTE / RWORD / RFLONG / RFVAR / RFVARS

WFBYTE / WWORD / WFLONG

XINIT / XZERO / XCONT - when the streamer mode engages the FIFO

It is not possible to execute code from hub addresses \$00000 through \$003FF, as the cog will instead read instructions from the cog register or lookup RAM as indicated above.

14.0.3) COG RAM

Each cog has a primary 512 x 32-bit dual-port RAM, which can be used in multiple ways:

- Direct/Register access
- As a source of program instructions

GENERAL PURPOSE REGISTERS

RAM registers \$000 through \$1EF are general-purpose registers for code and data usage.

DUAL-PURPOSE REGISTERS

RAM registers \$1F0 through \$1F7 may either be used as general-purpose registers, or may be used as special-purpose registers if their associated functions are enabled.

\$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

SPECIAL-PURPOSE REGISTERS

Each cog contains 8 special-purpose registers that are mapped into the RAM register address space from \$1F8 to \$1FF. In general, when specifying an address between \$1F8 and \$1FF, the instruction is accessing a special-purpose register, not just the underlying RAM.

\$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

LOOKUP RAM

Each cog has a secondary 512 x 32-bit dual-port RAM, which can be used in multiple ways:

- Load/Store access
- As a source or destination for the streamer hardware
- As a lookup table for bytecode execution
- As a data source for smart pins
- As a "RAM sharing" mechanism between paired cogs
- As a source of program instructions (see [COGS > INSTRUCTION MODES > LOOKUP EXECUTION](#))

NOTE: The term "lookup" (and "lut", which is short for "look-up table") is due to historical usage in the original Propeller microcontroller. This RAM can still be used in a "lookup" context, but can also be used for many other purposes, as indicated above.

PASM Communication Registers

Each of these cog registers can be referenced by name PR0-PR7

PR0 \$1D8

PR1 \$1D9

PR2 \$1DA

PR3 \$1DB

PR4 \$1DC

PR5 \$1DD

PR6 \$1DE

PR7 \$1DF

14.0.4) Program Blocks

Spin2 programs are built from one or more objects. Objects are files which contain at least one public method, along with optional constants, child objects, variables, additional methods, and data. Objects are assembled together into a top-level object with an internal hierarchy of sub-objects. Each object instance, at run-time, gets its own set of variables, as defined by the object, to maintain its unique operating state.

Different parts of an object are declared within blocks, which all begin with 3-letter block identifiers.

The compiler can actually generate PASM-only programs, as well as Spin2+PASM programs, depending upon which blocks are present in the .spin2 file.

Block Identifier	Block Contents	Spin2+PASM Programs	PASM-only Programs
CON	Constant declarations (CON is the initial/default block type)	Permitted	Permitted
OBJ	Child-object instantiations	Permitted	Not Allowed
VAR	Variable declarations	Permitted	Not Allowed
PUB	Public method for use by the parent object and within this object	Required	Not Allowed
PRI	Private method for use within this object	Permitted	Not Allowed
DAT	Data declarations, including PASM code	Permitted	Required

Here are some minimal Spin2 and PASM-only programs. If you copy and paste these into PNut.exe, you can hit F10 to run them.

Minimal Spin2 Program	<pre> PUB MinimalSpin2Program() 'first PUB method executes REPEAT PINWRITE(63..56, GETRND()) 'write a random pattern to P63..P56 WAITMS(100) 'wait 1/10th of a second, loop </pre>
Minimal PASM Program	<pre> DAT ORG 'start PASM at hub \$00000 for cog \$000 loop DRVRND #56 ADDPINS 7 'write a random pattern to P63..P56 WAITX ##clkfreq_/10 'wait 1/10th of a second, loop JMP #loop </pre>

Here is a Spin2 program which contains every block type.

All- Block Spin2 Program	<pre> CON _clkfreq = 297_000_000 'set clock frequency OBJ vga : "VGA_640x480_text_80x40" 'instantiate vga object VAR time, i 'declare object-wide variables PUB go() 'this first public method executes, cog stops after </pre>
-----------------------------------	---

vga.start(8)	'start vga on base pin 8
SEND := @vga.print	'establish SEND pointer
SEND(4, \$004040, 5, \$00FFFF)	'set light cyan on dark cyan
time := GETCT()	'capture time
i := @text	'print file to vga screen
REPEAT @textend-i	
SEND(byte[i++])	
time := GETCT() - time	'capture time delta in clock cycles
time := MULDIV64(time, 1_000_000, clkfreq)	'get time delta in microseconds
SEND(12, "Time elapsed during printing was ", dec(time), " microseconds.")	'print time delta
PR dec(value) flag, place, digit	'private method prints decimals, three local variables
flag~	'reset digit-printed flag

```

place := 1_000_000_000      'start at the one-billion's place and work
downward

REPEAT

  IF flag ||= (digit := value / place // 10) || place == 1    'print a digit?

    SEND("0" + digit)      'yes

    IF LOOKDOWN(place : 1_000_000_000, 1_000_000, 1_000)    'also print a
comma?

    SEND(",")              'yes

  WHILE place /= 10        'next place, done?

DAT

text    FILE    "VGA_640x480_text_80x40.txt"    'include raw file data for
printing

textend

```

14.1) CON Block

Symbolic constants are global to the object. If an object reference is declared in another object constants of the child object can be referenced using :

OBJ

Num : "Numbers" → Num.Constant_Symbol

Syntax 1 Symbol = Expression (constants)

Symbol –desired name of constant

Expression –any valid integer for floating point, or constant algebraic expression

Note: Constant can be used in algebraic expression but must be previously defined.

- Symbolic constants resolve to 32-bit values.
- Symbolic constants can be assigned using '=' or by just expressing their names in an enumeration list.
- Symbolic constants can be referenced by every block within the file, including CON blocks.

- Symbolic constants can be referenced by the parent object's methods via 'objectname.constantname' syntax.
- If a decimal point is present, the value is encoded in IEEE-754 single-precision format.

CON 'Direct Assignment

EnableFlow = 8 'single assignments LONG data type

x = 5, y = -5, z = 1 'comma-separated assignments

HalfPi = 1.5707963268 'single-precision float values

xy = x*y

Syntax 2 #Symbol (enumerated constants)

CON 'Direct Assignment

#3, a,b,c 'a=3 b=4 c=5precision float values

14.1.1) CON Compiler Enumeration Step option #conVar[step]

#0,a,b,c,d 'a=0, b=1, c=2, d=3 (start=0, step=1)

#1,e,f,g,h 'e=1, f=2, g=3, h=4 (start=1, step=1)

#4[2],i,j,k,l 'i=4, j=6, k=8, l=10 (start=4, step=2)

#-1[-1],m,n,p 'm=-1, n=-2, p=-3 (start=-1, step=-1)

#true,on,off 'true = \$FFFF_FFFF ,on = \$FFFF_FFFF,off = %0

#true +true,on,off 'true = \$FFFF_FFFE ,on = \$FFFF_FFFE,off = \$FFFF_FFFF

14.1.2) Vertical Constant Enumeration

Note: Constant[Step Increment] changes the step value for next constant if not defined the default step is 1 for example conVar[3] would increase next step value by 3

#16 'start=16, step=1 set enumeration

q 'q=16

r[0] 'r=17 ([0] is a step multiplier)

s 's=17

t 't=18

u[2] 'u=19 ([2] is a step multiplier)

v 'v=21

w 'w=22

#16[2] 'start=16 step=2 set enumeration

a 'a= 16

b 'b= 18

c 'c= 20

d 'd= 22

e 'e= 24

14.1_Example_WRD_Constant_Enumeration

```

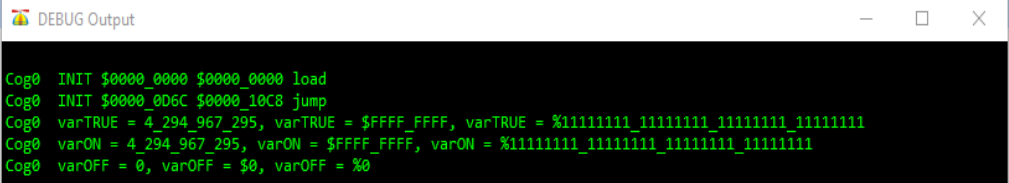
CON
_clkfreq = 200_000_000      'debug must have a clock greater than 10MHZ

CON
#TRUE,on,off                'constant is evaluated to a value and enumeration begins with this

VAR
long varTRUE
long varON
long varOFF

PUB public_method_name()
varTRUE := true
varON := on
varOFF := off
debug(udec(varTRUE),uhex(varTRUE),ubin(varTRUE))
debug(udec(varON),uhex(varON),ubin(varON))
debug(udec(varOFF),uhex(varOFF),ubin(varOFF))
repeat

```



The #TRUE directive causes the enumeration constant value to be evaluated and then assigned to constants.

The case of “true or True or tRue” all are legal and will resolve to **TRUE**. Case does not matter in spin2.

14.2) OBJ Block

Symbol<[count]>: "Object_Name"

Symbol –name for Object

count –number of objects to be made

Object_Name –object filename without extension

- Up to 32 different child objects can be incorporated into a parent object.
- Child objects can be instantiated singularly or in arrays of up to 255.
- Up to 1024 child objects are allowed per parent object.

OBJ

NUM: "Numbers"

Term: "Terminal"

Public Methods are accessed alias dot method: NUM.ToStr

OBJ	OBJ vga : "VGA_Driver" 'instantiate "VGA_Driver.spin2" as "vga"
Child-Object	mouse : "USB_Mouse" 'instantiate "USB_Mouse.spin2" as "mouse"
Instantiations	v[16] : "VocalSynth" 'instantiate an array of 16 objects '..v[0] through v[15]

14.2.1) Including Objects during Compiling

The example used as a child object is the "14.2_Example_WRD_FullDuplexSerial.spin2". This object is used to communicate with PST(Parallax Serial Terminal) which is launched from the "Propeller Tool".

The baud rate is 230400 bps and the com port is to match the loading for programs over pins :

RX1 = 63 { I } rogramming / debug

TX1 = 62 { O }

"14.2_Example_WRD_FullDuplexSerial_Demo.spin2" declares the Object "term" to be compiled with the program:

```
obj
term : "14.2_Example_WRD_FullDuplexSerial"          'serial IO for terminal
```

This is similar to an include file that a "C" program would use. The "spin2" extension is not included in the object declaration. The "Propeller Tool" checks the library folder and the folder containing the calling program for a file with the name and spin2 extension. Suggest keep all objects in the same folder as the parent program.

14.2.2) Accessing Constants and Pub Method from Objects

From within a parent-object method, a child-object method can be called by using the syntax:

`object_name.method_name({any_parameters})`

<code>term.dec(123)</code>	'Send character string representing decimal value 123
<code>term.fhex(\$FFFF,8)</code>	'Send string representing hex value \$FFFF padded with 0 for 8 digits 0000FFFF

From within a parent-object method, a child-object constant can be referenced by using the syntax:

`object_name.constant_name`

<code>term.tx(term.LF)</code>	'Send Constant in object term for Line Feed term.LF =10
<code>term.tx(term.CR)</code>	'Send Constant in object term for Carriage Return term.CR =11

14.1_Example_WRD_SmartSerial_Demo

This demo calls "1.1_Example_WRD_SmartSerial" as object and uses " Propeller IDE terminal program PST (Parallax Serial Terminal) check baud rate to match program. The PST recognizes the following screen control Characters

Parallax Serial Terminal

```
HOME   = 1 , CRSR_XY = 2, CRSR_LF = 3, CRSR_RT = 4, CRSR_UP = 5
CRSR_DN = 6 , BELL   = 7, BKSP   = 8, TAB    = 9, LF     = 10
CLR_EOL = 11 , CLR_DN = 12, CR     = 13, CRSR_X  = 14, CRSR_Y = 15
CLS     = 16
```

14.2_Example_WRD_FullDuplexSerial_Demo

```
{{14.2_Example_WRD_FullDuplexSerial_Demo}}
{{
```

```
=====
" File..... P2USB-format_strings_demo
" Based on jm_formatted_strings_test.spin2
" Objects: term : "jm_fullduplexserial"
" Secondary Object: nstr : "jm_nstr"
" Purpose.... Demonstrate Propeller II Serial
" Author..... WRD Copyright (c) 2020 Bob Drury
"      -- see below for terms of use
" E-mail..... bob_drury@hotmail.com
" Rev01..... 2020 Jan 5
=====
```

```
Object term: "jm_fullduplexserial"
```

```
Public Function Call
```

```
term.null()
term.tstart(baud) : result
term.start(rxpin, txpin, mode, baud) : result
term.stop()
term.rx() : b
term.rxcheck() : b
term.rxtime(ms) : b
term.rxtix(tix) : b
term.available() : count
term.rxflush()
term.tx(b)      "send single byte b to PST"
term.txn(b, n)  "send single byte b for n times to PST"
term.str(p_str) '
term.substr(p_str, len)
term.padstr(p_str, width, pad)
term.txflush()
term.fstr0(p_str) "
term.fstr1(p_str, arg1)
term.fstr2(p_str, arg1, arg2)
term.fstr3(p_str, arg1, arg2, arg3)
term.fstr4(p_str, arg1, arg2, arg3, arg4)
term.fstr5(p_str, arg1, arg2, arg3, arg4, arg5)
term.fstr6(p_str, arg1, arg2, arg3, arg4, arg5, arg6)
term.format(p_str, p_args)
term.lower(c) : result
term.fmt_number(value, base, digits, width, pad)
term.dec(value)      "Send string of characters representing decimal value to PST"
term.fdec(value, digits) "Send characters representing decimal value padded with 0 to digits PST"
term.jdec(value, digits, width, pad) "Send characters 8 digits with width padded to complete with pad
term.dpdec(value, dp) "Send value as decimal with decimal point of dp size
```

```

term.jdpdec(value, dp, width, pad)
term.hex(value)
term.fhex(value, digits)
term.jhex(value, digits, width, pad)
term.oct(value)
term.foct(value, digits)
term.joct(value, digits, width, pad)
term.qrt(value)
term.fqrt(value, digits)
term.jqrt(value, digits, width, pad)
term.bin(value)
term.fbin(value, digits)
term.jbin(value, digits, width, pad)
RX1   = 63 { I } Pin for serial input
TX1   = 62 { O } Pin for serial output

```

Parallax Serial Terminal

```

term.HOME   = 1
term.CRSR_XY = 2
term.CRSR_LF = 3
term.CRSR_RT = 4
term.CRSR_UP = 5
term.CRSR_DN = 6
term.BELL   = 7
term.BKSP   = 8
term.TAB     = 9
term.LF      = 10
term.CLR_EOL = 11
term.CLR_DN  = 12
term.CR      = 13
term.CRSR_X  = 14
term.CRSR_Y  = 15
term.CLS     = 16

```

Formatted Arguments

```

%w.pf      print argument as decimal width decimal point
%[w[.p]]d  print argument as decimal
%[w[.p]]u  print argument as unsigned decimal
%[w[.p]]x  print argument as hex
%[w[.p]]o  print argument as octal
%[w[.p]]q  print argument as quaternary
%[w[.p]]b  print argument as binary
%[w]s      print argument as string
%[w]c      print argument as character (

```

-- w is field width

* positive w causes right alignment in field

- * negative w causes left alignment in field
- %ws aligns s in field (may truncate)
- %wc prints w copies of c
- p is precision characters
- * number of characters to use, aligned in field
 - prefix with 0 if needed to match p
 - for %w.pf, p is number of digits after decimal point

Escaped characters

\\	backslash char
\%	percent char
\q	double quote
\b	backspace
\t	tab (horizontal)
\n	new line (vertical tab)
\r	carriage return
\nnn	arbitrary ASCII value (nnn is decimal)

}}

con { timing }

_clkfreq = 200_000_000	' set system clock
BR_TERM = 230_400	' terminal baud rate

'set Parallax Serial Terminal 230_400 Baud

con { fixed io pins }

RX1 = 63 { I }	' programming / debug
TX1 = 62 { O }	

FS_CS = 61 { O }	' flash storage
FS_SCLK = 60 { O }	
FS_MOSI = 59 { O }	
FS_MISO = 58 { I }	

SD_SCLK = 61 { O }	' usd card storage
SD_CS = 60 { O }	
SD_MOSI = 59 { O }	
SD_MISO = 58 { I }	

SDA1 = 57 { IO }	' i2c (optional)
SCL1 = 56 { IO }	

con

BUF_SIZE = 32	'input character buffer size
---------------	------------------------------

```

obj
    term : "14.2_Example_WRD_FullDuplexSerial"          ' * serial IO for terminal
var
    Byte buffer[BUF_SIZE]
    Long x00,x01,x02,x03,x04,x05
    Byte C01,C02,C03,C04,C05
    Long xFloat
dat
    Device    byte    "P2X8C4M64P\r", 0
    Arg00     byte    "Arg00",0
    Arg01     byte    "Arg01",0
    Arg02     byte    "Arg02",0
    Arg03     byte    "Arg03",0
    Arg04     byte    "Arg04",0
    Arg05     byte    "Arg05",0
    Char01    byte    "A"
    Char02    byte    "B"
    Char03    byte    "C"
    Char04    byte    "D"
    Char05    byte    "E"
pub main() | x, y

    setup()

    wait_for_terminal(true)

    term.fstr1(string("%s Formatted Strings Demo\r"), @Device)
    term.fstr1(string("%033c\r\r"), "-")
    term.fstr0(string("Enter to run Program: "))
    get_str(BUF_SIZE-2)
    term.fstr1(string("\r\rHello, %s, let me show you some \rformatted strings...\r\r"), @buffer)
    waitms(1000)
    repeat 10
        term.tx("a")
    term.tx(term.LF)    'Send Constant in object term for Line Feed term.LF
    term.tx(term.CR)    'Send Constant in object term for Carriage Return term.CR
    term.str(string("Hello Universe")) 'Send zero inline terminated string
    term.tx(term.LF)
    term.tx(term.CR)
    waitms(1000)
    term.dec(123)        'Send character string representing decimal value 123
    term.tx(term.LF)
    term.tx(term.CR)
    term.fdec(123,8)     'Send c string representing decimal value 123 padded with 0 for 8 digits 00000123
    term.tx(term.LF)
    term.tx(term.CR)

```

```

term.jdec(-123,8,11,"*") 'Total length of 11 with 8 digits padded with "*"
term.tx(term.LF)
term.tx(term.CR)
term.dpdec(12300, 2)      'Send character string representing decimal with 2 decimal place
term.tx(term.LF)
term.tx(term.CR)
term.jdpdec(54312, 2, 8, "*") 'Send string value with dp decimal total length width padding with pad
term.tx(term.LF)
term.tx(term.CR)
term.str(@Device)         'Send zero terminated string ignores format escape sequence
term.tx(term.LF)
term.tx(term.CR)
term.hex(255)              'Send character string representing hex value for decimal 255
term.tx(term.LF)
term.tx(term.CR)
term.hex($FF)             'Send character string representing hex value $FF
term.tx(term.LF)
term.tx(term.CR)
term.fhex($FFFF,8)        'Send string representing hex value $FFFF padded with 0 for 8 digits 0000FFFF
term.tx(term.LF)
term.tx(term.CR)
term.jhex($256,8,11,"*")  'Convert Decimal to hex of 11 with 8 digits padded with "*"
term.tx(term.LF)
term.tx(term.CR)
term.oct(15)               'Convert Decimal to oct and send character string
term.tx(term.LF)
term.tx(term.CR)
term.oct($FF)              'Convert Hex to Oct and send character string
term.tx(term.LF)
term.tx(term.CR)
term.foct($3FF,5)          'Convert Hex to Oct and send character string ($3FF = 1023 = 1777 octal)
term.tx(term.LF)
term.tx(term.CR)
term.joct($3FF,8,11,"*")  'Convert Hex to Oct send string of 11 with 8 digits padded with "*"
term.tx(term.LF)
term.tx(term.CR)
term.qrt(85)               'Convert 85 decimal to quaternary 1111 Base 4
term.tx(term.LF)
term.tx(term.CR)
term.fqrt($55,8)           'Convert hex $55 to quaternary 1111 Base 4
term.tx(term.LF)
term.tx(term.CR)
term.jqrt($55,8,11,"*")   'Convert hex $55 to qrt send string of 11 with 8 digits padded with "*"
term.tx(term.LF)
term.tx(term.CR)
term.bin(16)               'Convert 16 decimal to binary 10000 Base 2
term.tx(term.LF)

```



```

term.tx(term.CR)
term.fbin($10,8)      'Convert hex $10 to binary 10000 Base2 with 0 padded 8 digits
term.tx(term.LF)
term.tx(term.CR)
term.jbin($10,8,11,"*") 'Convert hex $F to binary send character string of 11 with 8 digits padded with
"*"

term.tx(term.LF)
term.tx(term.CR)
waitms(1000)
C01 := 65      'ASCII Dec 65 = A
term.tx(C01)   'Send Var C01
term.tx(term.LF) 'Send Constant in object term for Line Feed term.LF
C01 := 66      'ASCII Dec 66 = B
term.tx(C01)   'Send Var C01
term.tx(term.LF) 'Send Constant in object term for Line Feed term.LF
term.tx(term.CR) 'Send Constant in object term for Carriage Return term.CR
term.tx("C")    'Send Character C
term.tx(10)     'Send Number 10 to PST which causes LF
term.tx(13)     'Send Number 13 to PST which causes CR
term.txn("∞",45) 'Send character A for 45 times
term.fstr0(string("\r")) 'Send CR and LF
term.tx(term.LF) 'Send Constant in object term for Line Feed term.LF
term.tx(term.CR) 'Send Constant in object term for Carriage Return term.CR
term.fstr0(string("\176")) 'Send ASCII degree symbol° Use character chart to print
term.tx(10)     'Send Number 10 to PST which causes LF
term.tx(13)     'Send Number 13 to PST which causes CR
waitms(1000)
term.fstr0(@Device) 'Send string from Dat Address @Device
term.fstr1(string("fstr01 %s\r"),@Arg01)
term.fstr2(string("fstr02 %s %s\r"),@Arg01,@Arg02)
term.fstr3(string("fstr03 %s %s %s\r"),@Arg01,@Arg02,@Arg03)
term.fstr4(string("fstr04 %s %s %s %s\r"),@Arg01,@Arg02,@Arg03,@Arg04)
term.fstr5(string("fstr05 %s %s %s %s %s\r"),@Arg01,@Arg02,@Arg03,@Arg04,@Arg05)
waitms(1000)
x01 := 101
x02 := 102
x03 := 103
x04 := 104
x05 := 105
xFloat := 12.345
term.fstr1(string("xFloat= %f\r"),xFloat)
term.fstr1(string("x01= %d\r"),x01)
term.fstr2(string("x01= %d x02= %d\r"),x01,x02)
term.fstr3(string("x01= %d x02= %d x03= %d\r"),x01,x02,x03)
term.fstr4(string("x01= %d x02= %d x03= %d x04= %d\r"),x01,x02,x03,x04)
term.fstr5(string("x01= %d x02= %d x04= %d x04= %d x05= %d\r"),x01,x02,x03,x04,x05)
waitms(1000)

```

```

term.fstr1(string("Char01= %c\r"),Char01)
term.fstr2(string("Char01= %c Char02= %c\r"),Char01,Char02)
term.fstr3(string("Char01= %c Char02= %c Char03= %c\r"),Char01,Char02,Char03)
term.fstr4(string("Char01= %c Char02= %c Char03= %c Char04= %c\r"),Char01,Char02,Char03,Char04)
term.fstr5(string("Char01= %c Char02= %c Char04= %c Char04= %c Char05=
%c\r"),Char01,Char02,Char03,Char04,Char05)
waitms(1000)
C01 := "A"
C02 := "B"
C03 := "C"
C04 := "D"
C05 := "E"
term.fstr1(string("C01= %c\r"),C01)
term.fstr2(string("C01= %c C02= %c\r"),C01,C02)
term.fstr3(string("C01= %c C02= %c C03= %c\r"),C01,C02,C03)
term.fstr4(string("C01= %c C02= %c C03= %c C04= %c\r"),C01,C02,C03,C04)
term.fstr5(string("C01= %c C02= %c C04= %c C04= %c C05= %c\r"),C01,C02,C03,C04,C05)
waitms(1000)
term.fstr1(string("%040c\r"),"-")
term.fstr1(string("%040c\r")," *")
term.fstr1(string("%040c\r"),176) '176 is the decimal ascii for degree with PST
term.fstr1(string("x01 float= %13.3f\r"),x01)
term.fstr2(string("x01 in hex= %4x --> x01 in dec= %4d\r"),x01,x01)
term.fstr2(string("%d\176C --> %d\176F\r\r"),x01,x01*9/5+32)
term.fstr2(string("%-10d %13.3f\r"), x01, x01)
waitms(1000)
repeat x from 123 to 255
  term.fstr1(string("%040c\r"),x)
waitms(3000)

```

```

pub get_str(maxlen) : len | k
    bytefill(@buffer, 0, BUF_SIZE)          ' clear input buffer
    term.rxflush()                          ' clear trash from terminal
    repeat
        k := term.rx()                      ' wait for a character
        case k
            31..127 :                       ' if valid
                if (len < maxlen)           ' and room
                    buffer[len++] := k      ' add to buffer

        term.BKSP :
            if (len > 0)                    ' if character(s) in buffer
                buffer[--len] := 0          ' backup and erase last

        term.CR :
            buffer[len] := 0                ' terminate string
            return                          ' and return to caller

pub wait_for_terminal(clear)

    term.rxflush()
    term.rx()                              ' wait for keypress
    if (clear)
        term.tx(term.CLS)

pub setup()

    term.start(RX1, TX1, %0000, BR_TERM)   ' start terminal serial
con { license }
{{
    MIT License}}

```

14.2_Example_WRD_FullDuplexSerial

```
{{14.2_Example_WRD_FullDuplexSerial}}
{{
```

```
=====
"
" File..... jm_fullduplexserial.spin2
" Purpose.... Buffered serial communications using smart pins
"      -- mostly matches FullDuplexSerial from P1
"      -- does NOT support half-duplex communications using shared RX/TX pin
" Authors.... Jon McPhalen
"      -- based on work by Chip Gracey
"      -- see below for terms of use
" E-mail..... jon.mcphalen@gmail.com
" Started....
" Updated.... 06 SEP 2020
"
```

```
=====
Note: Buffer size no longer has to be power-of-2 integer.
```

```
Note: The dec(), bin(), and hex() methods will no longer require the digits parameter as
      in older versions of FullDuplexSerial. Use fdec(), fbin(), and fhex() for code that
      requires a specific field width.
```

```
The smart pin uarts use a 16-bit value for baud timing which can limit low baud rates for
some system frequencies -- beware of these limits when connecting to older devices.
```

Baud	20MHz	40MHz	80MHz	100MHz	200MHz	300MHz
300	No	No	No	No	No	No
600	Yes	No	No	No	No	No
1200	Yes	Yes	No	No	No	No
2400	Yes	Yes	Yes	Yes	No	No
4800	Yes	Yes	Yes	Yes	Yes	Yes

```
}}
```

```
con { fixed io pins }
```

```

RX1   = 63 { I }           ' programming / debug
TX1   = 62 { O }
```

```

SF_CS  = 61 { O }           ' serial flash
SF_SCK = 60 { O }
SF_SDO = 59 { O }
SF_SDI = 58 { I }
```

```

con { pst formatting }
  HOME   = 1
  CRSR_XY = 2
  CRSR_LF = 3
  CRSR_RT = 4
  CRSR_UP = 5
  CRSR_DN = 6
  BELL   = 7
  BKSP   = 8
  TAB    = 9
  LF     = 10
  CLR_EOL = 11
  CLR_DN  = 12
  CR      = 13
  CRSR_X  = 14
  CRSR_Y  = 15
  CLS     = 16
Con {FullDuplexSerial}
  BUF_SIZE = 64

```

```
obj
```

```

nstr : "14.2_Example_WRD_Num_To_Str"          ' number-to-string

```

```
var
```

```

long cog                ' cog flag/id

long rxp                ' rx smart pin
long txp                ' tx smart pin
long rxhub              ' hub address of rxbuf
long txhub              ' hub address of txbuf

long rxhead             ' rx head index
long rxtail             ' rx tail index
long txhead             ' tx head index
long txtail             ' tx tail index

long txdelay            ' ticks to transmit one byte

byte rxbuf[BUF_SIZE]    ' buffers
byte txbuf[BUF_SIZE]

byte pbuf[80]           ' padded strings

```

```

pub null()
" This is not a top-level object

pub tstart(baud) : result
" Start FDS with default pins/mode for terminal (e.g., PST)
return start(RX1, TX1, %0000, baud)

pub start(rxp, txp, mode, baud) : result | baudcfg, spmode
" Start simple serial coms on rxp and txp at baud
" -- rxp... receive pin (-1 if not used)
" -- txp... transmit pin (-1 if not used)
" -- mode.... %0xx1 = invert rx
"           %0x1x = invert tx
"           %01xx = open-drain/open-source tx

stop()

if (rxp == txp)                                ' pin must be unique
return false

longmove(@rxp, @rxpin, 2)                      ' save pins
rxhub := @rxbuf                                ' point to buffers
txhub := @txbuf

txdelay := clkfreq / baud * 11                  ' tix to transmit one byte

baudcfg := muldiv64(clkfreq, $1_0000, baud) & $FFFFFFC00 ' set bit timing
baudcfg |= (8-1)                                ' set bits (8)

if (rxp >= 0)                                  ' configure rx pin if used
spmode := P_ASYNC_RX
if (mode.[0])
    spmode |= P_INVERT_IN
pinstart(rxp, spmode, baudcfg, 0)

if (txp >= 0)                                  ' configure tx pin if used
spmode := P_ASYNC_TX | P_OE
case mode.[2..1]
    %01 : spmode |= P_INVERT_OUTPUT
    %10 : spmode |= P_HIGH_FLOAT                ' requires external pull-up
    %11 : spmode |= P_INVERT_OUTPUT | P_LOW_FLOAT ' requires external pull-down
pinstart(txp, spmode, baudcfg, 0)

cog := coginit(COGEXEC_NEW, @uart_mgr, @rxp) + 1 ' start uart manager cog

```

```

return cog

pub stop()

" Stop serial driver
" -- frees a cog if driver was running

if (cog)                                ' cog active?
  cogstop(cog-1)                        ' yes, shut it down
  cog := 0                              ' and mark stopped

longfill(@rxp, -1, 2)                   ' reset object globals
longfill(@rxhub, 0, 7)

pub rx() : b

" Pulls byte from receive buffer if available
" -- will wait if buffer is empty

repeat while (rxtail == rxhead)          ' hold while buffer empty

b := rxbuf[rxtail]                       ' get a byte
if (++rxtail == BUF_SIZE)                 ' update tail pointer
  rxtail := 0

pub rxcheck() : b

" Pulls byte from receive buffer if available
" -- returns -1 if buffer is empty

if (rxtail <> rxhead)                     ' something in buffer?
  b := rxbuf[rxtail]                     ' get it
  if (++rxtail == BUF_SIZE)               ' update tail pointer
    rxtail := 0
else
  b := -1                                ' mark no byte available

pub rxtime(ms) : b | mstix, t

" Wait ms milliseconds for a byte to be received
" -- returns -1 if no byte received, $00..$FF if byte

mstix := clkfreq / 1000

```

```
t := getct()
repeat until ((b := rxcheck()) >= 0) || (((getct()-t) / mstix) >= ms)
```

```
pub rxtix(tix) : b | t
```

```
" Waits tix clock ticks for a byte to be received
"-- returns -1 if no byte received
```

```
t := getct()
repeat until ((b := rxcheck()) >= 0) || ((getct()-t) >= tix)
```

```
pub available() : count
```

" Returns # of bytes waiting in rx buffer

```

if (rxtail <= rxhead)          ' if byte(s) available
    count := rxhead - rxtail    ' get count
if (count < 0)
    count += BUF_SIZE          ' fix for wrap around

```

```
pub rxflush()
```

" Flush receive buffer

```
repeat while (rxcheck() >= 0)
```

```
pub tx(b) | n
```

```
" Move byte into transmit buffer if room is available
```

" -- will wait if buffer is full

```
repeat
  n := txhead - txtail
  if (n < 0)
    n += BUF_SIZE
  if (n < BUF_SIZE-1)
    quit
```

```
txbuf[txhead] := b           ' move to buffer
if (++txhead == BUF_SIZE)    ' update head pointer
    txhead := 0
```



```

pub txn(b, n)
" Emit byte n times
repeat n
    tx(b)

pub str(p_str)
" Emit z-string at p_str
repeat (strsize(p_str))
    tx(byte[p_str++])

pub substr(p_str, len) | b
" Emit len characters of string at p_str
" -- aborts if end of string detected
repeat len
    b := byte[p_str++]
    if (b > 0)
        tx(b)
    else
        quit

pub padstr(p_str, width, pad)
" Emit p_str as padded field of width characters
" -- pad is character to use to fill out field
" -- positive width causes right alignment
" -- negative width causes left alignment
str(nstr.padstr(p_str, width, pad))

pub txflush()
" Wait for transmit buffer to empty
" -- will delay one byte period after buffer is empty
repeat until (txtail == txhead)          ' let buffer empty
    waitct(getct() + txdelay)            ' delay for last byte

pub fstr0(p_str)
" Emit string with formatting characters.
format(p_str, 0)

pub fstr1(p_str, arg1)
" Emit string with formatting characters and one argument.
format(p_str, @arg1)

pub fstr2(p_str, arg1, arg2)
" Emit string with formatting characters and two arguments.
format(p_str, @arg1)

```

```
pub fstr3(p_str, arg1, arg2, arg3)
" Emit string with formatting characters and three arguments.
  format(p_str, @arg1)

pub fstr4(p_str, arg1, arg2, arg3, arg4)
" Emit string with formatting characters and four arguments.
  format(p_str, @arg1)

pub fstr5(p_str, arg1, arg2, arg3, arg4, arg5)
" Emit string with formatting characters and five arguments.
  format(p_str, @arg1)

pub fstr6(p_str, arg1, arg2, arg3, arg4, arg5, arg6)
" Emit string with formatting characters and six arguments.
  format(p_str, @arg1)
```

```

pub format(p_str, p_args) | idx, c, asc, field, digits
" Emit formatted string with escape sequences and embedded values
" -- p_str is a pointer to the format control string
" -- p_args is pointer to array of longs that hold field values
" * field values can be numbers, characters, or pointers to strings
idx := 0                                ' value index
repeat
  c := byte[p_str++]
  if (c == 0)
    return
  elseif (c == "\\")
    c := lower(byte[p_str++])
    if (c == "\\")
      tx("\\")
    elseif (c == "%")
      tx("%")
    elseif (c == "q")
      tx(34)
    elseif (c == "b")
      tx(BKSP)
    elseif (c == "t")
      tx(TAB)
    elseif (c == "n")
      tx(LF)
    elseif (c == "r")
      tx(CR)
    elseif ((c >= "0") and (c <= "9"))
      --p_str
      p_str, asc, _ := get_nargs(p_str)
      if ((asc >= 0) and (asc <= 255))
        tx(asc)
    elseif (c == "%")
      p_str, field, digits := get_nargs(p_str)
      c := lower(byte[p_str++])
      if (c == "d")
        str(nstr.fmt_number(long[p_args][idx++], "d", digits, field, " "))
      elseif (c == "u")
        str(nstr.fmt_number(long[p_args][idx++], "u", digits, field, " "))
      elseif (c == "f")
        str(nstr.fmt_number(long[p_args][idx++], "f", digits, field, " "))
      elseif (c == "b")
        str(nstr.fmt_number(long[p_args][idx++], "b", digits, field, " "))
      elseif (c == "q")
        str(nstr.fmt_number(long[p_args][idx++], "q", digits, field, " "))
      elseif (c == "o")
        str(nstr.fmt_number(long[p_args][idx++], "o", digits, field, " "))
      elseif (c == "x")

```

```

    str(nstr.fmt_number(long[p_args][idx++], "x", digits, field, " "))
elseif (c == "s")
    str(nstr.padstr(long[p_args][idx++], field, " "))
elseif (c == "c")
    txn(long[p_args][idx++], (abs field) #> 1)
else
    tx(c)

pub lower(c) : result
if ((c >= "A") && (c <= "Z"))
    c += 32
return c

pri get_nargs(p_str) : p_str1, val1, val2 | c, sign
" Parse one or two numbers from string in n, -n, n.n, or -n.n format
" -- dpoint separates values
" -- only first # may be negative
" -- returns pointer to 1st char after value(s)
c := byte[p_str] ' check for negative on first value
if (c == "-")
    sign := -1
    ++p_str
else
    sign := 0
repeat ' get first value
    c := byte[p_str++]
    if ((c >= "0") && (c <= "9"))
        val1 := (val1 * 10) + (c - "0")
    else
        if (sign)
            val1 := -val1
        quit
if (c == ".") ' if dpoint
    repeat ' get second value
        c := byte[p_str++]
        if ((c >= "0") && (c <= "9"))
            val2 := (val2 * 10) + (c - "0")
        else
            quit
p_str1 := p_str - 1 ' back up to non-digit

```

```
pub fmt_number(value, base, digits, width, pad)
" Emit value converted to number in padded field
" -- value is converted using base as radix
" * 99 for decimal with digits after decimal point
" -- digits is max number of digits to use
" -- width is width of final field (max)
" -- pad is character that fills out field
  str(nstr.fmt_number(value, base, digits, width, pad))

pub dec(value)
" Emit value as decimal
  str(nstr.itoa(value, 10, 0))

pub fdec(value, digits)
" Emit value as decimal using fixed # of digits
" -- may add leading zeros
  str(nstr.itoa(value, 10, digits))

pub jdec(value, digits, width, pad)
" Emit value as decimal using fixed # of digits
" -- aligned in padded field (negative width to left-align)
" -- digits is max number of digits to use
" -- width is width of final field (max)
" -- pad is character that fills out field
  str(nstr.fmt_number(value, "d", digits, width, pad))

pub dpdec(value, dp)
" Emit value as decimal with decimal point
" -- dp is number of digits after decimal point
  str(nstr.dpdec(value, dp))

pub jdpdec(value, dp, width, pad)
" Emit value as decimal with decimal point
" -- aligned in padded field (negative width to left-align)
" -- dp is number of digits after decimal point
" -- width is width of final field (max)
" -- pad is character that fills out field
  str(nstr.fmt_number(value, "f", dp, width, pad))
```

```
pub hex(value)
" Emit value as hexadecimal
str(nstr.itoa(value, 16, 0))

pub fhex(value, digits)
" Emit value as hexadecimal using fixed # of digits
str(nstr.itoa(value, 16, digits))

pub jhex(value, digits, width, pad)
" Emit value as quaternary using fixed # of digits
" -- aligned inside field
" -- pad fills out field
str(nstr.fmt_number(value, "x", digits, width, pad))

pub oct(value)
" Emit value as octal
str(nstr.itoa(value, 8, 0))

pub foct(value, digits)
" Emit value as octal using fixed # of digits
str(nstr.itoa(value, 8, digits))

pub joct(value, digits, width, pad)
" Emit value as octal using fixed # of digits
" -- aligned inside field
" -- pad fills out field
str(nstr.fmt_number(value, "o", digits, width, pad))

pub qrt(value)
" Emit value as quaternary
str(nstr.itoa(value, 4, 0))

pub fqrt(value, digits)
" Emit value as quaternary using fixed # of digits
str(nstr.itoa(value, 4, digits))

pub jqrt(value, digits, width, pad)
" Emit value as quaternary using fixed # of digits
" -- aligned inside field
" -- pad fills out field
str(nstr.fmt_number(value, "q", digits, width, pad))

pub bin(value)
" Emit value as binary
str(nstr.itoa(value, 2, 0))
```

```

pub fbin(value, digits)
" Emit value as binary using fixed # of digits
str(nstr.itoa(value, 2, digits))

pub jbin(value, digits, width, pad)
" Emit value as binary using fixed # of digits
" -- aligned inside field
" -- pad fills out field
str(nstr.fmt_number(value, "b", digits, width, pad))
dat { smart pin uart/buffer manager }

    org

uart_mgr    setq    #4-1                ' get 4 parameters from hub
            rdlong  rxd, ptra

uart_main   testb   rxd, #31             wc   ' rx in use?
            if_nc   call   #rx_serial

            testb   txd, #31             wc   ' tx in use?
            if_nc   call   #tx_serial

            jmp     #uart_main

rx_serial   testp   rxd                  wc   ' anything waiting?
            if_nc   ret

            rdpin   t3, rxd              ' read new byte
            shr     t3, #24              ' align lsb
            mov     t1, p_rxbuf          ' t1 := @rxbuf
            rdlong  t2, ptra[4]          ' t2 := rxhead
            add     t1, t2
            wrbyte  t3, t1              ' rxbuf[rxhead] := t3
            incmod  t2, #(BUF_SIZE-1)    ' update head index
            _ret_   wrlong  t2, ptra[4]   ' write head index back to hub

tx_serial   rdpin   t1, txd              wc   ' check busy flag
            if_c     ret                  ' abort if busy

            rdlong  t1, ptra[6]          ' t1 = txhead
            rdlong  t2, ptra[7]          ' t2 = txtail
            cmp     t1, t2              wz   ' byte(s) to tx?
            if_e     ret

```

```

        mov     t1, p_txbuf          ' start of tx buffer
        add     t1, t2              ' add tail index
        rdbYTE  t3, t1              ' t3 := txbuf[txtail]
        wypin   t3, txd             ' load into sp uart
        incmod  t2, #(BUF_SIZE-1)   ' update tail index
_ret_      wrlong  t2, ptr[7]       ' write tail index back to hub

```

```

'-----

```

```

rx      res     1                  ' receive pin
tx      res     1                  ' transmit pin
p_rxbuf res     1                  ' pointer to rxbuf
p_txbuf res     1                  ' pointer to txbuf

```

```

t1      res     1                  ' work vars
t2      res     1
t3      res     1

```

```

fit     472

```

```

con { license }

```

```

{{

```

```

    Terms of Use: MIT License

```

```

}}

```


14.2_Example_WRD_NUM_To_STR

```

{{14.2_Example_WRD_NUM_To_STR}}
{{
=====
"
" File..... WRD_nstr.spin2
" Purpose.... Convert numbers to strings
" Authors.... Jon McPhalen
"      -- Copyright (c) 2020 Jon McPhalen
"      -- see below for terms of use
" E-mail..... jon.mcphalen@gmail.com
" Started....
" Updated.... 29 AUG 2020
"
=====
}}
con
  NBUF_SIZE = 48
  PBUF_SIZE = 128
var
  byte nbuf[NBUF_SIZE]          ' number conversions
  byte pbuf[PBUF_SIZE]          ' padded strings

```

```

pub null()
" This is not a top level object

pub fmt_number(value, radix, digits, width, pad) : p_str    ' *** changed 19 AUG 2020 ***
" Return pointer to string of value converted to number in padded field
" -- value is converted using radix
" -- radix is character indicating type          ' *** used to be number ***
" -- digits is max number of digits to use
" -- width is width of final fields (max)
" -- pad is character used to pad final field (if needed)
case radix
  "d", "D" : p_str := padstr(itoa(value, 10, digits), width, pad)
  "u", "U" : p_str := padstr(usdec(value, digits), width, pad)
  "f", "F" : p_str := padstr(dpdec(value, digits), width, pad)
  "b", "B" : p_str := padstr(itoa(value, 2, digits), width, pad)
  "q", "Q" : p_str := padstr(itoa(value, 4, digits), width, pad)
  "o", "O" : p_str := padstr(itoa(value, 8, digits), width, pad)
  "x", "X" : p_str := padstr(itoa(value, 16, digits), width, pad)
  other   : p_str := string("?")

```

```

pub dec(value, digits) : p_str | sign, len
" Convert decimal value to string
" -- digits is 0 (auto size) to 10
p_str := itoa(value, 10, digits)

pub usdec(value, digits) : p_str | len
" Convert unsigned decimal value to string
" -- digits is 0 (auto size) to 10
digits := 0 #> digits <# 10          ' limit printable digits
bytefill(@nbuf, 0, NBUF_SIZE)        ' clear buffer
p_str := @nbuf + 9                    ' point to end of udec string
len := 0
repeat
  byte[--p_str] := (value +// 10) + "0" ' extract digit, convert to ASCII
  value += 10                          ' remove digit from value
  if (digits)                          ' length limited?
    if (++len == digits)                ' check size
      quit
  else
    if (value == 0)                    ' done?
      quit

pub dpdec(value, dp) : p_str | len, byte scratch[12]
" Convert value to string with decimal point
" -- dp is digits after decimal point
" -- returns pointer to updated fp string
" -- modifies original string
" -- return pointer to converted string
p_str := itoa(value, 10, 0)
if (dp <= 0)                          ' abort if no decimal point
  return p_str
len := strsize(p_str)                 ' digits
bytefill(@scratch, 0, 12)             ' clear scratch buffer
if (value < 0)                         ' ignore "-" if present
  ++p_str
  --len
if (len < (dp+1))                     ' insert 0s?
  bytemove(@scratch, p_str, len)       ' move digits to scratch buffer
  bytefill(p_str, "0", dp+2-len)      ' pad string with 0s
  bytemove(p_str+dp+2-len, @scratch, len+1) ' move digits back
  byte[p_str+1] := "."                 ' insert dpoint
else
  bytemove(@scratch, p_str+len-dp, dp) ' move decimal part to buffer
  byte[p_str+len-dp] := "."            ' insert dpoint
  bytemove(p_str+len-dp+1, @scratch, dp+1) ' move decimal part back
if (value < 0)                        ' fix pointer for negative #s
  --p_str

```

```

pub itoa(value, radix, digits) : p_str | sign, len, d
" Convert signed integer to string
" -- supports radix 10, 2, 4, 8, and 16
" -- digits is 0 (auto size) to limit for long using radix
bytefill(@nbuf, 0, NBUF_SIZE)          ' clear buffer
p_str := @nbuf                          ' point to it
case radix                              ' limit printable digits
  02 : digits := 0 #> digits <# 32
  04 : digits := 0 #> digits <# 16
  08 : digits := 0 #> digits <# 11
  10 : digits := 0 #> digits <# 10
  16 : digits := 0 #> digits <# 8
other :
  byte[p_str] := 0
  return
if ((radix == 10) && (value < 0))          ' deal with negative decimals
  if (value == negx)
    sign := 2
    value := posx
  else
    sign := 1
    value := -value
else
  sign := 0
len := 0
repeat
  d := value +// radix                    ' get digit (1s column)
  byte[p_str++] := (d < 10) ? d + "0" : d - 10 + "A"    ' convert to ASCII
  value +/= radix                          ' remove digit
  if (digits)                              ' length limited?
    if (++len == digits)                    ' check size
      quit
  else
    if (value == 0)                        ' done?
      quit
if (sign)
  byte[p_str++] := "-"                      ' add sign if needed
  if (sign == 2)
    nbuf[0] := "8"                          ' fix negx if needed
byte[p_str++] := 0                          ' terminate string
return revstr(@nbuf)                        ' fix order (reverse)

```

```

pub revstr(p_str) : result | first, len, last
" Reverse the order of characters in a string.
result := first := p_str          ' start
len := strsize(p_str)             ' length
last := first + len - 1           ' end
repeat (len >> 1)                 ' reverse them
    byte[first++], byte[last--] := byte[last], byte[first]

pub padstr(p_str, width, padchar) : p_pad | len
" Pad string with padchar character
" -- positive width uses left pad, negative field width uses right pad
" -- truncate if string len > width
" -- input string is not modified
" -- returns pointer to padded string
bytefill(@pbuf, 0, PBUF_SIZE)     ' clear padded buffer
len := strsize(p_str)              ' get length of input
width := -PBUF_SIZE+1 #> width <# PBUF_SIZE-1 ' constrain to buffer size
if (width > 0)                     ' right-justify in padded field
    if (width > len)
        bytefill(@pbuf, padchar, width-len)
        bytemove(@pbuf+width-len, p_str, len)
        p_pad := @pbuf
    else
        bytemove(@pbuf, p_str+len-width, width) ' truncate to right-most characters
        p_pad := @pbuf
elseif (width < 0)                 ' left-justify in padded field
    width := -width
    if (width > len)
        bytemove(@pbuf, p_str, len)
        bytefill(@pbuf+len, padchar, width-len)
        p_pad := @pbuf
    else
        bytemove(@pbuf, p_str, width)           ' truncate to leftmost characters
        p_pad := @pbuf
else
    p_pad := p_str
con { license }
{{
    Terms of Use: MIT License
}}

```

14.3) VAR Block

VAR is an object Block where variables are defined of 3 data types Byte, Word and Long. The VAR block declares global variables to the object.

```
VAR
  byte a,b,c '8bits
  word a,b,c '16bits
  long a,b,c '32bits
```

- Variables can be longs (32 bits), words (16 bits), and bytes (8 bits).
- Variables can be declared as singles or arrays.
- Variables are packed in memory in the order they are declared, beginning at a long-aligned address.
- Variables are initialized to zero at run time.
- Each object's first 15 longs of variable memory are accessed via special bytecodes for improved efficiency.
- Each instance of an object will require one long, plus its declared amount of VAR space, plus 0..3 bytes to long-align for the next object's variable space.

14.3.1) VAR Compiler Directive

VAR

```
byte temp      'temp is a VAR of type byte at address @temp
byte temp[20]  'reserve 20 bytes at address @temp[0]
```

Program Usage

```
Some_Var := Byte[@MyData] 'read 1 byte from address @MyData
Some_Var := Byte[@MyData][Index++] 'read 1 byte from @MyData+Index offset
```

VAR

```
word temp      'temp is a VAR of type two byte at address @temp
word temp[20]  'reserve 40 bytes at address @temp[0]
```

14.3.2) Program Usage

```
Some_Var := word[@MyData] 'read 2 byte from address @MyData
Some_Var := word[@MyData][Index++] 'read 2 byte from @MyData+Index offset
```

VAR

```
Long temp      'temp is a VAR of type byte at address @temp
Long temp[20]  'reserve 80 bytes at address @temp[0]
```

Program Usage

```
Some_Var := Long[@MyData] 'read 4 byte from address @MyData
```

Some_Var := Long[@MyData][Index++] 'read 4 byte from @MyData+Index offset

14.3.3) Variables that are pre-defined and Permanent

In Spin2, there are both user-defined and permanent variables. The user-defined variable sources are listed below and the permanent variables are shown in the table.

VAR variables (hub)

PUB/PRI parameters, return values, and local variables (hub)

DAT symbols (hub)

Cog registers

Variables (all LONG)	Variable Name	Address or Offset	Description	Useful in Spin2	Useful in Spin2- PASM	Useful in PASM- Only
Hub Locations	CLKMODE	\$00040	Clock mode value	Yes	Yes	No
	CLKFREQ	\$00044	Clock frequency value	Yes	Yes	No
Hub VAR	VARBASE	+0	Object base pointer, @VARBASE is VAR base, used by method-pointer calls	Maybe	No	No
Cog Registers	PR0	\$1D8	Spin2 <-> PASM communication	Yes	Yes	No
	PR1	\$1D9		Yes	Yes	No
	PR2	\$1DA		Yes	Yes	No
	PR3	\$1DB		Yes	Yes	No
	PR4	\$1DC		Yes	Yes	No
	PR5	\$1DD		Yes	Yes	No
	PR6	\$1DE		Yes	Yes	No
	PR7	\$1DF		Yes	Yes	No
	IJMP3	\$1F0	Interrupt JMP's and RET's	No	Yes	Yes
	IRET3	\$1F1		No	Yes	Yes
			Pointer registers			

	IJMP2	\$1F2		No	Yes	Yes
	IRET2	\$1F3	Data pointer passed from COGINIT	No	Yes	Yes
	IJMP1	\$1F4	Code pointer passed from COGINIT	No	Yes	Yes
	IRET1	\$1F5		No	Yes	Yes
			Output enables for P31..P0			
	PA	\$1F6	Output enables for P63..P32	No	Yes	Yes
	PB	\$1F7	Output states for P31..P0	No	Yes	Yes
	PTRA	\$1F8	Output states for P63..P32	No	Yes	Yes
	PTRB	\$1F9	Input states from P31..P0	No	Yes	Yes
			Input states from P63..P32			
	DIRA	\$1FA		Yes	Yes	Yes
	DIRB	\$1FB		Yes	Yes	Yes
	OUTA	\$1FC		Yes	Yes	Yes
	OUTB	\$1FD		Yes	Yes	Yes
	INA	\$1FE		Yes	Yes	Yes
	INB	\$1FF		Yes	Yes	Yes

In Spin2, all variables can be indexed and accessed as bitfields. Additionally, symbolic hub variables can have BYTE/WORD/LONG size overrides:

Variable Usage	Example	Description
Plain	AnyVar HubVar.WORD BYTE[address] REG[register]	Hub or permanent register variable Hub variable with BYTE/WORD/LONG size override Hub BYTE/WORD/LONG by address Register, 'register' may be symbol declared in ORG section

With Index	AnyVar[index] HubVar.BYTE[index] LONG[address][index] REG[register][index]	Hub or permanent register variable with index Hub variable with size override and index Hub BYTE/WORD/LONG by address with index Register with index
With Bitfield	AnyVar.[bitfield] HubVar.LONG.[bitfield] WORD[address].[bitfield] REG[register].[bitfield]	Hub or permanent register variable with bitfield Hub variable with size override and bitfield Hub BYTE/WORD/LONG by address with bitfield Register with bitfield
With Index and Bitfield	AnyVar[index].[bitfield] HubVar.BYTE[index].[bitfield] LONG[address][index].[bitfield] REG[register][index].[bitfield]	Hub or permanent register variable with index and bitfield Hub variable with size override, index, and bitfield Hub BYTE/WORD/LONG by address with index and bitfield Register with index and bitfield

14.3.4) Accessing Bytes of Larger-Sized Variables

Var

Word WordVar

Long LongVar

Pub Main()

WordVar.Byte[0] := 0 '0000_0000

WordVar.Byte[1] := 100 '0110_0100 = 64 + 32 + 4 = 100

'0110_0100_0000_0000 = 16384 + 8192 + 1024 = 25600

LongVar.byte[0] := 25

LongVar.byte[1] := 50

LongVar.byte[2] := 75

LongVar.byte[3] := 100

'01100100_01001011_00110010_00011001 = 1,682,649,625

14.3.1_Example_WRD_VAR

Demonstrates use of Variables

14.4) PUB Block

PUB methodname({parameter{,...}}) {: result{,...}} { | {ALIGNW/ALIGNL} {BYTE/WORD/LONG}
localvar{{count}}{,...}}

PUB ObjectMethod(a,b,c) : result01,result02,result03) | var01,var02,var03

a,b,c are preloaded passed parameters from method call all longs

result01,result02,result03 all longs are cleared to zero before entry and return with a result from method, you must be able to receive the same number of result parameters.

var01,var02,var03 all longs are local parameters and could be pre-loaded from a previous method call you must manually clear this in the method unless you are using this retentive feature.

Note: In the P1 result was automatically understood as the return method value. In the P2 if no return parameter is declared in the method no return values are generated , ie result is not automatic you must declare if you want a return value.

14.4.1) PUB Block Constraints

- PUB methods are available to the parent object, as well as to the object they are defined in.
- PRI methods are available only to the object they are defined in.
- The first PUB method in an object is what executes when that object is run as the top-level object.
- Methods can have from 0 to 127 input parameters, all of which are single longs.
- Methods can have from 0 to 15 output results, all of which are single longs.
- Methods can have up to 64KB of local variables, which can be bytes, words, and longs (default), in both singles and arrays.
- Local variable size overrides (BYTE/WORD) apply only to the variable being declared, not subsequent variables.
- Results are initialized to zero on method entry, while local variables are undefined.
- Parameters, then results, and then local variables are packed into stack memory in the order they are declared.
- In-line PASM code can access the first 16 longs of parameters...results...locals via registers with the same symbolic names.

14.4.1_Example_WRD_Multiple_Result_Method

Note: The common naming convention do not clash because they are generated local to the method, result01,result02,result03 are separate and independent. Compiler keeps track.

14.4.2_Example_WRD_GETRND

14.4.3_Example_WRD_XYPOL (Polar Co-ordinates xy to length,angle32bit)

14.4.4_Example_WRD_POLXY(Polar Co-Ordinates Length,angle32bit to xy)

14.4.5_Example_WRD_ROTXY (Polar Co-Ordinate Rotation)

14.4.6_Example_WRD_QSIN_QCOS

14.4.7_Example_WRD_QSin_QCOS_Simple_Scope

14.4.8_Example_WRD_MULDIV64

14.4.9_Example_WRD_STRING

14.4.10_Example_WRD_REPEAT

14.4.11_Example_WRD_CASE

The CASE construct sequentially compares a target value to a list of possible matches. When a match is found, the related code executes.

Match values/ranges must be indented past the CASE keyword. Multiple match values/ranges can be expressed with comma separators. Any additional lines of code related to the match value/range must be indented past the match value/range:

CASE target	- CASE with target value
<match> : <code>	- match value and code
<indented code>	
<match..match> : <code>	- match range and code
<indented code>	
<match>,<match..match> : <code>	- match value, range, and code
<indented code>	
OTHER : <code>	- optional OTHER case, in case no match found
<indented code>	

14.4.12_Example_WRD_CASE_FAST

CASE_FAST is like CASE, but rather than sequentially comparing the target to a list of possible matches, it uses an indexed jump table of up to 256 entries to immediately branch to the appropriate code, saving time at a possible cost of larger compiled code. If there are only contiguous match values and no match ranges, the resulting code will actually be smaller than a normal CASE construct with more than several match values.

For CASE_FAST to compile, the match values/ranges must be unique constants which are all within 255 of each other.

CASE flag

```

0: CASE_FAST chr
  0:  BYTEFILL(@screen, " ", screen_size)
      col := row := 0
  1:  col := row := 0
  2..7: flag := chr
      RETURN
  8:  IF col
      col--
  9:  REPEAT
      out(" ")
      WHILE col & 7
  10: RETURN
  11: color := $00
  12: color := $80
  13: newline()
  OTHER: out(chr)

2:  col := chr // cols
3:  row := chr // rows
4..7: background0_[flag-$04] := chr << 8
flag := 0

```

14.4.13_Example_WRD_IF_IFNOT_ELSEIF_ELSEIFNOT_ELSE

The IF construct begins with IF or IFNOT and optionally employs ELSEIF, ELSEIFNOT, and ELSE. To all be part of the same decision tree, these keywords must have the same level of indentation.

The indented code under IF or ELSEIF executes if <condition> is not zero. The code under IFNOT or ELSEIFNOT executes if <condition> is zero. The code under ELSE executes if no other indented code executed:

IF / IFNOT <condition>	- Initial IF or IFNOT
<indented code>	
ELSEIF / ELSEIFNOT <condition>	- Optional ELSEIF or ELSEIFNOT
<indented code>	
ELSE	- Optional final ELSE
<indented code>	

14.4.14_Example_WRD_SPIN2_Differences

14.5) PRI Block

```
PRI methodname({parameter{,...}}) {: result{,...}} { | {ALIGNW/ALIGNL} {BYTE/WORD/LONG}  
localvar{[count]}{,...}}
```

The private methods are internal to the object and can not be referenced from outside the object.

result –is the return value

```
PRI privateMethod()
```

```
Code
```

14.6) Dat Block

Memory Declaration:

<Symbol> Alignment <Size> <Data> ' reserved memory

Symbol –optional name for the reserved space

Alignment <Size> –the byte alignment for reserved data byte,word,long (1byte,2byte,4byte)

Data –constant expression or comma separated variable or quoted strings treated as same

PASM Propeller Assembly Machine Code:

<Symbol> <Condition> Instruction <Effects> 'Propeller Assembly Code

Symbol –optional name for the command line

Condition –flag condition C Carry or Z Zero IF_C, IF_NC, IF_Z, IF_NZ

Instruction –assembly language Instruction eg. ADD,MOV,etc

Effects –effects that cause the result to be written when executed WR,WC,WZ

14.6.1) Common Dat Declaration Alignment

Data is declared with alignment and size.(Byte(1),Word(2),Long(4)).

Long	0				1			
Word	0		1		2		3	
Byte	0	1	2	3	4	5	6	7
Data	40	41	53	74	72	69	6E	67
Long	2				3			
Word	4		5		6		7	
Byte	8	9	10	11	12	13	14	15
Data	00	00	C2	FF	F8	24	00	00
		0 PAD			0 PAD		0 PAD	
Long	4				5			
Word	8		9		10		11	
Byte	16	17	18	19	20	21	22	23
Data	11	22	33	44	20	00	00	00
Dat								

	Long	\$44,332,211	32			
	S	t	r	i	n	g
String	53	74	72	69	6E	67
A	41					
75000	124F8					

124F8 Is larger than what a word can hold so upper nibbles lost

Long data type will always be placed with an alignment of 4 bytes from the beginning of memory and Will be padded to maintain convention. Word data type will always be placed with an alignment of 2 bytes from the beginning of memory and will be padded or truncated. Byte data will align to a single byte.

Dat

Byte word \$FFAA ,long \$BB995511

The above example specifies byte aligned data, but a word sized value followed by a long sized value. The result that memory contains consecutive data:

Long	0				1			
Word	0		1		2		3	
Byte	0	1	2	3	4	5	6	7
Data	FF	C2	11	55	99	BB		
Dat								
	byte	word	FFC2	long	BB995511			

This looks to be away to pack data and avoid 0 Padding.

```
DAT
MyData  byte 64,'$AA, 55      'creates data table
MyString byte "Hello World",0 'Zero 0 terminated string
```

Pub GetData |Temp

Temp := MyData[0] 'get first byte of Data Table

Pub GetData | Temp

Tem := BYTE[@MyData][0]

DAT

MyData word 40_000, \$BB50

MyList word long \$FF995544, long 1000 'needs clarification book may be wrong pg332

DAT

MyData Long 640_000, \$BB50

MyList byte long \$FF995544, long 1000 'needs clarification book may be wrong pg237

DAT

Data Pointers

DAT

Str0 BYTE "Monkeys",0 'strings with symbols

Str1 BYTE "Gorillas",0

Str2 BYTE "Chimpanzees",0

Str3 BYTE "Humanzees",0

StrList WORD @Str0 'in Spin2, these are offsets of strings relative to start of object

WORD @Str1 'in Spin2, @@StrList[i] will return address of Str0..Str3 for i = 0..3

WORD @Str2 'in PASM-only programs, these are absolute addresses of strings

WORD @Str3 '(use of WORD supposes offsets/addresses are under 64KB)

14.6.2) Filling Data Tables

symbol data_type fill_value[array_length]

DAT

Custom long -1[C_CHARS]

That is the equivalent of defining C_CHARS (length) longs filling the value of -1 in the DAT block -- it's creating a DAT array with all values initialized to -1. You can address each long individually using Custom[idx] where idx is 0 to 7. If you need the address of the array you can get it with @Custom. If you later need 10 longs, you only have to change the definition of C_CHARS.

Declaring Repeating Data (Syntax 1)

Data items may be repeated by using the optional *Count* field. For example:

DAT

MyData byte 64, \$AA[8], 55

The above example declares a byte-aligned, byte-sized data table, called MyData, consisting of the following ten values: 64, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, 55. There were eight occurrences of \$AA due to the [8] in the declaration immediately after it.

14.6.1_Example_WRD_ Data Block Address

```
{{14.6.1_Example_WRD_ Data Block Address}}
```

```
{{
```

```
Self Modifying Code and Pointer to Tables
```

```
ALTS D,{#}S
```

```
Alter S field of next instruction to (D + S) & $1FF. D += sign-extended S[17:9].
```

```
MOV D,{#}S {WC/WZ/WCZ}
```

```
Move S into D. D = S. C = S[31]. *
```

```
}}
```

```
CON
```

```
_clkfreq = 200_000_000 "Debug clock must be greater than 10MHZ
```

```
P0 = 0 , P1 = 1 , P2 = 2
```

```
VAR
```

```
Byte cogRunning 'cog ID started is returned or -1 if not started
```

```
PUB main()
```

```
  debug("-----")
```

```
  debug("Self Modifying Code")
```

```
  debug("Example Modifies MOV Destination Field")
```

```
  debug("Allows Pointer to increment through Table")
```

```
  debug("To Display TABLE Values In single variable Value")
```

```
  debug("-----")
```

```
  cogRunning := COGINIT(COGEXEC_NEW,@S0_Dat,0)
```

```
  debug(udec(cogRunning))
```

```
  repeat 'keep cog 0 running
```

```
DAT
```

```
  ORG 0
```

```
S0_Dat    DRVH  #P0          'P0 on program running
```

```
  debug("-----")
```

```
  MOV  Pointer,#valTable 'set Pointer to first address in valTable
```

```
_Next    ALTS Pointer,Index
```

```
  MOV  Value,#0          'place holder #0 value comes from ALTS D,S = Pointer +Index
```

```
  debug(udec(Pointer),udec(Index),udec(Value))
```

```
  debug("-----")
```

```
  ADD  Index,#1
```

```
  CMP  Index,#4 WZ
```

```
  IF_Z  JMP  #_Loop1
```

```
  JMP  #_Next
```

```
_Loop1    NOP
```

```
  JMP  #_Loop1          'remember # immediate
```

```
Index      long  0
```

```
Pointer     long  0
```

```
Value       long  0
```

```
valTable    long  0,1,2,3
```

15.0) Operators

15.1) Pre and Post Operators

Var-Prefix Operators	Term (method only)	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Exp
++ (pre)	++var	1	++var	1	Pre-increment	
-- (pre)	--var	1	--var	1	Pre-decrement	
?? (pre)	??var	1	??var	1	Iterate long per XOR032, return pseudo-random	
Var-Postfix Operators	Term (method only)	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Exp
(post) ++	var++	1	var++	1	Post-increment	
(post) --	var--	1	var--	1	Post-decrement	
(post) !!	var!!	1	var!!	1	Post-logical NOT (0 → -1, non-0 → 0)	
(post) !	var!	1	var!	1	Post-bitwise NOT	
(post) \	var\x	1	var\x	1	Post-assign x	
(post) ~	var~	1	var~	1	Post-clear all bits	
(post) ~~	var~~	1	var~~	1	Post-set all bits	

15.2) Operator ?? PsedoRandomNumberGenerator

PRNG of XOR032 requires variable to be non zero initially "Xorshift RNGs" by George Marsaglia describes a very efficient system for generating high-quality random numbers using very little compute and storage.

- <https://forums.parallax.com/discussion/168188/xoroshiro-random-number-generator/p1>

Here is the xoroshiro++ pseudo code:

```

;{s1,s0} = state (input and output)
;prn = pseudo-random number (output)
;tmp and prn can be the same register

;xoroshiro+
    xor s1,s0          ;s1 = s1 ^ s0
    mov tmp,s1
    rol s0,a           ;s0 = s0 rol a
    shl tmp,b          ;tmp = (s1 ^ s0) shl b
    xor s0,tmp
    xor s0,s1           ;s0 = s0 rol a ^ (s1 ^ s0) shl b ^ s1 ^ s0
    rol s1,c           ;s1 = (s1 ^ s0) rol c
    mov prn,s0
    add prn,s1          ;prn = s0 + s1
;xoroshiro++ enhancement

```

15.1_Example_WRD_Pre_and_Post_Operators

15.2_Example_Address_Operators

Address Operators	Term (method only)	Priority (term)			Description	Float Exp
@	@symbol	1			Hub address of VAR/PUB/PRI variable or DAT symbol	
@	@method	1			Pointer to method, may be @object[[i]].method	
@@	@@x	1			Hub address of object + x, 'DAT x long @dat_symbol'	
#	#reg_symbol	1			Register address of cog/LUT DAT symbol	

15.3_Example_WRD_Bitwise_Decod_and_ENCODE

Note: |< Decode 0-31 Does not exist In P2 works for P1

DECOD create a binary bit set in accordance with bit number 0-31 BinPattern = **DECOD** PinNum

PinNum = 31 BinPattern %10000000_00000000_00000000_00000000

ENCOD create a Decimal number of the highest bit set in a given number PinNum = **ENCOD** BinPattern

BinPattern = %11111111_11111111_11111111_11111111 PinNum = 31

15.4_Example_WRD_##_Operator

The 9 bit data field can be augmented with ## using AUGS or AUGD which is transparent compiler
Expands the ## operator .

```

{{
  15.4_Example_WRD_##_Operation
  Pr0 $1D8 Pr1 $1D9 Pr2 $1DA Pr3 $1DB Pr4 $1DC Pr5 $1DD Pr6 $1DE Pr7 $1DF
}}
CON
  _clkfreq = 200_000_000          ' system freq as a constant
  NumCon01 = $1DA
  NumCon02 = $1DB
PUB main()
  org
  mov Pr0,##NumCon01+NumCon02    'constant NumCon01 + NumCon02 is added imediate load to
Pr0
  mov Pr1,##(NumCon01+NumCon02)  '#imediate stops a register reference load
  mov Pr2,$1DA
  mov Pr3,##Pr2
  end
  debug(udec(Pr0),udec(Pr1),uhex(Pr2),uhex(Pr3))

```

16.0) Method Pointer

Method pointers are LONG values which point to a method and are then used to call that method indirectly. To establish a method pointer, you can assign a long variable using "@" before the method name.

Note that there are no parentheses after the method name:

LongVar := @SomeMethod 'a method within the current object

LongVar := @SomeObject.SomeMethod 'a method within a child object

LongVar := @SomeObject[index].SomeMethod 'a method within an indexed child object

Method pointers can be generated on-the-fly and passed as parameters:

SetUpIO(@InMethod,@OutMethod)

Method pointers are then used in the following ways to call methods:

LongVar() 'no parameters and no return values

LongVar(Par1, Par2) 'two parameters and no return values

Var := LongVar():1 'no parameters and one return value

Var1,Var2 := LongVar(Par1):2 'one parameters and two return values

Var1,Var2 := POLXY(LongVar(Par1,Par2,Par3):2) 'three parameters and two return values

There is **no compile-time awareness** of how many parameters the method pointed to actually has. You need to code your method pointer usage such that **you supply the proper number of parameters** and specify the **proper number of return values after a ":"**, so that there is agreement with the method pointed to.

Method pointers can be passed through object hierarchies to enable direct calling of any method from anywhere. They can also be used to dynamically point to different methods which have the same numbers of parameters and return values.

How Method Pointers Work

An @method expression generates a 32-bit value which has two bit fields:

[31..20] = Index of the method, relative to the method's object base. The index of the first method will be twice the number of objects instantiated

[19..0] = Address of the method's VAR base. The method's VAR base, in turn, contains the address of the method's object base.

By putting the method's index and VAR base address together into the 32-bit value, and having the VAR base contain the method's object base address, a complete method pointer is established in a single long, which can be treated as any other variable.

To accommodate method pointers, **each object instance reserves the first long of its VAR space for the object base address**. When an @method expression executes, that first long is written with the object's base address.

1.6.0.1_Example_WRD_METHOD_POINTER

```

{{16.0.1_Example_WRD_METHOD_POINTER}}
{{
LongVar := @SomeMethod           'a method within the current object
LongVar := @SomeObject.SomeMethod 'a method within a child object
LongVar := @SomeObject[index].SomeMethod 'a method within an indexed child object
}}
con
    _clkfreq = 200_000_000
var
long testMethodPointer
pub main() | byte Pin, long debugMsg1, long debugMsg2
    Pin := 0
    testMethodPointer := @OutMethodBlink
    debugMsg1, debugMsg2 := testMethodPointer(Pin):2 'must declare number of return variables
    debug(zstr(debugMsg1), zstr(debugMsg2))          'send messages
    repeat                                           'keep cog0 running
pub OutMethodBlink(Ppin): result1, result2
    pinhigh(Ppin)
    result1 := @msgON
    result2 := @msgReturn
    waitms(500)
dat
msgON      byte    "Pin is ON high ", 0           'zero terminated string
msgReturn  byte    "OutMethodBlink Executed ", 0   'zero terminated string

```

16.0.2_Example_WRD_METHOD_POINTER_Object

```

{{16.0.2_Example_WRD_METHOD_POINTER_Object}}
{{
LongVar := @SomeMethod 'a method within the current object
LongVar := @SomeObject.SomeMethod 'a method within a child object
LongVar := @SomeObject[index].SomeMethod 'a method within an indexed child object
}}
con
    _clkfreq = 200_000_000
obj
    methodPointer : "16.0.1_Example_WRD_METHOD_POINTER" 'define object to include
var
    long testMethodPointerInObject
pub main() | byte Pin, long debugMsg1, long debugMsg2
    Pin := 0 'P0 led 1k
    testMethodPointerInObject := @methodPointer.OutMethodBlink 'indirect object method pointer
    debugMsg1, debugMsg2 := testMethodPointerInObject(Pin):2 'declare number of return variables
    debug(zstr(debugMsg1), zstr(debugMsg2)) 'send messages
    repeat

```

16.0.3_Example_WRD_METHOD_POINTER_Object_Array

```

{{16.0.3_Example_WRD_METHOD_POINTER_Object_Array}}
{{
LongVar := @SomeMethod 'a method within the current object
LongVar := @SomeObject.SomeMethod 'a method within a child object
LongVar := @SomeObject[index].SomeMethod 'a method within an indexed child object
}}
con
    _clkfreq = 200_000_000
obj
    methodPointer[4] : "16.0.1_Example_WRD_METHOD_POINTER"    'define object to include
var
    long testMethodPointerInObject
pub main() | byte Pin, long debugMsg1, long debugMsg2
    Pin := 0
        'P0 led 1k
    testMethodPointerInObject := @methodPointer[0].OutMethodBlink    'indirect object method pointer
    debugMsg1, debugMsg2 := testMethodPointerInObject(Pin):2          'declare number of return variables
    debug(udec(Pin), zstr(debugMsg1), zstr(debugMsg2))                'send messages
    Pin := 1
        'P1 led 1k
    testMethodPointerInObject := @methodPointer[1].OutMethodBlink    'indirect object method pointer
    debugMsg1, debugMsg2 := testMethodPointerInObject(Pin):2          'declare number of return variables
    debug(udec(Pin), zstr(debugMsg1), zstr(debugMsg2))                'send messages
    Pin := 2
        'P2 led 1k
    testMethodPointerInObject := @methodPointer[2].OutMethodBlink    'indirect object method pointer
    debugMsg1, debugMsg2 := testMethodPointerInObject(Pin):2          'declare number of return variables
    debug(udec(Pin), zstr(debugMsg1), zstr(debugMsg2))                'send messages
    repeat

```


16.1) SEND

SEND is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. Its purpose is to provide an efficient output mechanism for data.

SEND can be assigned like a method pointer, but it must point to a method which takes one parameter and has no return values:

```
SEND := @OutMethod
```

When used as a method, SEND will pass all parameters, including any return values from called methods, to the method SEND points to:

```
SEND("Hello! ", GetDigit()+"0", 13)
```

Any methods called within the SEND parameters will inherit the SEND pointer, so that they can do SEND methods, too:

```
PUB Go()
```

```
    SEND := @SetLED
```

```
    REPEAT
```

```
        SEND(Flash(),$01,$02,$04,$08,$10,$20,$40,$80)
```

```
PRI Flash() : x
```

```
    REPEAT 2
```

```
        SEND($00,$FF,$00)
```

```
    RETURN $AA
```

```
PRI SetLED(x)
```

```
    PINWRITE(56 ADDPINS 7, !x)
```

```
    WAITMS(125)
```

In the above example, the following values are output in repeating sequence: \$00, \$FF, \$00, \$00, \$FF, \$00, \$AA, \$01, \$02, \$04, \$08, \$10, \$20, \$40, \$80 (but inverted for LEDs)

Though a called method inherits the current SEND pointer, it may change it for its own purposes. Upon return from that method, the SEND pointer will be back to what it was before the method was called. So, the SEND pointer value is propagated in method calls, but not in method returns.

SEND acts as a special type of method pointer, inherited from the calling method and, in turn, conveyed to all called methods. It provides an efficient output mechanism for data.

You may assign SEND as you would any method pointer, but it must point to a method that 1. *takes one parameter* and 2. *has no return values*:

```
SEND := @OutMethod 'SEND points to OutMethod
```

SEND will pass all parameters, including any return values from called methods, to the method SEND points to:

```
SEND("Hello! ", GetDigit()+"0", 13)
```

Any methods called from within the SEND parameters, such as GetDigit() in the example above, will inherit the SEND pointer, so that they also may use the SEND method. The following code provides an example of SEND use. It sends 8-bit patterns of 0s and 1s to LEDs at pins P56 through P63:

```
PUB go()
  SEND := @SetLED
  REPEAT
    SEND($01, $02, $04, $08, $10, $20, $40, $80)

PRI SetLED(x)
  PINWRITE(56 ADDPINS 7, !x)
  WAITMS(125)
```

Note: LEDs on the P2 EVAL Board are driven by active-low signals.

Within the go() method, the statement SEND := @SetLED, gives SEND the pointer to the SetLED method. This method satisfies the requirement: only one variable and no return value. Note the SetLED method above includes a short delay so the LED patterns remain visible long enough so you can see them.

Next the REPEAT loop executes the SEND(\$01, \$02...) statement that transfers the first parameter \$01 to

the SetLED method. The LED at pin P56 turns on. When this method finishes, it returns control to the SEND statement, which then sends \$02 to the SetLED method, which turns on the LED at P57. Each LED turns on and off in sequence again and again in the REPEAT loop.

A second example shows how other methods can inherit the SEND pointer. An added method, Flash(), will turn all LEDs on and off. This method includes a SEND statement, too.

PUB go()

SEND := @SetLED

REPEAT

SEND(Flash(), \$01, \$02, \$04, \$08, \$10, \$20, \$40, \$80)

PRI Flash() : x

REPEAT 2

SEND(\$00,\$FF,\$00)

RETURN \$AA

PRI SetLED(x)

PINWRITE(56 ADDPINS 7, !x)

WAITMS(125)

The program will call the Flash() method (in the first SEND() parameter) and will eventually pass the return value from Flash() to the SetLED() method (after Flash() has fully executed).

First, the Flash() method will run and send its own values, \$00, \$FF, \$00, to the LEDs two times. Then, if you watch the LEDs, they next display \$AA next. Why?

The Flash() method returns the value \$AA to the SEND statement: SEND(Flash()),\$01,\$02... In effect the \$AA value gets inserted in place of the call to Flash() in the list of parameters, making the whole program execution behave as if the SEND(Flash(), \$01, \$02...) had really been:

SEND(\$00,\$FF,\$00)

SEND(\$00,\$FF,\$00)

SEND(\$AA, \$01, \$02...)

The Flash() method inherited the SetLED() address and can use it independent of other uses in this program

16.1_Example_WRD_Send_Led

```
{{16.1_Example_WRD_Send_Led}}
```

```
{{
```

SEND can be assigned like a method pointer, but it must point to a method which takes one parameter and has no return values:

```
SEND := @OutMethod          'method can be either PUB or PRI
SEND(anotherMethod(),parameter)  'anotherMethod single return value is sent
```

Any methods called within the SEND parameters will inherit the SEND pointer, so that they can do SEND methods, too:

```
Pub anotherMethod() :result
  send(parameter)          'SEND method Pointer @OutMethod is inherited and can be called by
  anotherMethod
  result := $FF
```

```
Pub OutMethod
  "code to go here"
}}
```

```
con
```

```
_clkfreq = 200_000_000
```

```
PUB Go()
  SEND := @SetLED
  REPEAT
    SEND(Flash(),$0F,Flash(),$33,$DD,$33,$DD)
PRI Flash() : y
  y := $F0
  waitms(500)
PRI SetLED(x)
  PINWRITE(0 ADDPINS 7, x)
  WAITMS(500)
```

16.2) RECV

RECV, like SEND, is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. Its purpose is to provide an efficient input mechanism for data.

RECV can be assigned like a method pointer, but it must point to a method which takes no parameters and returns a single value:

```
RECV := @InMethod
```

An example of using RECV:

```
VAR i
PUB Go()
  RECV := @GetPattern
  REPEAT
    PINWRITE(56 ADDPINS 7, !RECV())
    WAITMS(125)
PRI GetPattern() : Pattern
  RETURN DECOD(i++ & 7)
```

In the above example, the following values are output in repeating sequence: \$01, \$02, \$04, \$08, \$10, \$20, \$40, \$80 (but inverted for LEDs)

Though a called method inherits the current RECV pointer, it may change it for its own purposes. Upon return from that method, the RECV pointer will be back to what it was before the method was called. So, the RECV pointer value is propagated in method calls, but not in method returns.

16.2_Example_WRD_Receive_Led

```

{{16.2_Example_WRD_Receive_Led}}
{{
RECV := @InMethod
DECOD create a binary bit set in accordance with bit number 0-31    BinPattern = DECOD PinNum
PinNum =31 BinPattern %10000000_00000000_00000000_00000000
}}
VAR i
PUB Go()
  RECV := @GetPattern
  REPEAT
    i := 0
    REPEAT 7
      PINWRITE(0 ADDPINS 7, RECV())
      WAITMS(250)
    i := 0
    REPEAT 7
      PINWRITE(0 ADDPINS 7,!RECV())
      WAITMS(250)
PRI GetPattern() : Pattern
  if i > 7
    i := 0
  RETURN DECOD(i++ & 7)

```

17.0) PASM Propeller Assembly Language

(Need more information some supposition)

Typical PASM program consists of Spin code to boot propeller and DAT section consisting of assembly code to be loaded into a Cog. The Propeller II allows also Inline assembly language not available with Propeller I. Cog 0 is launched with the spin byte interpreter.

Each Cog has **PC (Program Counter)** that is incremented with a Common System Clock
PC points to the next instruction to be Executed.

Each Cog Has a Instruction **Result Register or Flag Register** with two bits zero Z and carry C flags
Result or Flag Register can be thought as to why cogs are Risc processors (reduced instruction set computer)

17.0.0_Example_PASM_Template(used for testing PASM commands)

17.0.0.1_Example_WRD_PASM_COG_DAT_Launch_Template

The following program template launches Cog 1 to run “codePASM”:

```

{{17.0.0.1_Example_WRD_PASM_COG_DAT_Launch_Template}}
CON
  _clkfreq = 200_000_000    'Debug must be enabled clock must be greater than 10MHZ for Debug
VAR
  Byte cogRunning 'cog ID started is returned or -1 if not started
PUB main()
  cogRunning := COGINIT(COGEXEC_NEW,@codePASM,$FFF_FFFF)
  debug(udec(cogRunning))
  repeat 'keep cog 0 running
  'Start next available cog which is 1 load cog 1 memory with @codePASM at Cog Memory $10
  'PTRA register will be loaded with $FFF_FFFF
DAT  ORG 0 'COGINIT(COGEXEC_NEW,@codePASM,PTRAvale)
'Normally start at $000 but you don't have too
codePASM
  MOV  DIRA, #$FF    'Set the direction of the first 8 pins to Output
  GETCT cogCounterValue 'Get global system counter value
  ADDCT1 cogCounterValue,PTRA 'set CT1 event to trigger on CT = countvalue + PTRa
  _Loop WAITCT1
  ADDCT1 cogCounterValue,PTRA
  XOR  OUTA, #1
  NOP
  debug(ubin(OUTA))    'send status to Debug Window
  JMP  #_Loop    'don't forget #immediate setting or register is loaded
cogCounterValue Long 0 'counter value CT storage

```

Both data and PASM may be intermixed in the DAT section . The “COGINIT” command loads 496 consecutive long values starting from the aspecified address “ORG 0” The PASM starts to run from the specified address.

Note:

- 1) (TBD) The `:_Loop` instruction symbol name is forgotten by the compiler after compiling the `JMP #:_Loop` thus allowing `_Loop` to be reused in subsequent assembly code symbol labeling. (Needs Verification)
- 2) `PTRA` register can contain an address allowing a reference for acquiring other variables in the HUB in the above example a value is passed.
- 3) `ORG 0` directive causes compiled code to be loaded starting at 0 and initiating PC to start from 0 this could be changed for example to `ORG 10` thus the first registers 0-9 long in length would be available for program data storage.
- 4) `#` stands for immediate if not included the instruction is register reference and loads from register not value.

17.0.0.2_Example_WRD_Inline_PASM

```

{{
17.0.0.2_Example_WRD_PASM_COG_DAT_Launch_Template
}}
CON
_clkfreq = 200_000_000 "Debug must be enabled clock must be greater than 10MHZ for Debug
PUB main()
    PTRA := $FFF_FFFF 'spin assignment instruction
    ORG
codePASM
    MOV  DIRA, #$FF      'Set the direction of the first 8 pins to Output
    GETCT cogCounterValue 'Get global system counter value
    ADDCT1 cogCounterValue,PTRA 'set CT1 event to trigger on CT == countvalue + PTRA
._Loop WAITCT1          'wait till CT == cogCounterValue
    ADDCT1 cogCounterValue,PTRA
    XOR  OUTA, #1
    NOP
    debug(ubin(OUTA)) 'send status to Debug Window
    JMP  #._Loop      'JMP to Loop note the dot . if included name reference is forgotten by
compiler
cogCounterValue Long 0 'counter value CT storage

END

```


17.0.1) Assembly Language Syntax

Each assembly instruction has common syntax elements consisting of an optional label, optional condition, the instruction and optional effects:

<Label> <Condition> Instruction <Effects>

- **Label**- is an optional statement. Label can be global (starting with an underscore “_” or letter) or can be local (starting with a colon “:”) Local labels must be separated from other same named local labels by at least one global label. Label is used by instructions like JMP, CALL and COGINIT to designate the target destination.
- **Condition**- is an optional execution condition (IF_C, IF_Z etc) that causes an instruction to be executed or not. There are 32 possible condition checks.
- **Instruction**- is a Propeller Assembly Instruction (MOV, ADD, COGINIT etc) and its operands. There are 409 Instructions with up to 2 operands per instruction.
- **Effects**- is an optional list of one to three execution effects (WZ, WC, WR and NR). They cause the instruction to modify the Z flag, C flag and write or not write the instruction result value to the destination register.

#S-Immediate value to be used S-register contains value to be used S represent the source operand

#D-Immediate value to be used D-register is the result to be written D represents the destination operand

* Z = (result == 0).

** If #S and cogex, PC += signed(S). If #S and hubex, PC += signed(S*4). If S, PC = register S. (needs clarification)

The Instructions are listed without Labels or Conditions as an example ROR:

<Label> <Condition> ROR D, #S {WC, WZ, WCZ}

Rotate Right D = [31:0] of ({D[31:0], D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0].

Condition	Instruction	Effects(Flags)	Destination	Source
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00				
E E E E	0 0 0 0 0 0 0	C Z I	D D D D D D D D	S S S S S S S S
4 bit = 15	7 bit = 127	3 bit = 7	9 bit = 511 max address	9 bit = 511 max address

Key	Description
EEEE	Conditional test (see "Instruction Prefix" list at bottom of the instruction set spreadsheet)
C	0: Do not update the "C" register 1: Update the "C" register. In the instruction syntax, this is denoted by "WC" or "WCZ".
Z	0: Do not update the "Z" register 1: Update the "Z" register. In the instruction syntax, this is denoted by "WZ" or "WCZ".
I	0: Source field is a register address 1: Source field is a literal value. In the instruction syntax, this is denoted by the "#" character.
L	0: Destination field is a register address 1: Destination field is a literal value. In the instruction syntax, this is denoted by the "#" character.
DDDDDDDD D	Destination field
SSSSSSSS	Source field
N,NN,NNN	Index number. This is only used for instructions with a third index argument.
cccc	conditional test used to update C (%0000=clear, %1111=set, all others per EEEE)
zzzz	conditional test used to update Z (%0000=clear, %1111=set, all others per EEEE)

NOTE: Some instruction not using WC or WZ can use CZI field for different meanings such as:
CallPA #D #S → CZI = 1LI = 111 for #D #S (L=D I=S)

17.0.1.1_Example_WRD_PASM_Instruction_Syntax

This program can be run with `_RET_` and `debug(ubin(S1_ROR))` commented out, and can be run with uncommented to return condition code. When uncommented the `_RET_ ROR Pr0,Pr1` executes and closes the cog but debug window will return the Instruction code with condition code EEEE.

Note:

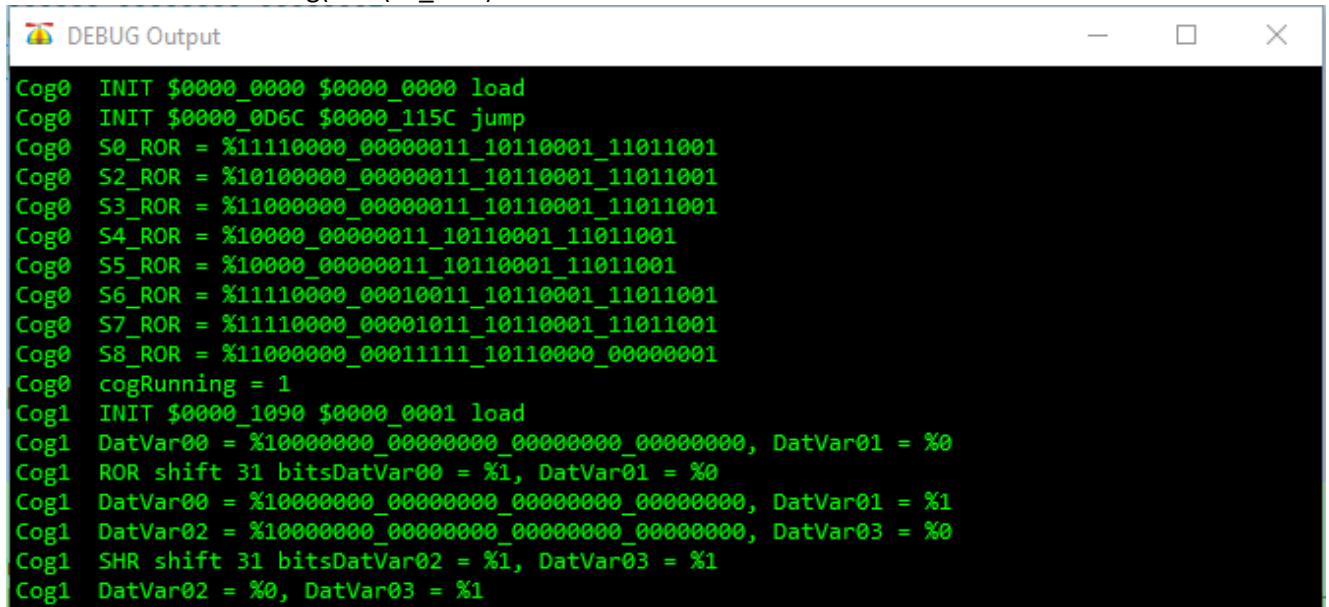
ROR rotate right shifts bit B0 back to B31. SHR shift right loses B0

Debug can be used to grab the Encoded instruction to determine condition expression

S[4:0] contains number of bits to shift maximum is 31 for 32 bit long word

Note:

'`S1_ROR _RET_ ROR Pr0,Pr1`' Execute <inst> always and return if no branch. If not branching pop stack this line and `debug(ubin(S1_ROR))` are commented out



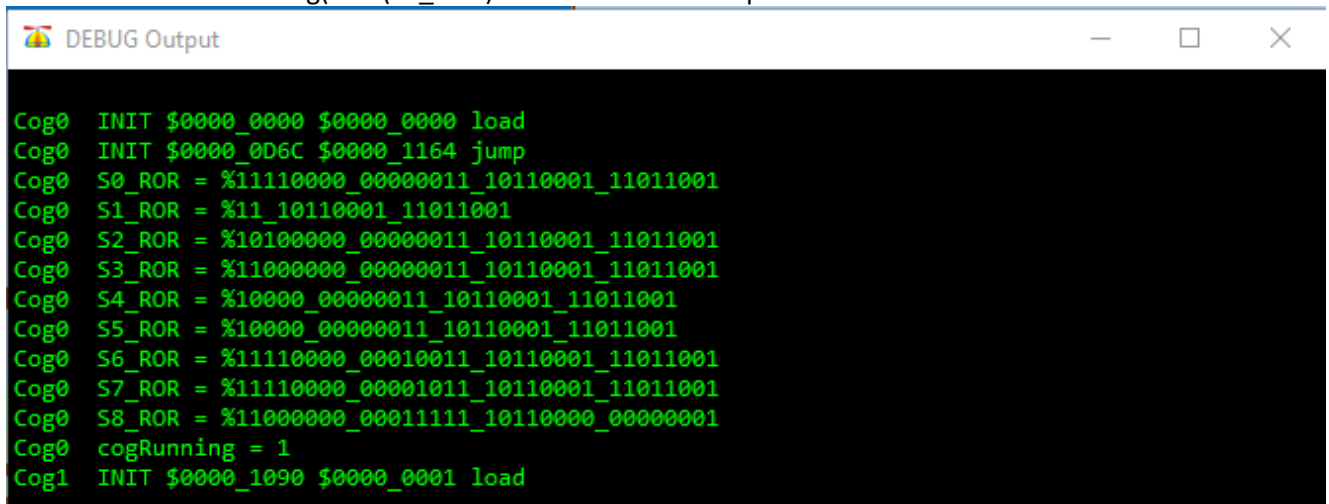
```

Cog0 INIT $0000_0000 $0000_0000 load
Cog0 INIT $0000_0D6C $0000_115C jump
Cog0 S0_ROR = %11110000_00000011_10110001_11011001
Cog0 S2_ROR = %10100000_00000011_10110001_11011001
Cog0 S3_ROR = %11000000_00000011_10110001_11011001
Cog0 S4_ROR = %10000_00000011_10110001_11011001
Cog0 S5_ROR = %10000_00000011_10110001_11011001
Cog0 S6_ROR = %11110000_00010011_10110001_11011001
Cog0 S7_ROR = %11110000_00001011_10110001_11011001
Cog0 S8_ROR = %11000000_00011111_10110000_00000001
Cog0 cogRunning = 1
Cog1 INIT $0000_1090 $0000_0001 load
Cog1 DatVar00 = %10000000_00000000_00000000_00000000, DatVar01 = %0
Cog1 ROR shift 31 bitsDatVar00 = %1, DatVar01 = %0
Cog1 DatVar00 = %10000000_00000000_00000000_00000000, DatVar01 = %1
Cog1 DatVar02 = %10000000_00000000_00000000_00000000, DatVar03 = %0
Cog1 SHR shift 31 bitsDatVar02 = %1, DatVar03 = %1
Cog1 DatVar02 = %0, DatVar03 = %1

```

Note:

`S1_ROR _RET_ ROR Pr0,Pr1` 'Execute <inst> always and return if no branch. If not branching pop stack this line and `debug(ubin(S1_ROR))` is allowed to be compiled.



```

Cog0 INIT $0000_0000 $0000_0000 load
Cog0 INIT $0000_0D6C $0000_1164 jump
Cog0 S0_ROR = %11110000_00000011_10110001_11011001
Cog0 S1_ROR = %11_10110001_11011001
Cog0 S2_ROR = %10100000_00000011_10110001_11011001
Cog0 S3_ROR = %11000000_00000011_10110001_11011001
Cog0 S4_ROR = %10000_00000011_10110001_11011001
Cog0 S5_ROR = %10000_00000011_10110001_11011001
Cog0 S6_ROR = %11110000_00010011_10110001_11011001
Cog0 S7_ROR = %11110000_00001011_10110001_11011001
Cog0 S8_ROR = %11000000_00011111_10110000_00000001
Cog0 cogRunning = 1
Cog1 INIT $0000_1090 $0000_0001 load

```

PASM Syntax			Encoded Instruction 32 Bit
S0_ROR	ROR Pr0,Pr1		11110000_00000011_10110001_11011001
S1_ROR _RET_	ROR Pr0,Pr1		00000000_00000011_10110001_11011001
S2_ROR IF_Z	ROR Pr0,Pr1		10100000_00000011_10110001_11011001
S3_ROR IF_C	ROR Pr0,Pr1		11000000_00000011_10110001_11011001
S4_ROR IF_NC_AND_NZ	ROR Pr0,Pr1		00010000_00000011_10110001_11011001
S5_ROR IF_NZ_AND_NC	ROR Pro,Pr1		00010000_00000011_10110001_11011001
S6_ROR	ROR Pr0,Pr1 WC		11110000_00010011_10110001_11011001
S7_ROR	ROR Pr0,Pr1 WZ		11110000_00001011_10110001_11011001
S8_ROR IF_C	ROR Pr0,#Shift WCZ		11000000_00011111_10110001_11011001

					CZI		
Symbol	Condition	Mnemonic	Condition	Instruction	Effects	Destination	Source
S0_ROR		ROR Pr0,Pr1	1111	00000000	000	111011000	111011001
S1_ROR	_RET	ROR Pr0,Pr1	0000	00000000	000	111011000	111011001
S2_ROR	IF_Z	ROR Pr0,Pr1	1010	00000000	000	111011000	111011001
S3_ROR	IF_C	ROR Pr0,Pr1	1100	00000000	000	111011000	111011001
S4_ROR	IF_NC_AND_NZ	ROR Pr0,Pr1	0001	00000000	000	111011000	111011001
S5_ROR	IF_NZ_AND_NC	ROR Pr0,Pr1	0001	00000000	000	111011000	111011001
S6_ROR		ROR Pr0,Pr1 WC	1111	00000000	100	111011000	111011001
S7_ROR		ROR Pr0,Pr1 WZ	1111	00000000	010	111011000	111011001
S8_ROR	IF_C	ROR Pr0,#Shift WCZ	1100	00000000	111	111011000	111011001

17.0.0.3_Example_WRD_ALT_R_D_S

ALTR D,{#}SAlter result register address (normally D field) of next instruction to $(D + S) \& \$1FF$.

D += sign-extended S[17:9].

ALTR D,{#}S	Alter result register address (normally D field) of next instruction to $(D + S) \& \$1FF$. D += sign-extended S[17:9].
ALTR D	Alter result register address (normally D field) of next instruction to D[8:0].
ALTD D,{#}S	Alter D field of next instruction to $(D + S) \& \$1FF$. D += sign-extended S[17:9].
ALTD D	Alter D field of next instruction to D[8:0].
ALTS D,{#}S	Alter S field of next instruction to $(D + S) \& \$1FF$. D += sign-extended S[17:9].
ALTS D	Alter S field of next instruction to D[8:0].
ALTB D,{#}S	Alter D field of next instruction to $(D[13:5] + S) \& \$1FF$. D += sign-extended S[17:9].

ALTR Alter R Field result register address (normally D field) of next instruction to $(D + S) \& \$1FF$.

D += sign-extended S[17:9]. D= IndexD[8:0] + sign-Extended [17:9]

ALTD Alter D field of next instruction to $(D + S) \& \$1FF$.

D += sign-extended S[17:9]. D= IndexD[8:0] + sign-Extended [17:9]

ALTS Alter S field of next instruction to $(D + S) \& \$1FF$.

D += sign-extended S[17:9]. D= IndexD[8:0] + sign-Extended [17:9]

RDSS= Offset S[17:9] + BaseAddressS[8:0]

D = IndexD[8:0] + sign-Extended [17:9]

ALTR D,{#}S

Alter result register address (normally D field) of next instruction to $(D + S) \& \$1FF$. $D += \text{sign-extended } S[17:9]$.

17.0.0.3_Example_WRD_ALTR D,R,{#}S_116}}

ALTR D,{#}S

Alter result register address (normally D field) of next instruction to $(D + S) \& \$1FF$.

$D += \text{sign-extended } S[17:9]$.

By Means of an example we want the result of XOR X,Y but you don't want to destroy register X.

By using the ALTR instruction you can avoid a bunch of move statements.

Also some registers cannot be written too. Using the ALTR instruction you can use the assembly instructions without destroying either register and writing the instruction operation to an alternate register.

$\text{ResultAddress} = \text{BaseAddressS}[8:0] + \text{OffsetD}[8:0] + \text{IndexD}[8:0]$

$S = \text{Offset } S[17:9] + \text{BaseAddressS}[8:0]$

$D = \text{IndexD}[8:0] + \text{sign-Extended } [17:9]$

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \wedge S$. C = parity of result. *

Example

Write the result of XOR Ax,Bx to 'xorResult' not affecting Ax or Bx use Offset and Index

17.1) NOP No operation

NOP instruction does not effect any flags but requires 2 cycles can be used for timming or a filler when programming.

17.1.1_Example_WRD_NOP_001

```
{{17.1_Example_WRD_NOP_001}}
```

```
{{
```

```
NOP instruction does not effect any flags
```

```
requires 2 cycles can be used for timming or a filler when programming.
```

```
}}
```

```
CON
```

17.2) ROR Rotate Right and ROL Rotate Left

ROL D,{#}S {WC/WZ/WCZ} Rotate left D. Note: B31 will rotate back to B0

D = [63:32] of ({D[31:0], D[31:0]} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *

ROR D,{#}S {WC/WZ/WCZ} Rotate right. Note: bit B0 will rotate back to B31

D = [32:63] of ({D[31:0], D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

Interperating the above command the destination register D = [32:63] It's an ordered bitwise assignment from D on leftside of "of", as a 64-bit word, to D on rightside leftside of "of", as a 32-bit word, with S specifying the alignment shift. This is analogy of functional description. In reality, the variability of S means the logic gates needed to achieve that, in what's called a "barrel-shifter", is a pretty large structure.

Imagine you have taken the register to be shifted and placed a copy of that beside it so now you have 64 bits, with the top half the same as the bottom half. You then use the S register value to shift the whole thing right, and you then take the top half, which gives the same effect as a circular shift of 32 bits. It should be noted that S can hold 0, which results in no shift, but can still set the flags which would put D[0] into the C flag for Rotate Right.

While a barrel shifter is fairly large compared to a simple shifter, it gives a speed advantage that scales with register size (a 31 bit shift takes the same time as a single bit shift, giving a x31 speed up)

ROR rotate right operand and ROL rotate left operand operate in the same manner shifting right or shifting left. The following description is for ROL rotate left.

Starting with 17.2.2_Example_WRD_ROL_0000001.

valPr0	long	%10101010_10101010_10101010_10101010	'cog 1 Dest
valPr1	long	%00000000_00000000_00000000_00000001	'cog 1 Src

D is register Dest, the hardware latches two copies of it to give 64 bits =

10101010_It then shifts bits left by the value in S the 64 bits =

01010101_01010101_01010101_01010101__01010101_01010101_01010101_0101010x

The result is then taken as bits 63 to 32 of this (bolded above) giving D = 01

For a double shift this example value is awkward as it appears to make no change. To illustrate it better the following shows a variety of bit shifts with different a starting value

Dest = DEADBEEF = 11011110_10101101_10111110_11101111

64 bits in the shifter (preshift) is

11011110_10101101_10111110_11101111__11011110_10101101_10111110_11101111

A single shift gives

10111101_01011011_01111101_11011111__10111101_01011011_01111101_1101111x

A double shift gives

01111010_10110110_11111011_10111111__01111010_10110110_11111011_101111xx

A 20 bit shift gives

11101110_11111101_11101010_11011011__11101110_1111xxxx_xxxxxxxxx_xxxxxxxxx

A 31 bit shift gives 11101111_01010110_11011111_01110111__1xxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx

Please note that the bits at the right hand end are x (don't care) values, because it is not known what the chip sets these to, and it doesn't matter because they never leave the shifter. In fact, bits 31:0 out of the shifter may not be implemented in the silicon to save gates, power, and heat.

All of these shifts take the same amount of time to occur, as they are all performed in parallel with only the correct one being passed to the result register.

17.2.1 Example_WRD_ROR_002

ROR D,{#}S {WC/WZ/WCZ} Rotate right. Note: bit B0 will rotate back to B31

D = [31:0] of ({D[31:0], D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

ROR will shift bits to the right and B0 gets shifted to B31 note that leading 0's is not shown by debug

17.2.2 Example_WRD_ROL_003

ROL D,{#}S {WC/WZ/WCZ} Rotate left.

D = [63:32] (Not sure ?) of ({D[31:0], D[31:0]} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *

ROL will shift bits to the Left and B31 gets shifted to B0 note that leading 0 is not shown by debug

17.3) SHR Shift Right and SHL Shift Left

SHR D,{#}S{WC\WZ\WCZ} Shift Right

D = [31:0] of ({32'b0, D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

SHL D,{#}S{WC\WZ\WCZ} Shift Left

Si31:0] > Analogy of 32 bit shift register with 0 being shifted in depending if SHR/SHL

Pr0

01010101_01010101_01010101_01010101

SHR Pr0,#1 WC

00101010_10101010_10101010_10101010 C = 1

Pr0

01010101_01010101_01010101_01010101

SHR Pr0,#2 WC

00010101_01010101_01010101_01010101 C = 0

Pr0
01010101_01010101_01010101_01010101
SHL Pr0,#1 WC
C=0 10101010_10101010_10101010_10101010

Pr0
01010101_01010101_01010101_01010101
SHL Pr0,#2 WC
C = 1 01010101_01010101_01010101_01010100

17.3.1_Example_WRD_SHR_004

SHR D,{#}S{WC\WZ\WCZ} Shift Right
D = [32:63] of ({32'b0, D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
SHR will shift bits to the right 0 fills Bits shifted}}

17.3.2_Example_WRD_SHL_005

SHL D,{#}S{WC\WZ\WCZ} Shift Left
D = [32:63] of ({32'b0, D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
SHR will shift bits to the Left 0 fills Bits shifted}}

17.4) RCR Rotate Carry Right and RCL Rotate Carry Left

RCR D,{#}S {WC/WZ/WCZ} Rotate Carry Right.
. D = [31:0] of ({32{C}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

RCL D,{#}S {WC/WZ/WCZ} Rotate Carry Left
D = [63:32] of ({D[31:0], {32{C}}} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *

RR performs a rotate right of D destination field ,bit times as {#}S using the C flags original value each of th MSB's affected. If WC effect is applied C will be changed to match B0 of D destination original value. D has bits specified by {#}S of C rotated right into D. **R** performs a rotate Left of D destination field ,bit times as {#}S using the C flags original value each of th LSB's affected. If WC effect is applied C will be changed to match B31 of D destination original value. D has bits specified by {#}S of C rotated right into D.

17.4.1_Example_WRD_RCR_006

RCR D,{#}S {WC/WZ/WCZ} Rotate Carry Right.
"D = [31:0] of ({32{C}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

17.4.2_Example_WRD_RCL_007

RCL D,{#}S {WC/WZ/WCZ} Rotate Carry Left
'D = [63:32] of ({D[31:0], {32{C}}} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *

17.5) SAR Shift Arithmetic Right and Shift Arithmetic Left

Shift Arithmetic Right can thought as dividing or Shift Arithmetic Left Multiplying the value.
It maintains the sign value.

17.5.1) SAR Shigt Aritmetic Right

SAR D,{#}S {WC/WZ/WCZ}

D = [31:0] of ({32{D[31]}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

Shift Arithmetic Right is a division of a signed binary number divided by 2
the sign bit remains unchanged

Positive Number

00000000_00000000_00000000_00001101 Before Right Shift = 13

Positive Number Shifted Right 0 Loaded Into MSB to Maintain Sign

00000000_00000000_00000000_00000110 After Shift Right = 6 Result is $13/2 = 6$ divide by 2

Negative Number

11111111_11111111_11111111_00011000 Before Shift Right = -232

00000000_00000000_00000000_11100111 2's complement

00000000_00000000_00000000_00000001

00000000_00000000_00000000_11101000 = 232

Negative Number Shifted 1 Loaded Into MSB to Maintain Sign

11111111_11111111_11111111_10001100 = After Shift Right = -116 result is $-232/2$

00000000_00000000_00000000_01110011 2'S complement

00000000_00000000_00000000_00000001

00000000_00000000_00000000_01110100 = 116

17.5.1 Example_WRD_SAR_008

SAR D,{#}S {WC/WZ/WCZ}

Shift arithmetic right. D = [31:0] of ({32{D[31]}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *

17.5.2) Shift Arithmetic Left

SAL D,{#}S {WC/WZ/WCZ}

Shift arithmetic left. D = [63:32] of ({D[31:0], {32{D[0]}}}) << S[4:0]. C = last bit shifted out if S[4:0] > 0, else D[31]. *

Shift Arithmetic Left is a multiplication of a signed binary number by 2

The sign bit remains unchanged

Positive Number

00000000_00000000_00000000_0001101 Before Right Left = 13

Positive Number Shifted Left 0 Loaded into LSB

00000000_00000000_00000000_00011010 After Left Shift = 26 Multiply by 2

Negative Number

11111111_11111111_11111111_00011000 Before Shift Left = -232

00000000_00000000_00000000_11100111 2's complement

00000000_00000000_00000000_00000001

00000000_00000000_00000000_11101000 = 232

Negative Number Shifted Left 0 Loaded into LSB

11111111_11111111_11111110_00110000 = After Shift Left = -464 result is 2 x -232

00000000_00000000_00000001_11001111 2's complement

00000000_00000000_00000000_00000001

00000000_00000000_00000001_11010000 = 464

*17.5.2_Example_WRD_SAL_009***SAL D,{#}S {WC/WZ/WCZ}**

Shift arithmetic left. D = [63:32] of ({D[31:0], {32{D[0]}}}) << S[4:0]. C = last bit shifted out if S[4:0] > 0, else D[31]. *

17.6) ADD Addition

ADD D,{#}S {WC/WZ/WCZ}	Add S into D. $D = D + S$. C = carry of (D + S). *
ADDX D,{#}S {WC/WZ/WCZ}	Add (S + C) into D, extended. $D = D + S + C$. C = carry of (D + S + C). $Z = Z \text{ AND } (\text{result} == 0)$.
ADDS D,{#}S {WC/WZ/WCZ}	Add S into D, signed. $D = D + S$. C = correct sign of (D + S). *
ADD SX D,{#}S {WC/WZ/WCZ}	Add (S + C) into D, signed and extended. $D = D + S + C$. C = correct sign of (D + S + C). $Z = Z \text{ AND } (\text{result} == 0)$.

17.6.1_Example_WRD_ADD_010

ADD D,{#}S {WC/WZ/WCZ}

Add S into D. $D = D + S$. C = carry of (D + S). *

Unsigned Addition

17.6.2_Example_WRD_ADDX_011

ADDX D,{#}S {WC/WZ/WCZ}

Add (S + C) into D, extended. $D = D + S + C$. C = carry of (D + S + C). $Z = Z \text{ AND } (\text{result} == 0)$.

17.6.3_Example_WRD_ADDS_012

ADDS D,{#}S {WC/WZ/WCZ}Add S into D, signed. $D = D + S$.

C = correct sign of (D + S). * Signed Addition

17.6.4_Example_WRD_ADDSX_013

ADD SX D,{#}S {WC/WZ/WCZ}

Add (S + C) into D, signed and extended.

 $D = D + S + C$. C = correct sign of (D + S + C). $Z = Z \text{ AND } (\text{result} == 0)$.

17.7) SUB Subtraction

SUB D,{#}S {WC/WZ/WCZ}	Subtract S from D. $D = D - S$. C = borrow of $(D - S)$. *
SUBX D,{#}S {WC/WZ/WCZ}	Subtract $(S + C)$ from D, extended. $D = D - (S + C)$. C = borrow of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.
SUBS D,{#}S {WC/WZ/WCZ}	Subtract S from D, signed. $D = D - S$. C = correct sign of $(D - S)$. *
SUBSX D,{#}S {WC/WZ/WCZ}	Subtract $(S + C)$ from D, signed and extended. $D = D - (S + C)$. C = correct sign of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.

17.7.1_Example_WRD_SUB_014

SUB D,{#}S {WC/WZ/WCZ}

Subtract S from D. $D = D - S$. C = borrow of $(D - S)$. *

Unsigned Subtraction

17.7.2_Example_WRD_SUBX_015

SUBX D,{#}S {WC/WZ/WCZ}

Subtract $(S + C)$ from D, extended. $D = D - (S + C)$. C = borrow of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.

17.7.3_Example_WRD_SUBS_016

SUBS D,{#}S {WC/WZ/WCZ}

Subtract S from D, signed.

 $D = D - S$. C = correct sign of $(D - S)$. *

Signed subtraction

17.7.4_Example_WRD_SUBSX_017

SUBSX D,{#}S {WC/WZ/WCZ}

Subtract $(S + C)$ from D, signed and extended. $D = D - (S + C)$. C = correct sign of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.

17.7.5_Example_WRD_SUBR_018

SUBR D,{#}S {WC/WZ/WCZ}

Subtract D from S (reverse). $D = S - D$.

C = borrow of $(S - D)$.

17.8) CMP Compare

CMP D,{#}S {WC/WZ/WCZ}	Compare D to S. C = borrow of (D - S). Z = (D == S).
CMPX D,{#}S {WC/WZ/WCZ}	Compare D to (S + C), extended. C = borrow of (D - (S + C)). Z = Z AND (D == S + C).
CMPS D,{#}S {WC/WZ/WCZ}	Compare D to S, signed. C = correct sign of (D - S). Z = (D == S).
CMPSX D,{#}S {WC/WZ/WCZ}	Compare D to (S + C), signed and extended. C = correct sign of (D - (S + C)). Z = Z AND (D == S + C).
CMPR D,{#}S {WC/WZ/WCZ}	Compare S to D (reverse). C = borrow of (S - D). Z = (D == S).
CMPM D,{#}S {WC/WZ/WCZ}	Compare D to S, get MSB of difference into C. C = MSB of (D - S). Z = (D == S).

Compare D register and S register with C/Z conditions entry flags and set flags accordingly.

17.8.1_Example_WRD_CMP_019

CMP D,{#}S {WC/WZ/WCZ}

Compare D to S. C = borrow of (D - S). Z = (D == S).

17.8.2_Example_WRD_CMPX_020

CMPX D,{#}S {WC/WZ/WCZ}

Compare D to (S + C), extended. C = borrow of (D - (S + C)). Z = Z AND (D == S + C).

17.8.3_Example_WRD_CMPS_021

CMPS D,{#}S {WC/WZ/WCZ}

Compare D to S, signed. C = correct sign of (D - S). Z = (D == S)=1 if equal.

17.8.4_Example_WRD_CMPSX_022

CMPSX D,{#}S {WC/WZ/WCZ}

Compare D to (S + C), signed and extended. C = correct sign of (D - (S + C)). Z = Z AND (D == S + C).

17.8.5_Example_WRD_CMPR_023

CMPR D,{#}S {WC/WZ/WCZ}

Compare S to D (reverse). C = borrow of (S - D). Z = (D == S).

17.8.6_Example_WRD_CMPM_024

CMPM D,{#}S {WC/WZ/WCZ}

Compare D to S, get MSB of difference into C. C = MSB of (D - S).

Z = (D == S).

17.8.7_Example_WRD_CMPSUB_025

CMPSUB D,{#}S {WC/WZ/WCZ}

Compare and subtract S from D if D >= S.

If D >= S then D = D - S and C = 1, else D same and C = 0. *

17.9 F Force FGE\FLE\FGES\FLES

FGE D,{#}S {WC/WZ/WCZ}	Force D >= S. If D < S then D = S and C = 1, else D same and C = 0. *
FLE D,{#}S {WC/WZ/WCZ}	Force D <= S. If D > S then D = S and C = 1, else D same and C = 0. *
FGES D,{#}S {WC/WZ/WCZ}	Force D >= S, signed. If D < S then D = S and C = 1, else D same and C = 0. *
FLES D,{#}S {WC/WZ/WCZ}	Force D <= S, signed. If D > S then D = S and C = 1, else D same and C = 0. *

17.9.1_Example_WRD_FGE_026

FGE D,{#}S {WC/WZ/WCZ}
 Force D >= S. If D < S then D = S and C = 1,
 else D same and C = 0. *

17.9.2_Example_WRD_FLE_027

FLE D,{#}S {WC/WZ/WCZ} Less than or Equal
 Force D <= S. If D > S then D = S and C = 1,
 else D same and C = 0. *

17.9.3_Example_WRD_FGES_028

FGES D,{#}S {WC/WZ/WCZ}
 Force D >= S, **signed**. If D < S then D = S and C = 1,
 else D same and C = 0. *

17.9.4_Example_WRD_FLES_029

FLES D,{#}S {WC/WZ/WCZ}
 Force D <= S, **signed**. If D > S then D = S and C = 1,
 else D same and C = 0. *

17.10) SUM ADD/SUB Based on C/Z

SUMC D,{#}S {WC/WZ/WCZ}	Sum +/-S into D by C. If C = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *
SUMNC D,{#}S {WC/WZ/WCZ}	Sum +/-S into D by !C. If C = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *
SUMZ D,{#}S {WC/WZ/WCZ}	Sum +/-S into D by Z. If Z = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *
SUMNZ D,{#}S {WC/WZ/WCZ}	Sum +/-S into D by !Z. If Z = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

17.10.1_Example_WRD_SUMC_030

SUMC D,{#}S {WC/WZ/WCZ}

Sum +/-S into D by C. If C = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

17.10.2_Example_WRD_SUMNC_031

SUMNC D,{#}S {WC/WZ/WCZ}

Sum +/-S into D by !C. If C = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

17.10.3_Example_WRD_SUMZ_032

SUMZ D,{#}S {WC/WZ/WCZ}

Sum +/-S into D by Z. If Z = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

{{17.10.3_Example_WRD_SUMZ_032}}

"SUMZ D,{#}S {WC/WZ/WCZ}

"Sum +/-S into D by Z. If Z = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

17.10..4_Example_WRD_SUMNZ_033

SUMNZ D,{#}S {WC/WZ/WCZ}

Sum +/-S into D by !Z. If Z = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *

17.11) TEST Register Bit Set Flags

TESTB D,{#}S WC/WZ	Test bit S[4:0] of D, write to C/Z. $C/Z = D[S[4:0]]$.
TESTBN D,{#}S WC/WZ	Test bit S[4:0] of !D, write to C/Z. $C/Z = !D[S[4:0]]$.
TESTB D,{#}S ANDC/ANDZ	Test bit S[4:0] of D, AND into C/Z. $C/Z = C/Z \text{ AND } D[S[4:0]]$.
TESTBN D,{#}S ANDC/ANDZ	Test bit S[4:0] of !D, AND into C/Z. $C/Z = C/Z \text{ AND } !D[S[4:0]]$.
TESTB D,{#}S ORC/ORZ	Test bit S[4:0] of D, OR into C/Z. $C/Z = C/Z \text{ OR } D[S[4:0]]$.
TESTBN D,{#}S ORC/ORZ	Test bit S[4:0] of !D, OR into C/Z. $C/Z = C/Z \text{ OR } !D[S[4:0]]$.
TESTB D,{#}S XORC/XORZ	Test bit S[4:0] of D, XOR into C/Z. $C/Z = C/Z \text{ XOR } D[S[4:0]]$.
TESTBN D,{#}S XORC/XORZ	Test bit S[4:0] of !D, XOR into C/Z. $C/Z = C/Z \text{ XOR } !D[S[4:0]]$.

Test with conditions status of D register bits as requested by S register if set place result in C/Z.

17.11.1_Example_WRD_TESTB_034

TESTB D,{#}S WC/WZ

Test bit S[4:0] of D, write to C/Z. $C/Z = D[S[4:0]]$.

17.11.2_Example_WRD_TESTBN_035

TESTBN D,{#}S WC/WZ

Test bit S[4:0] of !D, write to C/Z. $C/Z = !D[S[4:0]]$.

17.11.3_Example_WRD_TESTB_ANDC/ANDZ_036

TESTB D,{#}S ANDC/ANDZ

Test bit S[4:0] of D, AND into C/Z. $C/Z = C/Z \text{ AND } D[S[4:0]]$.

17.11.4_Example_WRD_TESTBN_ANDC/ANDZ_037

TESTBN D,{#}S ANDC/ANDZ

Test bit S[4:0] of !D, AND into C/Z. $C/Z = C/Z \text{ AND } !D[S[4:0]]$.

17.11.5_Example_WRD_TESTB_ORC/ORZ_038

TESTB D,{#}S ORC/ORZ

Test bit S[4:0] of D, OR into C/Z. $C/Z = C/Z \text{ OR } D[S[4:0]]$.

17.11.6_Example_WRD_TESTBN_ORC_ORZ_039

TESTBN D,{#}S ORC/ORZ

Test bit S[4:0] of !D, OR into C/Z. C/Z = C/Z OR !D[S[4:0]].

17.11.7_Example_WRD_TESTB_XORC_XORZ_040

TESTB D,{#}S XORC/XORZ

Test bit S[4:0] of D, XOR into C/Z. C/Z = C/Z XOR D[S[4:0]].

17.11.8_Example_WRD_TESTBN_XORC_XORZ_041

TESTBN D,{#}S XORC/XORZ

Test bit S[4:0] of !D, XOR into C/Z. C/Z = C/Z XOR !D[S[4:0]].

17.12) BIT Set Bits

BITL D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = 0$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITH D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = 1$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITC D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = C$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITNC D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = !C$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITZ D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = Z$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITNZ D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = !Z$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITRND D,{#}S {WCZ}	Bits $D[S[9:5]+S[4:0]:S[4:0]] = \text{RNDs}$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
BITNOT D,{#}S {WCZ}	Toggle bits $D[S[9:5]+S[4:0]:S[4:0]]$. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].

17.12.1_Example_WRD_BITL_042

BITL D,{#}S {WCZ}

Bits $D[S[9:5]+S[4:0]:S[4:0]] = 0$. Other bits unaffected. Prior SETQ overrides S[9:5].
C,Z = original D[S[4:0]].

This instruction can be used to set a bit to 0 or a group of bits to 0

D = 11111111_11111111_11111111_11111111

To set B0 =0 BitL D,#1

To set B31 =0 BITL D,#31

$D[BH:BL] = D[S[9:5] + S[4:0] : S[4:0]] = 0$

S=%00000000_00000000_000000_B9B8B7B6B5_B4B3B2B1B0

Where BH is high bit to be zero and BL the low bit to be zero of the group of zero bits

Example D[23:16] =0 Let D[31:0] =\$FFFF_FFFF

We want the following Pattern:

D[31:0] = 11111111_00000000_11111111_11111111 BH = 23 BL =16

then $S[9:5] + S[4:0] = 23$ $S[9:5] = 23 - S[4:0] = 23-16 =7$ Then $S[31:10] = 0$ $S[9:5] =7$ $S[4:0] = 16$ S= S

S = %00000000_00000000_000000_00111_10000

BITL D,#240

17.12.2_Example_WRD_BITH_043

BITH D,{#}S {WCZ}

Bits $D[S[9:5]+S[4:0]:S[4:0]] = 1$. Other bits unaffected. Prior SETQ overrides $S[9:5]$.

$C, Z = \text{original } D[S[4:0]]$.

This instruction can be used to set a bit to 1 or a group of bits to 1

$D = \%00000000_00000000_00000000_00000000$

To set $B_0 = 1$ BITH D,#1

To set $B_{31} = 1$ BITH D,#31

$D[BH:BL] = D[S[9:5] + S[4:0] : S[4:0]] = 1$

$S = \%00000000_00000000_00000000_B_9B_8B_7B_6B_5_B_4B_3B_2B_1B_0$

Where BH is high bit to be one and BL the low bit to be one zero of the group of one bits

Example $D[23:16] = 0$ Let $D[31:0] = \$0000_0000$

We want the following Pattern:

$D[31:0] = 00000000_11111111_00000000_00000000$ BH = 23 BL = 16

then $S[9:5] + S[4:0] = 23$ $S[9:5] = 23 - S[4:0] = 23 - 16 = 7$ Then $S[31:10] = 0$ $S[9:5] = 7$ $S[4:0] = 16$

$S = \%00000000_00000000_00000000_111_10000 = 240$

BITH D,#240

17.12.3_Example_WRD_BITC_044

BITC D,{#}S {WCZ}

Bits $D[S[9:5]+S[4:0]:S[4:0]] = C$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.

This instruction can be used to set a bit to C carry or a group of bits to C carry

$D = 11111111_11111111_11111111_11111111$

To set $B_0 = C$ BitC D,#1

To set $B_{31} = C$ BITC D,#31

$D[BH:BL] = D[S[9:5] + S[4:0] : S[4:0]] = 0$

$S = \%00000000_00000000_00000000_B_9B_8B_7B_6B_5_B_4B_3B_2B_1B_0$

Where BH is high bit to be zero and BL the low bit to be zero of the group of zero bits

Example $D[23:16] = 0$ Let $D[31:0] = \$FFFF_FFFF$

We want the following Pattern:

$D[31:0] = 11111111_CCCCCCCC_11111111_11111111$ BH = 23 BL = 16

then $S[9:5] + S[4:0] = 23$ $S[9:5] = 23 - S[4:0] = 23 - 16 = 7$ Then $S[31:10] = 0$ $S[9:5] = 7$ $S[4:0] = 16$

$S = \%00000000_00000000_00000000_111_10000 = 240$

BITC D,#240

17.12.4_Example_WRD_BITNC_045

BITNC D,{#}S {WCZ}

Bits $D[S[9:5]+S[4:0]:S[4:0]] = !C$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.

This instruction can be used to set a bit to !C carry or a group of bits to !C carry

17.12.5_Example_WRD_BITZ_046

BITZ D,{#}S {WCZ}

Bits D[S[9:5]+S[4:0]:S[4:0]] = Z. Other bits unaffected. Prior SETQ overrides S[9:5].

C,Z = original D[S[4:0]].

This instruction can be used to set a bit to Z flag or a group of bits to Z flag

Example D[23:16] = 0 Let D[31:0] = \$FFFF_FFFF

We want the following Pattern:

D[31:0] = 11111111_ZZZZZZZZ_11111111_11111111 BH = 23 BL = 16

then S[9:5] + S[4:0] = 23 S[9:5] = 23 - S[4:0] = 23-16 = 7

Then S[31:10] = 0 S[9:5] = 7 S[4:0] = 16 S=%00000000_00000000_000000_00111_10000 = 240

BITC D,#240

17.12.6_Example_WRD_BITNZ_047

BITNZ D,{#}S {WCZ}

Bits D[S[9:5]+S[4:0]:S[4:0]] = !Z. Other bits unaffected.

Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].

Example D[23:16] = 0 Let D[31:0] = \$FFFF_FFFF

We want the following Pattern:

D[31:0] = 11111111_zzzzzzzz_11111111_11111111 BH = 23 BL = 16 let !Z = z

then S[9:5] + S[4:0] = 23 S[9:5] = 23 - S[4:0] = 23-16 = 7

Then S[31:10] = 0 S[9:5] = 7 S[4:0] = 16 S=%00000000_00000000_000000_00111_10000 = 240

BITC D,#240

17.12.7_Example_WRD_BITRND_048

BITRND D,{#}S {WCZ}

Bits D[S[9:5]+S[4:0]:S[4:0]] = RNDs. Other bits unaffected.

Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].

Value Returned is Randomized

Example D[23:16] = 0 Let D[31:0] = \$FFFF_FFFF

We want the following Pattern:

D[31:0] = 11111111_RRANNDMM_11111111_11111111 BH = 23 BL = 16

then S[9:5] + S[4:0] = 23 S[9:5] = 23 - S[4:0] = 23-16 = 7

Then S[31:10] = 0 S[9:5] = 7 S[4:0] = 16 S=%00000000_00000000_000000_00111_10000 = 240

BITC D,#240

17.12.8_Example_WRD_BITNOT_049

BITNOT D,{#}S {WCZ}

Toggle bits D[S[9:5]+S[4:0]:S[4:0]]. Other bits unaffected. Prior SETQ overrides S[9:5].

C,Z = original D[S[4:0]].

Example D[23:16] = 0 Let D[31:0] = \$FF_%10101010_\$FFFF

We want the following Pattern:

D[31:0] = 11111111_01010101_11111111_11111111 BH = 23 BL = 16

then S[9:5] + S[4:0] = 23 S[9:5] = 23 - S[4:0] = 23-16 = 7

Then S[31:10] = 0 S[9:5] = 7 S[4:0] = 16 S=%00000000_00000000_000000_00111_10000 = 240

BITC D,#240

17.13) AND Boolean AND

AND D,{#}S {WC/WZ/WCZ}	AND S into D. $D = D \& S$. C = parity of result. *
ANDN D,{#}S {WC/WZ/WCZ}	AND !S into D. $D = D \& !S$. C = parity of result. *

17.13.1_Example_WRD_AND_050

AND D,{#}S {WC/WZ/WCZ}

AND S into D. $D = D \& S$. C = parity of result. *

Parity even if Dest has even number of 1 bits C = 0

Parity odd if Dest odd number bits C = 1

D	S	D
0	0	0
0	1	0
1	0	0
1	1	1

Symbol for AND = "&"

17.13.2 Example_WRD_ANDN_051

ANDN D,{#}S {WC/WZ/WCZ}

AND !S into D. $D = D \& !S$. C = parity of result. *

Parity even if Dest has even number of 1 bits C = 0

Parity odd if Dest odd number bits C = 1

D	S	!S	D
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

17.14) OR Boolean OR XOR

OR D,{#}S {WC/WZ/WCZ}	OR S into D. $D = D \mid S$. C = parity of result. *
XOR D,{#}S {WC/WZ/WCZ}	XOR S into D. $D = D \wedge S$. C = parity of result. *

D	S	D
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

D	S	D
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table

Symbol for OR = " \mid "Symbol for XOR = " \wedge "

17.14.1_Example_WRD_OR_052

OR D,{#}S {WC/WZ/WCZ}

OR S into D. $D = D \mid S$. C = parity of result. *

Parity even if Dest has even number of 1 bits C = 0

Parity odd if Dest odd number bits C = 1

17.14.2_Example_WRD_XOR_053

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \wedge S$. C = parity of result. *

Parity even if Dest has even number of 1 bits C = 0

Parity odd if Dest odd number bits C = 1

D	S	D
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table Symbol for exclusive or XOR = " \wedge "

17.15) MUX Mask Destination Register

MUXC D,{#}S {WC/WZ/WCZ}	Mux C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{C\}\})$. C = parity of result. *
MUXNC D,{#}S {WC/WZ/WCZ}	Mux !C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!C\}\})$. C = parity of result. *
MUXZ D,{#}S {WC/WZ/WCZ}	Mux Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{Z\}\})$. C = parity of result. *
MUXNZ D,{#}S {WC/WZ/WCZ}	Mux !Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!Z\}\})$. C = parity of result. *

17.15.1_Example_WRD_MUXC_054

MUXC D,{#}S {WC/WZ/WCZ}

Mux C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{C\}\})$. C = parity of result. *WC -Parity even if Dest has even number of 1 bits $C == 0$ Parity odd if Dest odd number bits $C == 1$

WZ-IF Destination result == 0 then Z =1 else Z = 0

17.15.2_Example_WRD_MUXNC_055

MUXNC D,{#}S {WC/WZ/WCZ}

Mux !C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!C\}\})$. C = parity of result. *WC -Parity even if Dest has even number of 1 bits $C == 1$ Parity odd if Dest odd number bits $C == 0$

WZ-IF Destination result == 0 then Z =1 else Z = 0

17.15.2-Example_MUXNC_055

MUXNC D,{#}S {WC/WZ/WCZ}

Mux !C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!C\}\})$. C = parity of result. *WC -Parity even if Dest has even number of 1 bits $C == 0$ Parity odd if Dest odd number bits $C == 1$

WZ-IF Destination result == 0 then Z =1 else Z = 0

17.15.3_Example_WRD_MUXZ_056

MUXZ D,{#}S {WC/WZ/WCZ}

Mux Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{Z\}\})$. C = parity of result. *WC -Parity even if Dest has even number of 1 bits $C == 0$ Parity odd if Dest odd number bits $C == 1$

WZ-IF Destination result == 0 then Z =1 else Z = 0

17.15.4_Example_WRD_MUXNZ_057

MUXNZ D,{#}S {WC/WZ/WCZ}

Mux !Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!Z\}\})$. C = parity of result. *

"D = $(!S \& D) \mid (S \& \{32\{!Z\}\})$. C = parity of result. *

"WC- Parity even if Dest has even number of 1 bits C == 0

" Parity odd if Dest odd number bits C == 1

"WZ-IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.16) MOV Move From Source to Destination Register

MOV D,{#}S

{WC/WZ/WCZ}

Move S into D. $D = S$. C = S[31]. *

17.16.1_Example_WRD_MOV_058

MOV D,{#}S {WC/WZ/WCZ}

Move S into D. $D = S$. C = S[31]. *

17.17) NOT Negate Destination Register

NOT D,{#}S

{WC/WZ/WCZ}

Get !S into D. $D = !S$. C = !S[31]. *

NOT D {WC/WZ/WCZ}

Get !D into D. $D = !D$. C = !D[31]. *

17.17.1_Example_WRD_NOTDS_059

NOT D,{#}S {WC/WZ/WCZ}

Get !S into D. $D = !S$. C = !S[31]. *

"WC- C = !S[31]

"WZ- IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.17.2_Example_WRD_NOTD_060

NOT D {WC/WZ/WCZ}

Get !D into D. $D = !D$. C = !D[31]. *

"WC- C = !D[31]

"WZ- IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.18) ABS Absolute Value

ABS D,{#}S {WC/WZ/WCZ}	Get absolute value of S into D. D = ABS(S). C = S[31]. *
ABS D {WC/WZ/WCZ}	Get absolute value of D into D. D = ABS(D). C = D[31]. *

17.18.1_Example_WRD_ABSDS_061

```

ABS D,{#}S {WC/WZ/WCZ}
Get absolute value of S into D. D = ABS(S). C = S[31]. *
"WC- C = S[31]
"WZ- IF Destination result == 0 then Z = 1
"  IF Destination result != 0 then Z = 0

```

17.18.2_Example_WRD_ABSD_062

```

ABS D {WC/WZ/WCZ}
Get absolute value of D into D. D = ABS(D). C = D[31]. *
"WC- C = S[31]
"WZ- IF Destination result == 0 then Z = 1
"  IF Destination result != 0 then Z = 0

```

17.19) NEG Negate Negative

NEG D, {#}S {WC/WZ/WCZ}	Negate S into D. D = -S. C = MSB of result. *
NEG D {WC/WZ/WCZ}	Negate D. D = -D. C = MSB of result. *
NEGC D, {#}S {WC/WZ/WCZ}	Negate S by C into D. If C = 1 then D = -S, else D = S. C = MSB of result. *
NEGC D {WC/WZ/WCZ}	Negate D by C. If C = 1 then D = -D, else D = D. C = MSB of result. *
NEGNC D, {#}S {WC/WZ/WCZ}	Negate S by !C into D. If C = 0 then D = -S, else D = S. C = MSB of result. *
NEGNC D {WC/WZ/WCZ}	Negate D by !C. If C = 0 then D = -D, else D = D. C = MSB of result. *
NEGZ D, {#}S {WC/WZ/WCZ}	Negate S by Z into D. If Z = 1 then D = -S, else D = S. C = MSB of result. *
NEGZ D {WC/WZ/WCZ}	Negate D by Z. If Z = 1 then D = -D, else D = D. C = MSB of result. *
NEGNZ D, {#}S {WC/WZ/WCZ}	Negate S by !Z into D. If Z = 0 then D = -S, else D = S. C = MSB of result. *
NEGNZ D {WC/WZ/WCZ}	Negate D by !Z. If Z = 0 then D = -D, else D = D. C = MSB of result. *

17.19.1_Example_WRD_NEGDS_063

NEG D, {#}S {WC/WZ/WCZ}
 Negate S into D. D = -S. C = MSB of result. *
 WC- C = MSB
 WZ- IF Destination result == 0 then Z = 1
 IF Destination result != 0 then Z = 0

17.19.2_Example_WRD_NEGD_064

NEG D {WC/WZ/WCZ}
 Negate D. D = -D. C = MSB of result. *
 WC- C = MSB
 WZ- IF Destination result == 0 then Z = 1
 IF Destination result != 0 then Z = 0

17.19.3_Example_WRD_NEGCDS_065

NEGC D, {#}S {WC/WZ/WCZ}

Negate S by C into D. If C = 1 then D = -S, else D = S. C = MSB of result. *

"WC- C = MSB

"WZ- IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.19.4_Example_WRD_NEGCD_066

NEGC D {WC/WZ/WCZ}

Negate D by C. If C = 1 then D = -D, else D = D. C = MSB of result. *

WC- C = MSB

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.19.5_Example_WRD_NEGNCDS_067

NEGNC D, {#}S {WC/WZ/WCZ}

Negate S by !C into D. If C = 0 then D = -S, else D = S. C = MSB of result. *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.19.6_Example_WRD_NEGNCD_068

NEGNC D {WC/WZ/WCZ}

Negate D by !C. If C = 0 then D = -D, else D = D. C = MSB of result. *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.19.7_Example_WRD_NEGZDS_069

NEGZ D, {#}S {WC/WZ/WCZ}

Negate S by Z into D. If Z = 1 then D = -S, else D = S. C = MSB of result. *

"WC- C = MSB

"WZ- IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.19.8_Example_WRD_NEGZD_070

NEGZ D {WC/WZ/WCZ}

Negate D by Z. If Z = 1 then D = -D, else D = D. C = MSB of result. *

"WC- C = MSB

"WZ- IF Destination result == 0 then Z = 1

" IF Destination result != 0 then Z = 0

17.19.9_Example_WRD_NEGNZDS_071

NEGNZ D,{#}S {WC/WZ/WCZ}

Negate S by !Z into D. If Z = 0 then D = -S, else D = S. C = MSB of result. *

WC- C = MSB

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.19.10_Example_WRD_NEGNZ_072

NEGNZ D {WC/WZ/WCZ}

Negate D by !Z. If Z = 0 then D = -D, else D = D. C = MSB of result. *

WC- C = MSB

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.20) INCMOD/DECMOD Increment Modulus

73	INCMOD D,{#}S {WC/WZ/WCZ}	Increment with modulus. If D = S then D = 0 and C = 1, else D = D + 1 and C = 0. *
74	DECMOD D,{#}S {WC/WZ/WCZ}	Decrement with modulus. If D = 0 then D = S and C = 1, else D = D - 1 and C = 0. *

S can be thought as the modulus. See section E.3 Modular Arithmetic. As an example modulus 12 used for time clock count goes from 0-12 when count at 12 reset 0 to allow increment. IF D = 0 wanting to go back in time by decrementing D reset 12 to decrement to 11.

17.20.1_Example_WRD_INCMOD_073

INCMOD D,{#}S {WC/WZ/WCZ}

Increment with modulus. If D = S then D = 0 and C = 1, else D = D + 1 and C = 0. *

If D = S then D = 0 and C = 1, else D = D + 1 and C = 0. *

WC- IF D = 0 then C = 1 else C = 0

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.20.2_Example_WRD_DECMOD_074

DECMOD D,{#}S {WC/WZ/WCZ}

Decrement with modulus. If D = 0 then D = S and C = 1, else D = D - 1 and C = 0. *

If D = 0 then D = S and C = 1, else D = D - 1 and C = 0. *

WC- IF D = S then C = 1 else C = 0

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.21 Zerop/Sign Extend

ZEROX D,{#}S {WC/WZ/WCZ}	Zero-extend D above bit S[4:0]. C = MSB of result. *
-----------------------------	--

SIGNX D,{#}S {WC/WZ/WCZ}	Sign-extend D from bit S[4:0]. C = MSB of result. *
-----------------------------	---

L[31:0] = B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄_ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆_ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈_ B₇B₆B₅B₄B₃B₂B₁B₀

S[4:0] = upto this value bits will remain the same

ZEROX = 0 will fill in all bits beyond S[4:0] value

SIGNX = sign value 1 will be filled in all bits beyond S[4:0] value

17.21.1_Example_WRD_ZEROX_075

ZEROX D,{#}S {WC/WZ/WCZ}

Zero-extend D above bit S[4:0]. C = MSB of result. *

ZEROX = 0 will fill in all bits beyond S[4:0] value

WC- C = MSB of result.

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄_ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆_ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈_ B₇B₆B₅B₄B₃B₂B₁B₀

17.21.2_Example_WRD_SIGNX_076

SIGNX D,{#}S {WC/WZ/WCZ}

Sign-extend D from bit S[4:0]. C = MSB of result. *

SIGNX = sign value 1 will be filled in all bits beyond S[4:0] value

WC- C = MSB of result.

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄_ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆_ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈_ B₇B₆B₅B₄B₃B₂B₁B₀

17.22) ENCOD Get top Bit Position

ENCOD D,{#}S {WC/WZ/WCZ}	Get bit position of top-most '1' in S into D. D = position of top '1' in S (0..31). C = (S != 0). *
ENCOD D {WC/WZ/WCZ}	Get bit position of top-most '1' in D into D. D = position of top '1' in S (0..31). C = (S != 0). *

17.22.1_Example_WRD_ENCOWDS_077

ENCOD D,{#}S {WC/WZ/WCZ}

Get bit position of top-most '1' in S into D. D = position of top '1' in S (0..31). C = (S != 0). *

D = position of top '1' in S (0..31). C = (S != 0). *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈ B₇B₆B₅B₄B₃B₂B₁B₀

17.22.1_Example_WRD_ENCOWD_078

ENCOD D {WC/WZ/WCZ}

Get bit position of top-most '1' in D into D. D = position of top '1' in S (0..31). C = (S != 0). *

D = position of top '1' in S (0..31). C = (S != 0). *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈ B₇B₆B₅B₄B₃B₂B₁B₀

17.23) Ones Count number of 1's in S Put value in D

ONES D, {#}S {WC/WZ/WCZ}	Get number of '1's in S into D. D = number of '1's in S (0..32). C = LSB of result. *
ONES D {WC/WZ/WCZ}	Get number of '1's in D into D. D = number of '1's in S (0..32). C = LSB of result. *

17.23.1_Example_WRD_ONESDS_079

ONES D, {#}S {WC/WZ/WCZ}

Get number of '1's in S into D. D = number of '1's in S (0..32). C = LSB of result. *

Number of ones in S are counted and this value is put in D

D = number of '1's in S (0..32). C = LSB of result. *

C = LSB of result. *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄_ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆_ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈_ B₇B₆B₅B₄B₃B₂B₁B₀

17.23.2_Example_WRD_ONESD_080

ONES D {WC/WZ/WCZ}

Get number of '1's in D into D. D = number of '1's in S (0..32). C = LSB of result. *

WC- C = LSB of result. *

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄_ B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆_ B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀B₉B₈_ B₇B₆B₅B₄B₃B₂B₁B₀

17.24) TEST set carry base on test D&S (no register changes)

TEST D,{#}S {WC/WZ/WCZ}	Test D with S. C = parity of (D & S). Z = ((D & S) == 0).
TEST D {WC/WZ/WCZ}	Test D. C = parity of D. Z = (D == 0).
TESTN D,{#}S {WC/WZ/WCZ}	Test D with !S. C = parity of (D & !S). Z = ((D & !S) == 0).

17.24.1_Example_WRD_TESTDS_081

TEST D,{#}S {WC/WZ/WCZ}

Test D with S. C = parity of (D & S). Z = ((D & S) == 0).

WC- C = parity of (D & S) odd parity = 1 even parity = 0

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.24.2_Example_WRD_TESTD_082

TEST D {WC/WZ/WCZ}

Test D. C = parity of D. Z = (D == 0).

WC- C = parity of (D) odd parity = 1 even parity = 0

WZ- IF Destination result == 0 then Z = 1

IF Destination result != 0 then Z = 0

17.24.3_Example_WRD_TESTND_83

TESTN D,{#}S {WC/WZ/WCZ}

Test D with !S. C = parity of (D & !S). Z = ((D & !S) == 0).

WC- C = parity of (D) odd parity = 1 even parity = 0

WZ- Z = ((D & !S) == 0)

17.25) SETNIB Set Nibble

SETNIB D,{#}S,#N	Set S[3:0] into nibble N in D, keeping rest of D same.
SETNIB {#}S	Set S[3:0] into nibble established by prior ALTSN instruction.

$S[31:0] = N_7N_6N_5N_4N_3N_2N_1N_0$ $N_7 = n_{73}n_{72}n_{71}n_{70}$ $N_6 = n_{63}n_{62}n_{61}n_{60}$ $N_5 = n_{53}n_{52}n_{51}n_{50}$
 $N_7 = n_{73}n_{72}n_{71}n_{70}$ $N_6 = n_{63}n_{62}n_{61}n_{60}$ $N_5 = n_{53}n_{52}n_{51}n_{50}$ $N_4 = n_{43}n_{42}n_{41}n_{40}$
 $N_3 = n_{33}n_{32}n_{31}n_{30}$ $N_2 = n_{23}n_{22}n_{21}n_{20}$ $N_1 = n_{13}n_{12}n_{11}n_{10}$ $N_0 = n_{03}n_{02}n_{01}n_{00}$

The idea is that **D/** can serve as a register base address and D can be used as an index.

ALTSN (offset + N field),BaseAddress

Next Instruction :

D Field = (D[11:3] + S)&1FF N Field = D[2:0]

ALTSN D,{#}S

Alter subsequent SETNIB instruction. Next D field = (D[11:3] + S) & \$1FF, N field = D[2:0].

D += sign-extended S[17:9].

- 1) D field = (D[11:3] + S) & \$1FF register to have SETNIB D,{#}S,#N point too
- 2) S is the BaseAddress and D[11:3] is the offset from the Base
- 3) N field = D[2:0] nibble to write too in SETNIB D,{#}S,#N

17.25.1_Example_WRD_SETNIBDS_084

SETNIB D,{#}S,#N

Set S[3:0] into nibble N in D, keeping rest of D same.

The nibble Number N is a fixed value and cannot be indexed

$S[31:0] = N_7N_6N_5N_4N_3N_2N_1N_0$

$N_7 = n_{73}n_{72}n_{71}n_{70}$ $N_6 = n_{63}n_{62}n_{61}n_{60}$ $N_5 = n_{53}n_{52}n_{51}n_{50}$ $N_4 = n_{43}n_{42}n_{41}n_{40}$

$N_3 = n_{33}n_{32}n_{31}n_{30}$ $N_2 = n_{23}n_{22}n_{21}n_{20}$ $N_1 = n_{13}n_{12}n_{11}n_{10}$ $N_0 = n_{03}n_{02}n_{01}n_{00}$

17.25.2_Example_WRD_ALTSN D_SETNIB {#}S_085

SETNIB {#}S

Set S[3:0] into nibble established by prior ALTSN instruction.

This allows the Indexing of the nibble Number N

ALTSN D

Alter subsequent SETNIB instruction. Next D field = D[11:3], N field = D[2:0].

S[31:0] = N7N6_N5N4_N3N2_N1N0

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

PR0=\$1D8 PR1=\$1D9 PR2=\$1DA PR3=\$1DB PR4=\$1DC PR5=\$1DD PR6=\$1DE PR7=\$1DF

Example

Set up the SETNIB {#}S Instruction to write to register PR0 nibble N7-N0

Solution D[11:3] = PR0 = \$1D8 = 472 = %111011000 D[2:0] = \$7 = 7 = %111

Dest = %00000000_00000000_0000_111011000_111

= %00000000_00000000_00001110_11000111

= \$EC7

= 3783

ALTSN Dest

17.25.3_Example_WRD_ALTSN D,{#},#N_SETNIB D,{#}S#N_084

SETNIB D,{#}S,#N

Set S[3:0] into nibble N in D, keeping rest of D same.

The nibble Number N is a fixed value and cannot be indexed

1) D is set by ALTSN D,{#}S where D is an indexed possible register to be written too.

2) S[3:0] is the nibble value to be written. It is not from ALTSN D,{#}S

3) N is nibble to be written set by ALTSN D,{#}S

ALTSN D,{#}S

Alter subsequent SETNIB instruction. Next D field = (D[11:3] + S) & \$1FF, N field = D[2:0].

D += sign-extended S[17:9].

1) D field = (D[11:3] + S) & \$1FF register to have SETNIB D,{#}S,#N point too

2) S is the BaseAddress and D[11:3] is the offset from the Base

3) N field = D[2:0] nibble to write too in SETNIB D,{#}S,#N

17.26) GETNIB Get nibble from register

GETNIB D,{#}S,#N	Get nibble N of S into D. D = {28'b0, S.NIBBLE[N]}.
GETNIB D	Get nibble established by prior ALTGN instruction into D.

$$S[31:0] = N_7N_6N_5N_4N_3N_2N_1N_0$$

$$N_7 = n_{73}n_{72}n_{71}n_{70} \quad N_6 = n_{63}n_{62}n_{61}n_{60} \quad N_5 = n_{53}n_{52}n_{51}n_{50} \quad N_4 = n_{43}n_{42}n_{41}n_{40}$$

$$N_3 = n_{33}n_{32}n_{31}n_{30} \quad N_2 = n_{23}n_{22}n_{21}n_{20} \quad N_1 = n_{13}n_{12}n_{11}n_{10} \quad N_0 = n_{03}n_{02}n_{01}n_{00}$$

GETNIB D,{#}S,#N

Get nibble N of S into D. D = {28'b0, S[N4+3:N4]}

1) N is nibble number 0-7 in S that is to be moved to D N0

2) 28'b0 stands for 28 bits of type 0 the {,} stands for concatenate (join)

3) S.NIBBLE[N] is the nibble position with right to left level of significance. (B3B2B1B0)

Note: D = {28'b0, S[N4+3:N4]} this is Verilog notation see section E.4

GETNIB D

Get nibble established by prior ALTGN instruction into D.

ALTGN D,{#}S (104)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = (D[11:3] + S) & \$1FF, N field = D[2:0]. D += sign-extended S[17:9].

1) D field = (D[11:3] + S) & \$1FF register to have SETNIB D,{#}S,#N point too

2) S is the BaseAddress and D[11:3] is the offset from the Base

3) N field = D[2:0] nibble to write too in GETNIB D,{#}S,#N

ALTGN D (105)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = D[11:3], N field = D[2:0].

17.26.1_EXAMPLE_WRD_GETNIB D,{#}S,#N_086

Get nibble N of S into D. D = D = {28'b0, S[N4+3:N4]}

GETNIB D,{#}S,#N

Get nibble N of S into D. D = {28'b0, S[N*4+3:N*4]}

1) N nibble of N7N6N5N4N3N2N1N0

2) S register with nibbles required

3) D is target register of the nibble

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

Example

Get Nibble N7 in register Src and move to Dest register

17.26.2_EXAMPLE_GETNIB D_087

GETNIB D

Get nibble established by prior ALTGN instruction into D.

ALTGN D,{#}S

Alter subsequent GETNIB/ROLNIB instruction.

Next S field = (D[11:3] + S) & \$1FF, N field = D[2:0]. D += sign-extended S[17:9].

1) S is the addressBase D[11:3]

2) S Field = (D[11:3] + S) & \$1FF, D[11:3] is the offset index from addressBase

3) N Field = D[2:0]

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

Example

Get Nibble N7 in register Src and move to Dest register N0

17.26.3_Example_WRD_GETNIB D_087

GETNIB D

Get nibble established by prior ALTGN instruction into D.

ALTGN D

Alter subsequent GETNIB/ROLNIB instruction. Next S field = D[11:3], N field = D[2:0].

1) D[11:3] is the addressBase

2) N Field = D[2:0]

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

Example

Get Nibble N7 in register Src and move to Dest register N0

{{17.26.3_Example_WRD_GETNIB D_087}}

GETNIB D

Get nibble established by prior ALTGN instruction into D.

1) D is target register of the nibble

17.27) ROLNIB Rotate Nibble

ROLNIB D,{#}S,#N	Rotate-left nibble N of S into D. D = {D[27:0], S.NIBBLE[N]}.
ROLNIB D	Rotate-left nibble established by prior ALTGN instruction into D.

$$S[31:0] = N_7N_6N_5N_4N_3N_2N_1N_0$$

$$N_7 = n_{73}n_{72}n_{71}n_{70} \quad N_6 = n_{63}n_{62}n_{61}n_{60} \quad N_5 = n_{53}n_{52}n_{51}n_{50} \quad N_4 = n_{43}n_{42}n_{41}n_{40}$$

$$N_3 = n_{33}n_{32}n_{31}n_{30} \quad N_2 = n_{23}n_{22}n_{21}n_{20} \quad N_1 = n_{13}n_{12}n_{11}n_{10} \quad N_0 = n_{03}n_{02}n_{01}n_{00}$$

ROLNIB D,{#}S,#N

Rotate-left nibble N of S into D. D = {D[27:0], S.NIBBLE[N]}.

- 1) N is nibble number 7 in S that is to be moved to D to N0
 - 2) 28'b0 stands for 28 bits of type 0 the {,} stands for concatenate (join)
 - 3) S.NIBBLE[N] is the nibble position with right to left level of significance .(B3B2B1B0)
- Note: D = D = {28'b0, S[N4+3:N4]} this is Verilog notation see section E.4

ROLNIB D

Rotate-left nibble established by prior ALTGN instruction into D.

ALTGN D,{#}S (104)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = (D[11:3] + S) & \$1FF, N field = D[2:0]. D += sign-extended S[17:9].

- 1) D field = (D[11:3] + S) & \$1FF register to have SETNIB D,{#}S,#N point too
- 2) S is the BaseAddress and D[11:3] is the offset from the Base
- 3) N field = D[2:0] nibble to write too in GETNIB D,{#}S,#N

ALTGN D (105)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = D[11:3], N field = D[2:0].

17.27.1_Example_WRD_ROLNIBDSN_088

ROLNIB D,{#}S,#N

Rotate-left nibble N of S into D. $D = \{D[27:0], S.NIBBLE[N]\}$.

S contains the nibble values the N7 nibble is placed into D nibble N0.

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

Example

Get Nibble N7 in register Src and move to Dest register

17.27.2_Example_WRD_ROLNIBD_089

ROLNIB D (089)

Rotate-left nibble established by prior ALTGN instruction into D.

1) D is the register that will contain nibble pointed from ALTGN statement

ALTGN D,{#}S (104)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. D += sign-extended $S[17:9]$.

1) D field = $(D[11:3] + S) \& \$1FF$ register to have SETNIB D,{#}S,#N point too

2) S is the BaseAddress and $D[11:3]$ is the offset from the Base

3) N field = $D[2:0]$ nibble to write too in GETNIB D,{#}S,#N

Note:

S with an offset can point to different words holding nibbles and the N field can point different nibbles in the different words. The Nibble is always in the ROLNIB D register.

17.27.3_Example_WRD_ROLNIBDSN_ALTGNDS_088

ROLNIB D,{#}S,#N

Rotate-left nibble N of S into D. $D = \{D[27:0], S.NIBBLE[N]\}$.

ALTGN D,{#}S (104)

Alter subsequent GETNIB/ROLNIB instruction. Next S field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. D += sign-extended $S[17:9]$.

1) D field = $(D[11:3] + S) \& \$1FF$ register to have SETNIB D,{#}S,#N point too

2) S is the BaseAddress and $D[11:3]$ is the offset from the Base

3) N field = $D[2:0]$ nibble to write too in GETNIB D,{#}S,#N

N7 = n73n72n71n70 N6 = n63n62n61n60 N5 = n53n52n51n50 N4 = n43n42n41n40

N3 = n33n32n31n30 N2 = n23n22n21n20 N1 = n13n12n11n10 N0 = n03n02n01n00

Note:

S with an offset can point to different words holding nibbles and the N field can point different nibbles in the different words. The Nibble is always in the ROLNIB D register.

17.28) SETBYTE Set Byte N into Register

SETBYTE D,{#}S,#N	Set S[7:0] into byte N in D, keeping rest of D same.
SETBYTE {#}S	Set S[7:0] into byte established by prior ALTSB instruction.

Byte Addressing

D[31:0] = D₃D₂D₁D₀

Bit Addressing

D[31:0]

= d₃₁d₃₀d₂₉d₂₈d₂₇d₂₆d₂₅d₂₄_d₂₃d₂₂d₂₁d₂₀d₁₉d₁₈d₁₇d₁₆_d₁₅d₁₄d₁₃d₁₂d₁₁d₁₀d₀₉d₀₈_d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂d₀₁d₀₀

Byte Addressing

S[31:0] = S₃S₂S₁S₀

Byte Bit Addressing

S[31:0] = S₃₇S₃₆S₃₅S₃₄S₃₃S₃₂S₃₁S₃₀_S₂₇S₂₆S₂₅S₂₄S₂₃S₂₂S₂₁S₂₀_S₁₇S₁₆S₁₅S₁₄S₁₃S₁₂S₁₁S₁₀_S₀₇S₀₆S₀₅S₀₄S₀₃S₀₂S₀₁S₀₀

Next D Field D[10:2] = d₁₀d₀₉d₀₈d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂ 9 bit address range \$000-\$1FF (0 – 511)

Word Field D[1:0] = d₀₁d₀₀ 2 bit address range 0-3

17.28.1_Example_WRD_SETBYTE D,{#}S,#N_090

SETBYTE D,{#}S,#N

Set S[7:0] into byte N in D, keeping rest of D same.

Byte Addressing

D[31:0] = D₃D₂D₁D₀

= d₃₁d₃₀d₂₉d₂₈d₂₇d₂₆d₂₅d₂₄_d₂₃d₂₂

d₂₁d₂₀d₁₉d₁₈d₁₇d₁₆_d₁₅d₁₄d₁₃d₁₂d₁₁d₁₀d₀₉d₀₈_d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂d₀₁d₀₀

Byte Addressing

S[31:0] = S₃S₂S₁S₀

= S₃₇S₃₆S₃₅S₃₄S₃₃S₃₂S₃₁S₃₀_S₂₇S₂₆S₂₅S₂₄S₂₃S₂₂S₂₁S₂₀_S₁₇S₁₆S₁₅S₁₄S₁₃S₁₂S₁₁S₁₀_

S₀₇S₀₆S₀₅S₀₄S₀₃S₀₂S₀₁S₀₀

Example

Set S[7:0] = %10101010 and move this byte into B2 in D

17.28.2_Example_WRD_SETBYTE {#}S_091

SETBYTE {#}S

Set S[7:0] into byte established by prior ALTSB instruction.

ALTSB D

Alter subsequent SETBYTE instruction. Next D field = D[10:2], N field = D[1:0].

D[31:0] = D₃D₂D₁D₀

= d₃₁d₃₀d₂₉d₂₈d₂₇d₂₆d₂₅d₂₄d₂₃d₂₂d₂₁d₂₀d₁₉d₁₈d₁₇d₁₆d₁₅d₁₄d₁₃d₁₂d₁₁d₁₀d₀₉d₀₈d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂d₀₁d₀₀

Next D Field D[10:2] = d₁₀d₀₉d₀₈d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂ 9 bit address range \$000-\$1FF (0 – 511)

Word Field D[1:0] = d₀₁d₀₀ 2 bit address range 0-3

Example

Set S[7:0] = %10101010 and move this byte into D2 in D

17.28.3_Example_WRD_SETBYTE D,{#}S,#N_ALTSB D,{#}S_090

SETBYTE D,{#}S,#N

Set S[7:0] into byte N in D, keeping rest of D same.

ALTSB D,{#}S

Alter subsequent SETBYTE instruction. Next D field = (D[10:2] + S) & \$1FF, N field = D[1:0]. D += sign-extended S[17:9].

Byte Addressing

D[31:0] = D₃D₂D₁D₀

=

d₃₁d₃₀d₂₉d₂₈d₂₇d₂₆d₂₅d₂₄d₂₃d₂₂d₂₁d₂₀d₁₉d₁₈d₁₇d₁₆d₁₅d₁₄d₁₃d₁₂d₁₁d₁₀d₀₉d₀₈d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂d₀₁d₀₀

Byte Addressing

S[31:0] = S₃S₂S₁S₀

= S₃₇S₃₆S₃₅S₃₄S₃₃S₃₂S₃₁S₃₀S₂₇S₂₆S₂₅S₂₄S₂₃S₂₂S₂₁S₂₀S₁₇S₁₆S₁₅S₁₄S₁₃S₁₂S₁₁S₁₀S₀₇S₀₆S₀₅S₀₄S₀₃S₀₂S₀₁S₀₀

Example

Set S[7:0] = %10101010 and move this byte into B2 in D

To test example modify Dest_Offset and Byte_Offset

17.29) GETBYTE Get Byte N of S into D

GETBYTE D,{#}S,#N	Get byte N of S into D. D = {24'b0, S.BYTE[N]}.
GETBYTE D	Get byte established by prior ALTGB instruction into D.

17.29.1_Example_WRD_GETBYTE D,{#}S,#N_092

GETBYTE D,{#}S,#N

Get byte N of S into D. D = {24'b0, S.BYTE[N]}.

Byte Addressing

D[31:0] = D3D2D1D0

= d31d30d29d28d27d26d25d24_d23d2

2d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00

Byte Addressing

S[31:0] = S3S2S1S0

= S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00

Example

Load S = 1000_0000__1000_0001__1000_0010__

17.29.2_Example_WRD_ALTGB_GETBYTE D_093

GETBYTE D

Get byte established by prior ALTGB instruction into D.

ALTGB D

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = D[10:2], N field = D[1:0].

D[31:0] = D3D2D1D0

= d31d30d29d28d27d26d25d24_d23d2

2d21d20d19d18d17d16_d15d14d13d12d11_d10d09d08d07d06d05d04d03d02_d01d00

Next D Field D[10:2] = d10d09d08d07d06d05d04d03d02 9 bit address range \$000-\$1FF (0 - 511)

Word Field D[1:0] = d01d00 2 bit address range 0-3

Example

Set S[7:0] = %10101010 and move this byte into D2 in D

17.29.3_Example_WRD_ALTGB D,S_GETBYTE D,{#}S,#N

GETBYTE D,{#}S,#N

Get byte N of S into D. $D = \{24'b0, S.BYTE[N]\}$.

ALTGB D,{#}S

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = $(D[10:2] + S) \& \$1FF$, N field = $D[1:0]$. D += sign-extended $S[17:9]$.

Byte Addressing

 $D[31:0] = D3D2D1D0$ $= d31d30d29d28d27d26d25d24_d23d2$ $2d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00$

Byte Addressing

 $S[31:0] = S3S2S1S0$ $= S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00$

Example

BaseAddress = D_GETBYTE0 Offset = 0-3 S Field = BaseAddress + Offset

ByteNum = N = 0,1,2,3 Byte to be taken from long (BaseAddress + Offset)

17.30) ROLBYTE Rotate Left Byte N of S Into D

ROLBYTE D,{#}S,#N

Rotate-left byte N of S into D. $D = \{D[23:0], S.BYTE[N]\}$.

ROLBYTE D

Rotate-left byte established by prior ALTGB instruction into D.

17.30.1_Example_WRD_ROLBYTE D,{#}S,#N_094

ROLBYTE D,{#}S,#N

Rotate-left byte N of S into D. $D = \{D[23:0], S.BYTE[N]\}$.

Byte Addressing

 $D[31:0] = D3D2D1D0$ $= d31d30d29d28d27d26d25d24_d23d2$ $2d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00$

Byte Addressing

 $S[31:0] = S3S2S1S0$ $= S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00$

Example

SElect Byte in selByteLong and rotate left into valLong do this for all 4 byte 0-3

17.30.2_Example_WRD_ALTGB D_ROLBYTE D_095

ROLBYTE D

Rotate-left byte established by prior ALTGB instruction into D.

ALTGB D

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = D[10:2], N field = D[1:0].

Note: Rotate-left byte N of S into D. $D = \{D[23:0], S.BYTE[N]\}$.

17.30.3_Example_WRD_ALTGB D,{#}S_ROLBYTE D,{#}S,#N_094

ROLBYTE D,{#}S,#N

Rotate-left byte N of S into D. $D = \{D[23:0], S.BYTE[N]\}$.

ALTGB D,{#}S

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = $(D[10:2] + S) \& \$1FF$, N field = D[1:0]. D += sign-extended S[17:9].

Byte Addressing

$D[31:0] = D3D2D1D0$

$= d31d30d29d28d27d26d25d24_d23d2$

$2d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00$

Byte Addressing

$S[31:0] = S3S2S1S0$

$= S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00$

Example

Select Byte from Src_Register and Shift to D_ROLBYTE reverse order of Src_Register is the result

17.31) SETWORD Set S[15:0] into Word N in D

SETWORD D,{#}S,#N	Set S[15:0] into word N in D, keeping rest of D same.
SETWORD {#}S	Set S[15:0] into word established by prior ALTSW instruction.

SETWORD D,{#}S,#N (096)

Set S[15:0] into word N in D, keeping rest of D same.

SETWORD {#}S (097)

Set S[15:0] into word established by prior ALTSW instruction.

ALTSW D,{#}S (110)

Alter subsequent SETWORD instruction. Next D field = (D[9:1] + S) & \$1FF, N field = D[0]. D += sign-extended S[17:9].

ALTSW D (111)

Alter subsequent SETWORD instruction. Next D field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

17.31.1_Example_WRD_SETWORD D,{#}S,#N_096

SETWORD D,{#}S,#N

Set S[15:0] into word N in D, keeping rest of D same.

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

D_SETWORD is to have selWord written to word 0 and word 1

17.31.2_Example_WRD_SETWORD {#}S_097

SETWORD {#}S (097)

Set S[15:0] into word established by prior ALTSW instruction.

ALTSW D (111)

Alter subsequent SETWORD instruction. Next D field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

D_SETWORD is to have selWord written to word 0 and word 1

17.31.3_Example_WRD_ALTSW D,{#}S_SETWORD D,{S},#N

SETWORD D,{#}S,#N (096)

Set S[15:0] into word N in D, keeping rest of D same.

ALTSW D,{#}S (110)

Alter subsequent SETWORD instruction. Next D field = (D[9:1] + S) & \$1FF, N field = D[0]. D += sign-extended S[17:9].

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

Write 'valWord' to W0 of 'Src_Register0' then write 'valWord' to W1 of 'Src_Register1'

17.32) GETWORD Get Word N of S into D

GETWORD D,{#}S,#N	Get word N of S into D. D = {16'b0, S.WORD[N]}.
-------------------	---

GETWORD D	Get word established by prior ALTGW instruction into D.
-----------	---

GETWORD D,{#}S,#N (098)

Get word N of S into D. D = {16'b0, S.WORD[N]}.

GETWORD D (099)

Get word established by prior ALTGW instruction into D.

ALTGW D,{#}S (112)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].

ALTGW D (113)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

17.32.1_Example_WRD_GETWORD D,{#}S,#N _098

GETWORD D,{#}S,#N (098)

Get word N of S into D. D = {16'b0, S.WORD[N]}.

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

write W0 of 'sellong' to 'D_GETWORD' then write W1 to 'D_GETWORD'

17.32.2_Example_WRD_ALTGW D_GETWORD D_099

GETWORD D (099)

Get word established by prior ALTGW instruction into D.

ALTGW D (113)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

Write 'selWord' W0 too 'D_GETWORD' then write W1 to 'D_GETWORD' note upper bits cleared in 'D_GETWORD'

17.32.3_Example_WRD_ALTGW D,{#},S_GETWORD D,{#}S,#N_098

GETWORD D,{#}S,#N (098)

Get word N of S into D. D = {16'b0, S.WORD[N]}.

ALTGW D,{#},S (112)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

Write W0 of 'Src_Register0' too 'D_GETWORD' then write W1 of 'Src_Register1' too 'D_GETWORD'

17.33) ROLWORD Rotate Left Word N of S Into D

ROLWORD D,{#}S,#N	Rotate-left word N of S into D. D = {D[15:0], S.WORD[N]}.
ROLWORD D	Rotate-left word established by prior ALTGW instruction into D.

ROLWORD D,{#}S,#N (100)
 Rotate-left word N of S into D. D = {D[15:0], S.WORD[N]}.

ROLWORD D (101)
 Rotate-left word established by prior ALTGW instruction into D.

ALTGW D,{#}S (112)
 Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].

ALTGW D (113)
 Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =
 w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_
 w7w6w5w4w3w2w1w0
 W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000
 W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

17.33.1_Example_WRD_ROLWORD D,{#}S_100

ROLWORD D,{#}S,#N (100)
 Rotate-left word N of S into D. D = {D[15:0], S.WORD[N]}.

Word Addressing
 W[31:0] = W1W0 =
 w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_
 w7w6w5w4w3w2w1w0
 W0 = w015w014w013w012w01w010w009w008w007w006w005w004w003w002w001w000
 W1 = w115w114w113w112w11w110w109w108w107w106w105w104w103w102w101w100

Example

'selLong' rotates N 0,1,2,3, of 'S_ROLWORD' into 'D_ROLWORD' keep the same byte order

17.33.2_Example_WRD_ALTGW D_ROLWORD D_101

ROLWORD D (101)

Rotate-left word established by prior ALTGW instruction into D.

```
{{17.33.2_Example_WRD_ALTGW D_ROLWORD {#}S_101}}
```

```
{{
```

```
ROLWORD D (101)
```

Rotate-left word established by prior ALTGW instruction into D.

ALTGW D (113)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

Word Addressing

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w011w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w111w110w109w108w107w106w105w104w103w102w101w100

Example

Rotate left Word W0 from 'Src_Long0' to 'D_ROLWORD'

then Rotate left Word W1 from Src_Long1 to 'D_ROLWORD'

17.33.3_Example_WRD_ALTGW D,{#},S_ ROLWORD D,{#}S,#N_100

ROLWORD D,{#}S,#N (100)

Rotate-left word N of S into D. D = {D[15:0], S.WORD[N]}.

ALTGW D,{#},S (112)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].

W[31:0] = W1W0 =

w31w30w29w28w27w26w25w24_w23w22w21w20w19w18w17w16_w15w14w13w12w11w10w9w8_w7w6w5w4w3w2w1w0

W0 = w015w014w013w012w011w010w009w008w007w006w005w004w003w002w001w000

W1 = w115w114w113w112w111w110w109w108w107w106w105w104w103w102w101w100

Example

ROTATE left W0 of 'Src_Register0' too 'D_ROLWORD' then ROTATE left W1 of 'Src_Register1' too 'D_ROLWORD'

17.34) ALTSN Alter Susequent SETNIB Instruction

ALTSN D,{#}S	Alter subsequent SETNIB instruction. Next D field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. $D += \text{sign-extended } S[17:9]$.
ALTSN D	Alter subsequent SETNIB instruction. Next D field = $D[11:3]$, N field = $D[2:0]$.

See 17.25) SETNIB Set Nibble for examples

The idea is that $S/\#$ can serve as a register base address and D can be used as an index.

ALTSN (offset + N field),BaseAddress

Next Instruction :

D Field = $(D[10:2] + S) \& 1FF$ N Field = $D[1:0]$

ALTGN D,{#}S

Alter subsequent SETNIB instruction. Next D field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$.

$D += \text{sign-extended } S[17:9]$.

1) D field = $(D[10:2] + S) \& \$1FF$ register to have SETBYTE {#}S,#N point too

2) S is the BaseAddress and $D[11:3]$ is the offset from the Base

3) N field = $D[2:0]$ byte to write too in SETBYTE D,{#}S,#N

17.34.1_Example_WRD_ALTSN D,{#}S_102

ALTSN D,{#}S

Alter subsequent SETNIB instruction. Next D field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. $D += \text{sign-extended } S[17:9]$.

17.34.2.103_Example_WRD_ALTSN D_103

ALTSN D

Alter subsequent SETNIB instruction. Next D field = $D[11:3]$, N field = $D[2:0]$.

17.35) ALTGN Alter Subsequent GETNIB/ROLNIB Instruction

ALTGN D,{#}S	Alter subsequent GETNIB/ROLNIB instruction. Next S field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. D += sign-extended S[17:9].
ALTGN D	Alter subsequent GETNIB/ROLNIB instruction. Next S field = $D[11:3]$, N field = $D[2:0]$.

See 17.26) GETNIB Getnibble from register

See 17.27) ROLNIB Rotate Nibble

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTGN (offset + N field),BaseAddress

Next Instruction :

S Field = $(D[11:3] + S) \& 1FF$ N Field = $D[2:0]$

ALTGN D,{#}S

Alter subsequent SETNIB instruction. Next D field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$.

D += sign-extended S[17:9].

1) S Field = $(D[11:3] + S) \& \$1FF$ register to have SETNIB {#}S,#N point too

2) S is the BaseAddress and $D[11:3]$ is the offset from the Base

3) N field = $D[2:0]$ nibble to write too in SETNIB D,{#}S,#N

17.35.1_Example_WRD_ALTGN D,{#}S_104

ALTGN D,{#}S

Alter subsequent GETNIB/ROLNIB instruction. Next S field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. D += sign-extended S[17:9].

17.35.2_Example_WRD_ALTGN D_105

ALTGN D

Alter subsequent GETNIB/ROLNIB instruction. Next S field = $D[11:3]$, N field = $D[2:0]$.

17.36) ALTSB Alter Subsequent SETBYTE Instruction

106	ALTSB D,{#}S	Alter subsequent SETBYTE instruction. Next D field = $(D[10:2] + S) \& \$1FF$, N field = D[1:0]. D += sign-extended S[17:9].
107	ALTSB D	Alter subsequent SETBYTE instruction. Next D field = D[10:2], N field = D[1:0].

See 17.28) SETBYTE Set Byte N into Register

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTSB (offset + N field),BaseAddress

Next Instruction :

D Field = $(D[10:2] + S) \& 1FF$ N Field = D[1:0]

ALTGN D,{#}S

Alter subsequent SETNIB instruction. Next D field = $(D[10:2] + S) \& \$1FF$, N field = D[1:0].

D += sign-extended S[17:9].

1) D field = $(D[10:2] + S) \& \$1FF$ register to have SETBYTE D,#N point too

2) S is the BaseAddress and D[10:2] is the offset from the Base

3) N field = D[1:0] byte to write too in SETBYTE D,{#}S,#N

17.36.1_Example_WRD_ALTSB D,{#}S_106

ALTSB D,{#}S

Alter subsequent SETBYTE instruction. Next D field = $(D[10:2] + S) \& \$1FF$, N field = D[1:0]. D += sign-extended S[17:9].

17.36.2_Example_WRD_ALTSB D_107

ALTSB D

Alter subsequent SETBYTE instruction. Next D field = D[10:2], N field = D[1:0].

17.37) ALTGB Alter Subsequent GETBYTE/ROLBYTE

ALTGB D,{#}S	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = (D[10:2] + S) & \$1FF, N field = D[1:0]. D += sign-extended S[17:9].
ALTGB D	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = D[10:2], N field = D[1:0].

See 17.29) GETBYTE Get Byte N of S into D

See 17.30) ROLBYTE Rotate Left Byte N of S Into D

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTGB (offset + N field),BaseAddress

Next Instruction :

S Field = (D[10:2] + S)&1FF N Field = D[1:0]

ALTGB D,{#}S

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = (D[10:2] + S) & \$1FF, N field = D[1:0].
D += sign-extended S[17:9].

1) S field = (D[10:2] + S) & \$1FF register to have GETBYTE/ROLBYTE D,{#}S,#N point too

2) S is the BaseAddress and D[10:2] is the offset from the Base

3) N field = D[1:0] byte to write too in SETBYTE D,{#}S,#N

17.37.1_Example_WRD_ALTGB D,{#}S_108

ALTGB D,{#}S

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = (D[10:2] + S) & \$1FF, N field = D[1:0]. D += sign-extended S[17:9].

17.37.2_Example_WRD_ALTGB_109

ALTGB D

Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = D[10:2], N field = D[1:0].

17.38) ALTSW Alter Subsequent SETWORD

ALTSW D,{#}S	Alter subsequent SETWORD instruction. Next D field = $(D[9:1] + S) \& \$1FF$, N field = D[0]. D += sign-extended S[17:9].
ALTSW D	Alter subsequent SETWORD instruction. Next D field = D[9:1], N field = D[0].

See 17.31) SETWORD Set S[15:0] into Word N in D

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTSW (offset + N field),BaseAddress

Next Instruction :

D Field = $(D[10:2] + S) \& \$1FF$ N Field = D[1:0]

ALTGN D,{#}S

Alter subsequent SETNIB instruction. Next D field = $(D[10:2] + S) \& \$1FF$, N field = D[1:0].

D += sign-extended S[17:9].

1) D field = $(D[10:2] + S) \& \$1FF$ register to have SETBYTE D,{#}S,#N point too

2) S is the BaseAddress and D[10:2] is the offset from the Base

3) N field = D[1:0] byte to write too in SETBYTE D,{#}S,#N

17.38.1_Example_WRD_ALTSW D,{#}S_110

ALTSW D,{#}S

Alter subsequent SETWORD instruction. Next D field = $(D[9:1] + S) \& \$1FF$, N field = D[0]. D += sign-extended S[17:9].

17.38.2_Example_WRD_ALTSW D_111

ALTSW D

Alter subsequent SETWORD instruction. Next D field = D[9:1], N field = D[0].

17.39 ALTGW Alter Subsequent GETWORD

ALTGW D,{#}S	Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].
ALTGW D	Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

See 17.32) GETWORD Get Word N of S into D

ALTGW (offset + N field),BaseAddress

Next Instruction :

S Field = (D[9:1] + S)&1FF N Field = D[0]

ALTGW D,{#}S

Alter subsequent GETWORD instruction. Next S field = (D[9:1] + S) & \$1FF, N field = D[0].
D += sign-extended S[17:9].

- 1) S field = (D[9:1] + S) & \$1FF register to have GETWORD {#}S,#N point too
- 2) S is the BaseAddress and D[9:1] is the offset from the Base
- 3) N field = D[0] byte to write too in GETWORD {#}S,#N

17.39.1_Example_WRD ALTGW D,{#}S_112

ALTGW D,{#},S (112)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & \$1FF), N field = D[0]. D += sign-extended S[17:9].

17.39.1_Example_WRD ALTGW D_113

ALTGW D (113)

Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].

17.40) ALTR D Alter result Register D of next instruction

ALTR D,{#}S	Alter result register address (normally D field) of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
ALTR D	Alter result register address (normally D field) of next instruction to D[8:0].

$RDS_{address} = BaseAddressS[8:0] + OffsetS[17:9] + IndexD[8:0]$

$S = OffsetS[17:9] + BaseAddressS[8:0]$

$D = IndexD[8:0] + \text{sign-Extended}[17:9]$

Hardware Registers

DIRA \$1FA Output enables for P31..P0
 DIRB \$1FB Output enables for P63..P32
 OUTA \$1FC Output states for P31..P0
 OUTB \$1FD Output states for P63..P32
 INA \$1FE Input states from P31..P0
 INB \$1FF Input states from P63..P32

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \oplus S$. C = parity of result. *

For some reason (quite a reach for XOR INA,INB) we want the result of XOR X,Y but you don't want to destroy register X. By using the ALTR instruction you can avoid a bunch of move statements. Also some registers cannot be written too. Using the ALTR instruction you can use the instructions without destroying either register. Typically in PASM the D register is where the instruction result is stored.

ALTR index,#table 'set next write to table+index

XOR INA,INB 'write $INA \oplus INB$ to register[table+index]

ALTR D,{#}S

AlternateRegister = (D + S) & \$1FF

D = Offset(Index)

S = BaseAddress(Table)

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \oplus S$. C = parity of result. *

17.40.1_Example_WRD_ALTR D,{#}S_114

ALTR D,{#}S

Alter result register address (normally D field) of next instruction to $(D + S) \& \$1FF$. D += sign-extended S[17:9].

By Means of an example we want the result of XOR X,Y but you don't want to destroy register X.

By using the ALTR instruction you can avoid a bunch of move statements.

Also some registers cannot be written too. Using the ALTR instruction you can use the assembly instructions without destroying either register and writing the instruction operation to an alternate register.

ALTR index,#table 'set next write to table+index

XOR INA,INB 'write $INA \wedge INB$ to register[table+index]

Raddress= BaseAdressS[8:0] + OffsetS[17:9] + IndexD[8:0]

S= Offset S[17.9] + BaseAddressS[8:0]

D = IndexD[8:0] + sign-Extended [17:9]

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \wedge S$. C = parity of result. *

Example

Write the result of XOR Ax,Bx to 'xorResult' not affecting Ax or Bx

17.40.2_Example_WRD_ALTR D_115

ALTR D

Alter result register address (normally D field) of next instruction to D[8:0].

By Means of an example we want the result of XOR X,Y but you don't want to destroy register X.

By using the ALTR instruction you can avoid a bunch of move statements.

Also some registers cannot be written too. Using the ALTR instruction you can use the assembly instructions without destroying either register and writing the instruction operation to an alternate register.

XOR D,{#}S {WC/WZ/WCZ}

XOR S into D. $D = D \wedge S$. C = parity of result. *

Example

Write the result of XOR Ax,Bx to 'xorResult' not affecting Ax or Bx

17.41) ALTD D Alter D Field of next Instruction

ALTD D,{#}S	Alter D field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
ALTD D	Alter D field of next instruction to D[8:0].

$RDS_{address} = BaseAddressS[8:0] + OffsetS[17:9] + IndexD[8:0]$

$S = OffsetS[17:9] + BaseAddressS[8:0]$

$D = IndexD[8:0] + \text{sign-Extended } [17:9]$

17.41.1_Example_WRD_ALTD D,{#}S_116

ALTD D,{#}S

Alter D field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].

MOV D,{#}S {WC/WZ/WCZ}

Move S into D. $D = S$. $C = S[31]$. *

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTS Offset,BaseAddress

Next Instruction :

$S_{Field} = (D + S) \& 1FF$

$D[31:0] = D3D2D1D0$

$D[31:0] =$

d31d30d29d28d27d26d25d24d23d22d21d20d19d18d17d16d15d14d13d12d11d10d09d08d07d06d05d04d03d02d01d00

$S[31:0] = S3S2S1S0$

$S[31:0] =$

s31s30s29s28s27s26s25s24s23s22s21s20s19s18s17s16s15s14s13s12s11s10s09s08s07s06s05s04s03s02s01s00

$D_{Field} = (D + S) \& \$1FF = 00000000_00000000_00000000_d08d07d06d05d04d03d02d01d00$ typical 9 bit address

Example

Self Modifying Code Alter D Field allows 'Pointer' to be indexed through a table

17.41.2_Example_WRD_ALTD D_117

ALTD D

Alter D field of next instruction to D[8:0].

MOV D,{#}S {WC/WZ/WCZ} (058) EEEE 0110000 CZI DDDDDDDDD SSSSSSSSS

Move S into D. D = S. C = S[31]. *

D Field = D & \$1FF = 00000000_00000000_00000000_d08d07d06d05d04d03d02d01d00 typical 9 bit address

Example

D[31:0] = D3D2D1D0

D[31:0]

=d31d30d29d28d27d26d25d24_d23d22d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00

S[31:0] = S3S2S1S0

S[31:0] = S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00

Example

'D_ALTD' Destination Register write value 'S_MOV' too 'valTable0'

17.42) ALTS Alter S field of next Instruction

ALTS D,{#}S	Alter S field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
ALTS D	Alter S field of next instruction to D[8:0].

$RDS_{address} = BaseAddressS[8:0] + OffsetD[17:9] + IndexD[8:0]$

$S = OffsetS[17:9] + BaseAddressS[8:0]$

$D = IndexD[8:0] + \text{sign-Extended}[17:9]$

Byte Addressing

$D[31:0] = D_3D_2D_1D_0$

$S[31:0] = S_3S_2S_1S_0$

Bit Addressing

$D[31:0] = d_{31}d_{30}d_{29}d_{28}d_{27}d_{26}d_{25}d_{24}d_{23}d_{22}d_{21}d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_{09}d_{08}d_{07}d_{06}d_{05}d_{04}d_{03}d_{02}d_{01}d_{00}$

$S[31:0] = s_{37}s_{36}s_{35}s_{34}s_{33}s_{32}s_{31}s_{30_}s_{27}s_{26}s_{25}s_{24}s_{23}s_{22}s_{21}s_{20_}s_{17}s_{16}s_{15}s_{14}s_{13}s_{12}s_{11}s_{10_}s_{07}s_{06}s_{05}s_{04}s_{03}s_{02}s_{01}s_{00}$

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTS Offset,BaseAddress

Next Instruction :

S Field = (D + S) & 1FF

ALTS D,{#}S

Alter subsequent instruction. Next S field = (D + S) & \$1FF

D += sign-extended S[17:9].

ALTS D

Alter S field of next instruction to D[8:0]. Next S Field = D[8:0]

17.42.1_Example_WRD_ALTS D,{#}S_118

ALTS D,{#}S

Alter S field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].

MOV D,{#}S {WC/WZ/WCZ}

Move S into D. D = S. C = S[31]. *

The idea is that S/# can serve as a register base address and D can be used as an index.

ALTS Offset,BaseAddress

Next Instruction :

S Field = (D + S) & 1FF

D[31:0] = D3D2D1D0

D[31:0]

=d31d30d29d28d27d26d25d24_d23d22d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00

S[31:0] = S3S2S1S0

S[31:0] = S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00

S Field = (D + S) & \$1FF =00000000_00000000_00000000_d08d07d06d05d04d03d02d01d00 typical 9 bit address

Example

Self Modifying Code Alter S Field allows 'S_Pointer + Index' to be indexed through a table

17.42.2_Example_WRD_ALTS D_119

ALTS D

Alter S field of next instruction to D[8:0].

MOV D,{#}S {WC/WZ/WCZ} (058) EEEE 0110000 CZI DDDDDDDDD SSSSSSSSS

Move S into D. D = S. C = S[31]. *

S Field = D & \$1FF =00000000_00000000_00000000_d08d07d06d05d04d03d02d01d00 typical 9 bit address

Example

D[31:0] = D3D2D1D0

D[31:0]

=d31d30d29d28d27d26d25d24_d23d22d21d20d19d18d17d16_d15d14d13d12d11d10d09d08_d07d06d05d04d03d02d01d00

S[31:0] = S3S2S1S0

S[31:0] = S37S36S35S34S33S32S31S30_S27S26S25S24S23S22S21S20_S17S16S15S14S13S12S11S10_S07S06S05S04S03S02S01S00

17.43) ALTB D Alter D field of next instruction usually associated with Bits

ALTB D,{#}S	Alter D field of next instruction to $(D[13:5] + S) \& \$1FF$. D += sign-extended S[17:9].
ALTB D	Alter D field of next instruction to D[13:5].

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

TESTB D,{#}S WC/WZ (034)
 Test bit S[4:0] of D, write to C/Z. C/Z = D[S[4:0]].

BITC D,{#}S {WCZ} (044)
 Bits $D[S[9:5]+S[4:0]:S[4:0]] = C$. Other bits unaffected. Prior SETQ overrides S[9:5].
 C,Z = original D[S[4:0]].
 This instruction can be used to set a bit to C carry or a group of bits to C carry

17.43.1 Example WRD_ALTB D,{#}S_120

ALTB D,{#}S
 Alter D field of next instruction to $(D[13:5] + S) \& \$1FF$. D += sign-extended S[17:9].

ALTB D,{#}S (120)
 Alter D field of next instruction to $(D[13:5] + S) \& \$1FF$. D += sign-extended S[17:9].

$S = \text{OffsetBitIndexS}[17:9] \mid \text{BaseAddressS}[8:0]$
 $D = \text{IndexWordD}[13:5] \mid \text{BitIndexD}[4:0]$

Next Instruction D Field = $(D[13:5] + S[17:9] + S) \& \$1FF$
 Note: S[17:9] is a bit offset in addition to the BitIndex

TESTB D,{#}S WC/WZ (034)
 Test bit S[4:0] of D, write to C/Z. C/Z = D[S[4:0]].

Example

Use ALTB BaseAddress with Index and Offset to test bit

17.43.2_Example_WRD_ALTB D_121

ALTB D

Alter D field of next instruction to D[13:5].

TESTB D,{#}S WC/WZ (034)

Test bit S[4:0] of D, write to C/Z. C/Z = D[S[4:0]].

Example

Use ALTB BaseAddress

Defined in valBit0 bit position B9 S[4:0= %1001} = 9

Check by changing word valBit0 B9 from 1 to zero

17.44) ALTI Substitute next instructions I/R/D/S Fields

ALTI D,{#}S	Substitute next instruction's I/R/D/S fields with fields from D, per S. Modify D per S.
ALTI D	Execute D in place of next instruction. D stays same.

I = Instruction Field [27:21]

R = Result Field [27:19]

D = Destination Field [17:9]

S = Source Field [8:0]

Condition	Instruction	Effects(Flags)	Destination	Source
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00	E E E E 0 0 0 0 0 0 0	C Z I	D D D D D D D D D	S S S S S S S S S
4 bit = 15	7 bit = 127	3 bit = 7	9 bit = 511 max address	9 bit = 511 max address

First, ALTI For more complex S field, D field, and result register substitutions, there is the ALTI instruction. ALTI actually does a few different things 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:

'set next D and S fields, increment ptrs[17:9] and ptrs[8:0]

ALTI ptrs,##%111_111

ADD 0,0 'add registers

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

NOP 'NOP becomes inst

Note: Not going to do examples at this time modifying Instructions not of high priority for learning existing instructions. If someone wants to submit an example please do so.

17.44.1_Example_WRD_ALTI D,{#}S_122

ALTI D,{#}S

Substitute next instruction's I/R/D/S fields with fields from D, per S. Modify D per S.

17.44.2_Exempl_WRD_ALTI D_123

ALTI D

Execute D in place of next instruction. D stays same.

17.45) SETR SETD SETS Set Instruction Field of Register

SETR D,{#}S	Set R field of D to S[8:0]. D = {D[31:28], S[8:0], D[18:0]}.
SETD D,{#}S	Set D field of D to S[8:0]. D = {D[31:18], S[8:0], D[8:0]}.
SETS D,{#}S	Set S field of D to S[8:0]. D = {D[31:9], S[8:0]}.

I = Instruction Field [27:21]

R = Result Field [27:19]

D = Destination Field [17:9]

S = Source Field [8:0]

Condition	Instruction	Effects(Flags)	Destination	Source
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00				
E E E E	0 0 0 0 0 0 0	C Z I	D D D D D D D D	S S S S S S S S
4 bit = 15	7 bit = 127	3 bit = 7	9 bit = 511 max address	9 bit = 511 max address

Ea IF_Z MOV PRO,PR1 WZMOV D,{#}S {WC/WZ/WCZ} EEEE 0110000 CZI DDDDDDDDD SSSSSSSSS

IF_Z EEEE = 1010

MOV Instruction = 0110000

C Effects C = 0 (no WC)

Z Effects Z = 1 (WZ)

I Effects I = 0 (# not present)

PRO DDDDDDDDD = \$1D8 = %111011000

PR1 SSSSSSSSSS = \$1D9 = %111011001

Condition	Instruction	Effects(Flags)	Destination	Source
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00				
1 0 1 0	0 1 1 0 0 0 0	0 1 0	1 1 1 0 1 1 0 0	1 1 1 0 1 1 0 0
4 bit = 15	7 bit = 127	3 bit = 7	9 bit = 511 max address	9 bit = 511 max address

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]

S/# = operate on bit field contained in S[31:0] or #value bit field #[31:0]

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:

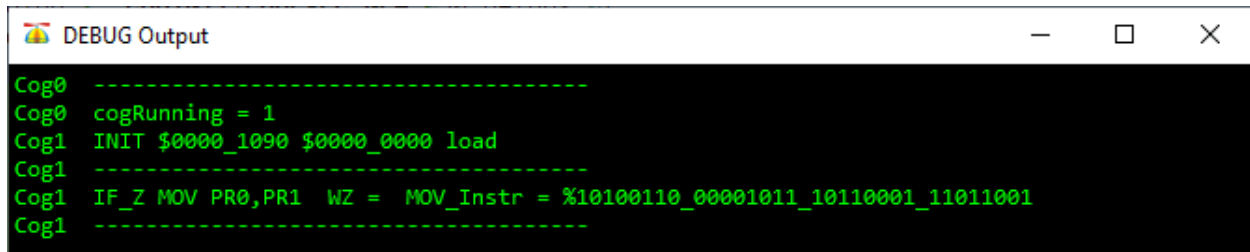
```
SETR    inst,op      'set reg[27:19] to op[8:0]
NOP                                           'first spacer instruction, could be anything
NOP                                           'second spacer instruction, could be anything
inst MOV    x,y      'operate on x using y, MOV can become
AND/OR/XOR/etc.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

Not going to do examples at this time modifying instructions not of high priority for learning existing instructions. If someone wants to submit an example please do so.

17.45.1_Example_WRD_Get_MOV_Instruction_Code

This program can be used to get the actual instruction code value.

Example: Get “IF_Z MOV PR0,PR1 WZ “ binary instruction code.



```
DEBUG Output
Cog0 -----
Cog0 cogRunning = 1
Cog1 INIT $0000_1090 $0000_0000 load
Cog1 -----
Cog1 IF_Z MOV PR0,PR1 WZ = MOV_Instr = %10100110_00001011_10110001_11011001
Cog1 -----
```

17.46) DECOD value 0-31 into Long with Corresponding bit set High

DECOD D,{#}S	Decode S[4:0] into D. $D = 1 \ll S[4:0]$.
DECOD D	Decode D[4:0] into D. $D = 1 \ll D[4:0]$.

DECOD PASM instruction performs the same way as Bitwise Decode spin operator

$|< (Pin := |< PinNum$ Bitwise Decode decodes a value (0-31) into a 32 bit long value with a single bit set high corresponding to the bit position of the original value

17.46.1_Example_WRD_DECOD D,{S}#_127

DECOD D,{#}S

Decode S[4:0] into D. $D = 1 \ll S[4:0]$.

17.46.2_Example_WRD_DECOD D_128

DECOD D

Decode D[4:0] into D. $D = 1 \ll D[4:0]$.

17.47) BMASKD Get LSB Justified bit mask

BMASK D,{#}S	Get LSB-justified bit mask of size (S[4:0] + 1) into D. $D = (\$0000_0002 \ll S[4:0]) - 1$.
BMASK D	Get LSB-justified bit mask of size (D[4:0] + 1) into D. $D = (\$0000_0002 \ll D[4:0]) - 1$.

17.47.1_Example_WRD_BMASK D,{#}S_129

BMASK D,{#}S

Get LSB-justified bit mask of size (S[4:0] + 1) into D. $D = (\$0000_0002 \ll S[4:0]) - 1$.

Example

Get Mask from 0-31 and then Mask out TestMask value

17.47.2_Example_WRD_BMASK D_130

BMASK D

Get LSB-justified bit mask of size (D[4:0] + 1) into D. $D = (\$0000_0002 \ll D[4:0]) - 1$.

Example

Get Mask from 0-31 and then Mask out TestMask value

17.48) CRCBIT Cyclic Reduncy Check of Byte

CRCBIT D,{#}S

Iterate CRC value in D using C and polynomial in S. If (C XOR D[0]) then D = (D >> 1) XOR S, else D = (D >> 1).

CRCNIB D,{#}S

Iterate CRC value in D using Q[31:28] and polynomial in S. Like CRCBIT x 4. Q = Q << 4. Use 'REP #n,#1'+SETQ+CRCNIB+CRCNIB+CRCNIB...

Note: See [“Appendix E.2\) CRC8 Cycle Redundancy Check”](#)

17.48.1_Example_WRD_CRCBIT D,{#}S_131

CRCBIT D,{#}S

Iterate CRC value in D using C and polynomial in S. If (C XOR D[0]) then D = (D >> 1) XOR S, else D = (D >> 1).

17.48.2_Example_WRD_CRCNIB D,{#}S_132

CRCNIB D,{#}S

Iterate CRC value in D using Q[31:28] and polynomial in S.

Like CRCBIT x 4. Q = Q << 4. Use 'REP #n,#1'+SETQ+CRCNIB+CRCNIB+CRCNIB...

17.48.3_Example_WRD_CRCBIT_Function

This is included with Program examples it is a means to emulate what CRCBIT performs.

Testing crcbit instruction Chris Gadd source

crcbit crc,POLY performs the operations:

```
testb crc,#0 wz
shr  crc,#1
if_c_ne_z xor  crc,POLY
```

17.48.4_Example_WRD_CRCBIT D,{#}S_131

CRCBIT D,{#}S

Iterate CRC value in D using C and polynomial in S. If (C XOR D[0]) then D = (D >> 1) XOR S, else D = (D >> 1).

Example

send n bytes to have string CRC generated code . n is set for one for one byte testing ,modify n for multiple bytes

17.49) MUXNITS/MUXNIB D,{#}S Set bit in D from S

133	MUXNITS D,{#}S	For each non-zero bit pair in S, copy that bit pair into the corresponding D bits, else leave that D bit pair the same.
134	MUXNIBS D,{#}S	For each non-zero nibble in S, copy that nibble into the corresponding D nibble, else leave that D nibble the same.

17.49.1_Example_WRD_MUXNITS D,{#}S _133

MUXNITS D,{#}S

For each non-zero bit pair in S, copy that bit pair into the corresponding D bits, else leave that D bit pair the same.

Example

"Copy bit pattern in S to D"

17.49.2_Example_WRD_MUXNIBS D,{#}S _134

MUXNIBS D,{#}S

For each non-zero nibble in S, copy that nibble into the corresponding D nibble, else leave that D nibble the same.

Example

"Copy bit pattern in S to D"

17.50) MUXQ D,{#}S

MUXQ D,{#}S	Used after SETQ. For each '1' bit in Q, copy the corresponding bit in S into D. $D = (D \& !Q) (S \& Q)$.
-------------	--

17.50.1_Example_WRD_MUXQ D,{#}S _135

MUXQ D,{#}S

Used after SETQ. For each '1' bit in Q, copy the corresponding bit in S into D. $D = (D \& !Q) | (S \& Q)$.

SETQ {#}D

Set Q to D. Use before RDLONG/WRLONG/WMLONG to set block transfer.

Also used before MUXQ/COGINIT/QDIV/QFRAC/QROTATE/WAITxxx.

Example

"Copy bit pattern in S to D masked by Q"



17.51) MOVBYT D,{#}S move bytes within a register

MOVBYTS D,{#}S

Move bytes within D, per S. D = {D.BYTE[S[7:6]], D.BYTE[S[5:4]], D.BYTE[S[3:2]], D.BYTE[S[1:0]]}.

Byte Addressing $D_B[3:0]$ $=D_3D_2D_1D_0$ **Byte Bit Addressing** $D_{BB}[37:00]$ $=d_{37}d_{36}d_{35}d_{34}d_{33}d_{32}d_{31}d_{30}d_{27}d_{26}d_{25}d_{24}d_{23}d_{22}d_{21}d_{20}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_{07}d_{06}d_{05}d_{04}d_{03}d_{02}d_{01}d_{00}$ $D_3 = S(7:6)=0-3 \quad D_2 = S(5:4)=0-3 \quad D_1 = S(3:2)=0-3 \quad D_0 = S(1:0)=0-3$

17.51_Example_WRD_MOVBYTS D,{#}S_136

MOVBYTS D,{#}S

Move bytes within D, per S. D = {D.BYTE[S[7:6]], D.BYTE[S[5:4]], D.BYTE[S[3:2]], D.BYTE[S[1:0]]}.

Example

Reverse bytes in a register

MOVBYTS D_MOVBYT,S_MOVBYT 'S(7:6)=0 S(5:4)=1 S(3:2)=2 S(1:0)=3

17.52) MUL D, {#}S Multiply D x S

MUL D, {#}S {WZ}	D = unsigned (D[15:0] * S[15:0]). Z = (S == 0) (D == 0).
MULS D, {#}S {WZ}	D = signed (D[15:0] * S[15:0]). Z = (S == 0) (D == 0).

17.52.1_Example_WRD_MUL D, {#}S {WZ}_137

MUL D, {#}S {WZ}

D = unsigned (D[15:0] * S[15:0]). Z = (S == 0) | (D == 0).

Example

Multiply two unsigned numbers

17.52.2_Example_WRD_MULS D, {#}S {WZ}_138

MULS D, {#}S {WZ}

D = signed (D[15:0] * S[15:0]). Z = (S == 0) | (D == 0).

Example

Multiply two signed numbers

17.53) SCA D,{#}S Multiply and shift result

SCA D,{#}S {WZ}	Next instruction's S value = unsigned (D[15:0] * S[15:0]) >> 16. *
SCAS D,{#}S {WZ}	Next instruction's S value = signed (D[15:0] * S[15:0]) >> 14. In this scheme, \$4000 = 1.0 and \$C000 = -1.0. *

17.53.1_Example_WRD_SCA D,{#}S {WZ}_139

SCA D,{#}S {WZ}

Next instruction's S value = unsigned (D[15:0] * S[15:0]) >> 16. *

Example

Multiply two unsigned numbers

17.53.2_Example_WRD_SCAS D,{#}S {WZ}_140

SCAS D,{#}S {WZ}

Next instruction's S value = signed (D[15:0] * S[15:0]) >> 14. In this scheme, \$4000 = 1.0 and \$C000 = -1.0. *

17.54) ADDPIX D,{#}S for setting pix intensity

ADDPIX D,{#}S	Add bytes of S into bytes of D, with \$FF saturation.
---------------	---

17.54.1_Example-WRD_ADDPIX D,{#}S_141

ADDPIX D,{#}S

Add bytes of S into bytes of D, with \$FF saturation.

Example

add D3+S3 ,D2+S2,D1+S1,D0+S0 max value in DX = FF

17.55) MULPIX D,{#}S Multiply Bytes Dx*Sx_142 (info from AJL/ARIBA)

MULPIX D,{#}S

Multiply bytes of S into bytes of D, where \$FF = 1.0 and \$00 = 0.0.

$$D_B[3:0] = D_3D_2D_1D_0 \quad S_B[3:0] = S_3S_2S_1S_0$$

$$D_3 = D_3 * S_3 / 255 \quad D_2 = D_2 * S_2 / 255 \quad D_1 = D_1 * S_1 / 255 \quad D_0 = D_0 * S_0 / 255$$

From the instruction description, with \$FF = 1.0 and \$00 = 0.0 then with D of \$0080FFFF and S of \$008080FF I would expect a result of something like \$004080FF.

That's:

$$\text{\$00} \times \text{\$00} = 0.0 \times 0.0 = 0.0 = \text{\$00}$$

$$\text{\$80} \times \text{\$80} = 0.5 \times 0.5 = 0.25 = \text{\$40}$$

$$\text{\$80} \times \text{\$FF} = 0.5 \times 1.0 = 0.5 = \text{\$80}$$

$$\text{\$FF} \times \text{\$FF} = 1.0 \times 1.0 = 1.0 = \text{\$FF}$$

17.55.1_Example_WRD_MULPIX D,{#}S_142

MULPIX D,{#}S

Multiply bytes of S into bytes of D, where \$FF = 1.0 and \$00 = 0.0.

17.56) BLNPIX D,{#}S_143 Alpha-blend bytes of S into bytes of D Using SETPIV

BLNPIX D,{#}S Alpha-blend bytes of S into bytes of D, using SETPIV value.

BLNPIX D,{#}S (142)

Alpha-blend bytes of S into bytes of D, using SETPIV value.

Note: Personnel Knowledge not capable of describing or using

17.57) MIXPIX D,{#}S_144 Mix bytes of S into bytes of D using SETPIX and SETPIV

MIXPIX D,{#}S Mix bytes of S into bytes of D, using SETPIX and SETPIV values.

MIXPIX D,{#}S (144)

Mix bytes of S into bytes of D, using SETPIX and SETPIV values.

Note: Personnel Knowledge not capable of describing or using

17.58) ADDCT1\ADDCT2\ADDCT3 D,{#}S_145\146\147 Counter Passed event

ADDCT1 D,{#}S	Set CT1 event to trigger on $CT = D + S$. Adds S into D.
ADDCT2 D,{#}S	Set CT2 event to trigger on $CT = D + S$. Adds S into D.
ADDCT3 D,{#}S	Set CT3 event to trigger on $CT = D + S$. Adds S into D.

Typical CTxUsage

```

GETCT    cog1CountValue           'the counter value is now in in cog1CountValue
ADDCT1   cog1CountValue,cog1WaitTime 'add counter tick for delay set event CT1
        'the cog1CountValue + cog1WaitTime = CT1 result is placed in the CT1 event register
WAITCT1                                     'wait for cog1 program counter to pass CT1
cog1WaitTime    long    150_000_000
cog1CountValue  long    200_000_000

```

Wait for CTx event flag to be passed

```

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

```

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.

17.58.1_Example_ADDCTX_WAITCTX

Demonstrate wait for counter event CT1,CT2,CT3 Using GETCT\ADDCTXWAITCTX
Each cog has separate CTx events

17.58.2_Example_WRD_ADDCTX_POLLCTX

Demonstrate polling (check and continue) counter event CT1,CT2,CT3 Using GETCT\ADDCTXWAITCTX
Each cog has separate CTx events

17.59) WMLONG Write Non \$00 bytes in D to HubAddress

WMLONG D,{#}S/P

Write only non-\$00 bytes in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.

WMLONG D,{#}S/P

Write only non-\$00 bytes in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.

17.60) 149 RD Read Pin, LUT (LookUpTable) Byte, Word, Long (149..154)

RQPIN D, {#}S {WC}	Read smart pin S[5:0] result "Z" into D, don't acknowledge smart pin ("Q" in RQPIN means "quiet"). C = modal result.
RDPIN D, {#}S {WC}	Read smart pin S[5:0] result "Z" into D, acknowledge smart pin. C = modal result.
RDLUT D, {#}S/P {WC/WZ/WCZ}	Read data from LUT address {#}S/PTRx into D. C = MSB of data. *
RDBYTE D, {#}S/P {WC/WZ/WCZ}	Read zero-extended byte from hub address {#}S/PTRx into D. C = MSB of byte. *
RDWORD D, {#}S/P {WC/WZ/WCZ}	Read zero-extended word from hub address {#}S/PTRx into D. C = MSB of word. *
RDLONG D, {#}S/P {WC/WZ/WCZ}	Read long from hub address {#}S/PTRx into D. C = MSB of long. * Prior SETQ/SETQ2 invokes cog/LUT block transfer.

17.60.1_Example_WRD_RQPIN_149

RQPIN D, {#}S {WC}

Read smart pin S[5:0] result "Z" into D, don't acknowledge smart pin ("Q" in RQPIN means "quiet"). C = modal result.

17.60.2_Example_WRD_RDPIN_150

RDPIN D, {#}S {WC}

Read smart pin S[5:0] result "Z" into D, acknowledge smart pin. C = modal result.

17.60.3_Example_WRD_RDLUT_151

RDLUT D, {#}S/P {WC/WZ/WCZ}

Read data from LUT address {#}S/PTRx into D. C = MSB of data. *

17.60.4_Example_WRD_RDBYTE_152

RDBYTE D, {#}S/P {WC/WZ/WCZ}

Read zero-extended byte from hub address {#}S/PTRx into D. C = MSB of byte. *

17.65.5_Example_WRD_RDWORD_153

RDWORD D, {#}S/P {WC/WZ/WCZ}

Read zero-extended word from hub address {#}S/PTRx into D. C = MSB of word. *

17.66.6_Example_WRD_RDLONG_154

RDLONG D, {#}S/P {WC/WZ/WCZ}

Read long from hub address {#}S/PTRx into D. C = MSB of long. * Prior SETQ/SETQ2 invokes cog/LUT block transfer.

17.61) POPA\POPB Read long from HUB Address pointed to by PTR\PTRB

POPA D {WC/WZ/WCZ}	Read long from hub address --PTRA into D. C = MSB of long. *
--------------------	--

POPB D {WC/WZ/WCZ}	Read long from hub address --PTRB into D. C = MSB of long. *
--------------------	--

17.61.1) POPA D {WC/WZ/WCZ} Read long hub address --PTRA into D. C = MSB of long. * (156)

17.61.2) POPB D *{WC/WZ/WCZ} Read long hub address --PTRB into D. C = MSB of long. (157)

17.62) CALLD D,{#}S Call to S** by writing {C, Z, 10'b0, PC[19:0]} to D. C = S[31], Z = S[30].

CALLD D,{#}S {WC/WZ/WCZ}	Call to S** by writing {C, Z, 10'b0, PC[19:0]} to D. C = S[31], Z = S[30].
-----------------------------	--

CALLD D,{#}S {WC/WZ/WCZ} (158

Call to S** by writing {C, Z, 10'b0, PC[19:0]} to D. C = S[31], Z = S[30].

17.63) RESIO\1\2\3 Resume from Interrupt

RESI3	Resume from INT3. (CALLD \$1F0,\$1F1 WCZ)
RESI2	Resume from INT2. (CALLD \$1F2,\$1F3 WCZ)
RESI1	Resume from INT1. (CALLD \$1F4,\$1F5 WCZ)
RESIO	Resume from INT0. (CALLD \$1FE,\$1FF WCZ)

17.63.1) RESI3 Resume from INT3. (CALLD \$1F0,\$1F1 WCZ) (158)

17.63.2) RESI2 Resume from INT2. (CALLD \$1F2,\$1F3 WCZ) (159)

17.63.3) RESI1 Resume from INT1. (CALLD \$1F4,\$1F5 WCZ) (160)

17.63.4) RESIO Resume from INT0. (CALLD \$1FE,\$1FF WCZ) (161)

17.64) RETIO\1\2\3 Return from Interrupt (162..165)

RETI3	Return from INT3. (CALLD \$1FF,\$1F1 WCZ)
RETI2	Return from INT2. (CALLD \$1FF,\$1F3 WCZ)
RETI1	Return from INT1. (CALLD \$1FF,\$1F5 WCZ)
RETIO	Return from INT0. (CALLD \$1FF,\$1FF WCZ)

17.64.1) RETI3 Return from INT3. (CALLD \$1FF,\$1F1 WCZ) (162)

17.64.2) RETI2 Return from INT2. (CALLD \$1FF,\$1F3 WCZ) (163)

17.64.3) RETI1 Return from INT1. (CALLD \$1FF,\$1F5 WCZ) (164)

17.64.4) RETIO Return from INT0. (CALLD \$1FF,\$1FF WCZ) (165)

17.65) CALLPA\PB Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PA.

CALLPA {#}D,{#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PA.
CALLPB {#}D,{#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PB.

17.65.1) CALLPA {#}D,{#}S Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PA.
(166)

17.65.2) CALLPB {#}D,{#}S Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PB.
(167)

17.66) Decrement and Jump (168..171)

DJZ D,{#}S	Decrement D and jump to S** if result is zero.
DJNZ D,{#}S	Decrement D and jump to S** if result is not zero.
DJF D,{#}S	Decrement D and jump to S** if result is \$FFFF_FFFF.
DJNF D,{#}S	Decrement D and jump to S** if result is not \$FFFF_FFFF.

DJZ D,{#}S Decrement D and jump to S** if result is zero. (168)

DJNZ D,{#}S Decrement D and jump to S** if result is not zero. (169)

DJF D,{#}S Decrement D and jump to S** if result is \$FFFF_FFFF. (170)

DJNF D,{#}S Decrement D and jump to S** if result is not \$FFFF_FFFF. (171)

17.67) Increment and Jump (172..173)

IJZ D,{#}S	Increment D and jump to S** if result is zero.
IJNZ D,{#}S	Increment D and jump to S** if result is not zero.

17.67.1) IJZ D,{#}S Increment D and jump to S** if result is zero. (172)

17.67.2) IJNZ D,{#}S Increment D and jump to S** if result is not zero. (173)

17.68) Test D and Jump (173..179)

TJZ D,{#}S	Test D and jump to S** if D is zero.
TJNZ D,{#}S	Test D and jump to S** if D is not zero.
TJF D,{#}S	Test D and jump to S** if D is full (D = \$FFFF_FFFF).
TJNF D,{#}S	Test D and jump to S** if D is not full (D != \$FFFF_FFFF).
TJS D,{#}S	Test D and jump to S** if D is signed (D[31] = 1).
TJNS D,{#}S	Test D and jump to S** if D is not signed (D[31] = 0).
TJV D,{#}S	Test D and jump to S** if D overflowed (D[31] != C, C = 'correct sign' from last addition/subtraction).

17.68.1) TJZ D,{#}S Test D and jump to S** if D is zero. (174)

17.68.2) TJNZ D,{#}S Test D and jump to S** if D is not zero. (175)

17.68.3) TJF D,{#}S Test D and jump to S** if D is full (D = \$FFFF_FFFF). (176)

17.68.4) TJNF D,{#}S Test D and jump to S** if D is not full (D != \$FFFF_FFFF). (177)

17.68.5) TJS D,{#}S Test D and jump to S** if D is signed (D[31] = 1). (178)

17.68.6) TJNS D,{#}S Test D and jump to S** if D is not signed (D[31] = 0). (179)

17.68.7) TJV D,{#}S Test D and jump to S** if D overflowed (D[31] != C, (180)

C = 'correct sign' from last (addition/subtraction).

17.69) Jump to event flag is set (181..212)

JINT {#}S	Jump to S** if INT event flag is set.
JINT {#}S	Jump to S** if INT event flag is set.
JCT1 {#}S	Jump to S** if CT1 event flag is set.
JCT2 {#}S	Jump to S** if CT2 event flag is set.
JCT3 {#}S	Jump to S** if CT3 event flag is set.
JSE1 {#}S	Jump to S** if SE1 event flag is set.
JSE2 {#}S	Jump to S** if SE2 event flag is set.
JSE3 {#}S	Jump to S** if SE3 event flag is set.
JSE4 {#}S	Jump to S** if SE4 event flag is set.
JPAT {#}S	Jump to S** if PAT event flag is set.
JFBW {#}S	Jump to S** if FBW event flag is set.
JXMT {#}S	Jump to S** if XMT event flag is set.
JXFI {#}S	Jump to S** if XFI event flag is set.
JXRO {#}S	Jump to S** if XRO event flag is set.
JXRL {#}S	Jump to S** if XRL event flag is set.
JATN {#}S	Jump to S** if ATN event flag is set.
JQMT {#}S	Jump to S** if QMT event flag is set.
JNINT {#}S	Jump to S** if INT event flag is clear.
JNCT1 {#}S	Jump to S** if CT1 event flag is clear.
JNCT2 {#}S	Jump to S** if CT2 event flag is clear.
JNCT3 {#}S	Jump to S** if CT3 event flag is clear.
JNSE1 {#}S	Jump to S** if SE1 event flag is clear.
JNSE2 {#}S	Jump to S** if SE2 event flag is clear.

JNSE3 {#}S	Jump to S** if SE3 event flag is clear.
JNSE4 {#}S	Jump to S** if SE4 event flag is clear.
JNPAT {#}S	Jump to S** if PAT event flag is clear.
JNFBW {#}S	Jump to S** if FBW event flag is clear.
JNXMT {#}S	Jump to S** if XMT event flag is clear.
JNXFI {#}S	Jump to S** if XFI event flag is clear.
JNXRO {#}S	Jump to S** if XRO event flag is clear.
JNXRL {#}S	Jump to S** if XRL event flag is clear.
JNATN {#}S	Jump to S** if ATN event flag is clear.
JNQMT {#}S	Jump to S** if QMT event flag is clear.

17.70) Empty Instruction (213..214)

17.71) SETPAT {#}D,{#}S Set Pin Pattern for PAT (pattern) event (215)

SETPAT {#}D,{#}S

Set pin pattern for PAT event. C selects INA/INB, Z selects =/!=, D provides mask value, S provides match value.

17.72) AKIN\WRPIN\WXPIN\WYPI Smart Pin Commands (216..219)

AKPIN {#}S	Acknowledge smart pins S[10:6]+S[5:0]..S[5:0]. Wraps within A/B pins. Prior SETQ overrides S[10:6].
WRPIN {#}D,{#}S	Set mode of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].
WXPIN {#}D,{#}S	Set "X" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].
WYPIN {#}D,{#}S	Set "Y" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].

S is just for specifying which pins are written. D is mode being written.

WRPIN PinField and Pin Definition = S[10:6]+S[5:0]..S[5:0]

Pinfield = 11 bits = %LLLLL_PPPPPP = S[10:6] + S[5:0]

S[5:0] = BASE_PIN = 6 bits = PPPPPP = 0..63

S[10:6] = ADDPINS = 5 bits = LLLLLL = 0..31

Definition of Pin

Pin = 5 bits = %PPPPPP = S[5:0] = 0..63

D is mode being written.

D = %AAAA_BB BB_FF_MMMMMMMMMMMMMM_TT_SSSSS_0

A = PINA input selector

B = PINB input selector

F = PINA and PINB input logic/filtering (after PINA and PINB input selectors)

M = pin mode

T = pin DIR/OUT control (default = %00)

S = smart mode

17.72.1 AKPIN {#}S Acknowledge Smart Pins (216)

Acknowledge smart pins S[10:6]+S[5:0]..S[5:0]. Wraps within A/B pins. Prior SETQ overrides S[10:6].

17.72.2) WRPIN {#}D,{#}S Set mode of smart pins (217)

WRPIN {#}D,{#}S (217)

Set mode of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].

17.72.3) WXPIN {#}D,{#}S Set "X" mode specific Parameter (218)

WXPIN {#}D,{#}S Set "X" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].

17.72.4) WYPIN {#}D,{#}S Set "Y" mode specific Parameter (219)

Set "Y" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].

17.73) WRLU\WRBYTE\WRWORD\WRLONG Write to Destination (220..223)

WRLUT {#}D,{#}S/P	Write D to LUT address {#}S/PTRx.
WRBYTE {#}D,{#}S/P	Write byte in D[7:0] to hub address {#}S/PTRx.
WRWORD {#}D,{#}S/P	Write word in D[15:0] to hub address {#}S/PTRx.
WRLONG {#}D,{#}S/P	Write long in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.

17.74) XZERO\XCONT New Streamer Command (224..225)

XZERO {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, zeroing phase.
XCONT {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, continuing phase.

17.75) RDFAST\WRFAST Fast HUB Read\Write (226..227)

RDFAST {#}D,{#}S	Begin new fast hub read via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.
WRFAST {#}D,{#}S	Begin new fast hub write via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.

17.76) FBLOCK {#}D,{#}S Set next block for when block wraps (228)

FBLOCK {#}D,{#}S	Set next block for when block wraps. D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.
------------------	--

17.77) XINIT\XSTOP\XZERO\XCONT Streamer Commands (229..232)

XINIT {#}D,{#}S	Issue streamer command immediately, zeroing phase.
XSTOP	Stop streamer immediately.
XZERO {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, zeroing phase.
XCONT {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, continuing phase.

17.78) REP {#}D,{#}S Repeat Instruction S Times (233)

REP {#}D,{#}S

Execute next D[8:0] instructions S times. If S = 0, repeat instructions infinitely. If D[8:0] = 0, nothing repeats.

```

_Symbol    REP @Done,S
           First Pasm Instruction to be repeated
           .
           .
           .
           Last Pasm Instruction to be repeated
_Done      PASM instruction to be run after REp

```

The REP instruction needs (the number of ins to repeat)-1 so the calculation is done by the compiler and the result is stored in the rep instruction generated.

Since instruction counting and adjusting is tedious, the @ syntax was to look a bit alike the P1 syntax, consider the pair of labels:

xxx and

xxx _ret

which also create a instruction when you write CALL xxx and the compiler writes a JMPRET instruction for you.

So you can use rep without any label, but it is more tedious.(Note: Not sure how this is done guessing that you can manipulate the count inside D register of REP D,S)

```
rstart rep #rend-#rstart-1,S_REP
```

```
...
```

```
...
```

```
rend ...
```

That it is @ again its just another character to distinguish the operation REP.

The @ ins and the friends @@ and @@@ are usually a Spin-syntax but often used in DAT sections thus also valid for assembler. So in the case of REP the @ has a complete different meaning. @ instructs the compiler to See how many instructions follow the REP instruction to the symbol pointed to with @Symbol.

REP puts a hold on interrupts, and debug is highest level IRQ in the prop2 . So debug won't respond until the REP is completed. The REP instruction is built this way to prevent unexpected branching.

Debug will possibly create a bug because a branch instruction cancels the REP for good. Branching out of a REP is legal, but you need to account for it terminating the REP.

17.78.1_Example_WRD_REP {#}D,{#}S_233

REP {#}D,{#}S

Execute next D[8:0] instructions S times. If S = 0, repeat instructions infinitely. If D[8:0] = 0, nothing repeats.

```
_Start      REP @Symbol,S
            First Pasm Instruction to be repeated
            .
            .
            Last PASM Instruction to be repeated
_Symbol     PASM instruction to be run after REP
```

@Symbolsyntax instructs the compiler to See how many instructions follow the REP instruction to the symbol pointed to with @Symbol this value is then stored in D register.

Example

Demonstrate REP using ADD Instruction and @Symbol Compiler directive

17.79) COGINIT {#}D,{#}S {WC} Start Cog (234)

COGINIT {#}D,{#}S {WC} Start cog selected by D. S[19:0] sets hub startup address and PTRB of cog. Prior SETQ sets PTRB of cog.

Note: This is detailed in "[12.0\) Cog Overview – CogInit\CogStop](#)"

17.80) QMUL\QDIV\QFRAC\QSQRT\QROTATE\QVECTOR Cordic Commands(235..240)

QMUL {#}D,{#}S	Begin CORDIC unsigned multiplication of $D * S$. GETQX/GETQY retrieves lower/upper product.
QDIV {#}D,{#}S	Begin CORDIC unsigned division of {SETQ value or 32'b0, D} / S. GETQX/GETQY retrieves quotient/remainder.
QFRAC {#}D,{#}S	Begin CORDIC unsigned division of {D, SETQ value or 32'b0} / S. GETQX/GETQY retrieves quotient/remainder.
QSQRT {#}D,{#}S	Begin CORDIC square root of {S, D}. GETQX retrieves root.
QROTATE {#}D,{#}S	Begin CORDIC rotation of point (D, SETQ value or 32'b0) by angle S. GETQX/GETQY retrieves X/Y.
QVECTOR {#}D,{#}S	Begin CORDIC vectoring of point (D, S). GETQX/GETQY retrieves length/angle.

17.81) HUBSET {#}D Set Hub configuration (241)

HUBSET {#}D	Set hub configuration to D.
-------------	-----------------------------

Note: For Details see [“Appendix I Hub Operation”](#)

17.82) COGID {#}D {WC} Get Cog ID (0..15) (242..247)

COGID {#}D {WC}	If D is register and no WC, get cog ID (0 to 15) into D. If WC, check status of cog D[3:0], C = 1 if on.
COGSTOP {#}D	Stop cog D[3:0].
LOCKNEW D {WC}	Request a LOCK. D will be written with the LOCK number (0 to 15). C = 1 if no LOCK available.
LOCKRET {#}D	Return LOCK D[3:0] for reallocation.
LOCKTRY {#}D {WC}	Try to get LOCK D[3:0]. C = 1 if got LOCK. LOCKREL releases LOCK. LOCK is also released if owner cog stops or restarts.
LOCKREL {#}D {WC}	Release LOCK D[3:0]. If D is a register and WC, get current/last cog id of LOCK owner into D and LOCK status into C.

17.83) QLOG\QEXP Cordic Conversion (248..249)

QLOG {#}D	Begin CORDIC number-to-logarithm conversion of D. GETQX retrieves log {5'whole_exponent, 27'fractional_exponent}.
QEXP {#}D	Begin CORDIC logarithm-to-number conversion of D. GETQX retrieves number.

17.84) RFBYTE\RFWORD|RFLONG|RFVAR|RFVARS\WFBYTE\WFWORD\WFLONG(250..257)

RFBYTE D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended byte from FIFO into D. C = MSB of byte. *
RFWORD D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended word from FIFO into D. C = MSB of word. *
RFLONG D {WC/WZ/WCZ}	Used after RDFAST. Read long from FIFO into D. C = MSB of long. *
RFVAR D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended 1..4-byte value from FIFO into D. C = 0. *
RFVARS D {WC/WZ/WCZ}	Used after RDFAST. Read sign-extended 1..4-byte value from FIFO into D. C = MSB of value. *
WFBYTE {#}D	Used after WRFAST. Write byte in D[7:0] into FIFO.
WFWORD {#}D	Used after WRFAST. Write word in D[15:0] into FIFO.
WFLONG {#}D	Used after WRFAST. Write long in D[31:0] into FIFO.

17.85) GETQX\GETQY\GETCT\GETRND D\GETRND Get Result Value (258..262)

GETQX D {WC/WZ/WCZ}	Retrieve CORDIC result X into D. Waits, in case result not ready. C = X[31]. *
GETQY D {WC/WZ/WCZ}	Retrieve CORDIC result Y into D. Waits, in case result not ready. C = Y[31]. *
GETCT D {WC}	Get CT[31:0] or CT[63:32] if WC into D. GETCT WC + GETCT gets full CT. CT=0 on reset, CT++ on every clock. C = same.
GETRND D {WC/WZ/WCZ}	Get RND into D/C/Z. RND is the PRNG that updates on every clock. D = RND[31:0], C = RND[31], Z = RND[30], unique per cog.
GETRND WC/WZ/WCZ	Get RND into C/Z. C = RND[31], Z = RND[30], unique per cog.

17.86) SETDACS\SETXFRQ Set DAC Set Streamer NCO frequency(263..264)

SETDACS {#}D	DAC3 = D[31:24], DAC2 = D[23:16], DAC1 = D[15:8], DAC0 = D[7:0].
SETXFRQ {#}D	Set streamer NCO frequency to D.

17.87) GETXACC D Get the streamer's Goertzel X accumulator (265)

GETXACC D	Get the streamer's Goertzel X accumulator into D and the Y accumulator into the next instruction's S, clear accumulators.
-----------	---

17.88) WAITX {#}D {WC/WZ/WCZ} Wait Clock Cycles (266)

WAITX {#}D {WC/WZ/WCZ} C/Z = 0.	Wait 2 + D clocks if no WC/WZ/WCZ. If WC/WZ/WCZ, wait 2 + (D & RND) clocks.
---------------------------------	---

17.89) SETSE1\SETSE2\SETSE3\SETSE4 Set SEx Event Configuration (266..269)

SETSE1 {#}D	Set SE1 event configuration to D[8:0].
SETSE2 {#}D	Set SE2 event configuration to D[8:0].
SETSE3 {#}D	Set SE3 event configuration to D[8:0].
SETSE4 {#}D	Set SE4 event configuration to D[8:0].

17.90) POLLx Poll Event x Flag (271..286)

POLLINT {WC/WZ/WCZ}	Get INT event flag into C/Z, then clear it.
POLLCT1 {WC/WZ/WCZ}	Get CT1 event flag into C/Z, then clear it.
POLLCT2 {WC/WZ/WCZ}	Get CT2 event flag into C/Z, then clear it.
POLLCT3 {WC/WZ/WCZ}	Get CT3 event flag into C/Z, then clear it.
POLLSE1 {WC/WZ/WCZ}	Get SE1 event flag into C/Z, then clear it.
POLLSE2 {WC/WZ/WCZ}	Get SE2 event flag into C/Z, then clear it.
POLLSE3 {WC/WZ/WCZ}	Get SE3 event flag into C/Z, then clear it.
POLLSE4 {WC/WZ/WCZ}	Get SE4 event flag into C/Z, then clear it.
POLLPAT {WC/WZ/WCZ}	Get PAT event flag into C/Z, then clear it.
POLLFBW {WC/WZ/WCZ}	Get FBW event flag into C/Z, then clear it.
POLLXMT {WC/WZ/WCZ}	Get XMT event flag into C/Z, then clear it.
POLLXFI {WC/WZ/WCZ}	Get XFI event flag into C/Z, then clear it.
POLLXRO {WC/WZ/WCZ}	Get XRO event flag into C/Z, then clear it.
POLLXRL {WC/WZ/WCZ}	Get XRL event flag into C/Z, then clear it.
POLLATN {WC/WZ/WCZ}	Get ATN event flag into C/Z, then clear it.
POLLQMT {WC/WZ/WCZ}	Get QMT event flag into C/Z, then clear it.

17.90.1) POLLINT {WC/WZ/WCZ} Get INT event flag into C/Z, then clear it.

17.90.2) POLLCT1/WAITCT1 Get CT1 event flag into C/Z, then clear it. 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.

17.90.3) POLLCT2/WAITCT2 Get CT2 event flag into C/Z, then clear it.
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.

17.90.4) POLLCT3/WAITCT3 Get CT3 event flag into C/Z, then clear it.
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.

Note: For Detailed discussion see [17.58\) ADDCT1\ADDCT2\ADDCT3](#)

17.91) WAITx Wait until x event flag is set (287..301)

WAITINT {WC/WZ/WCZ}	Wait for INT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITCT1 {WC/WZ/WCZ}	Wait for CT1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITCT2 {WC/WZ/WCZ}	Wait for CT2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITCT3 {WC/WZ/WCZ}	Wait for CT3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITSE1 {WC/WZ/WCZ}	Wait for SE1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITSE2 {WC/WZ/WCZ}	Wait for SE2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITSE3 {WC/WZ/WCZ}	Wait for SE3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITSE4 {WC/WZ/WCZ}	Wait for SE4 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITPAT {WC/WZ/WCZ}	Wait for PAT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITFBW {WC/WZ/WCZ}	Wait for FBW event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITXMT {WC/WZ/WCZ}	Wait for XMT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITXFI {WC/WZ/WCZ}	Wait for XFI event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITXRO {WC/WZ/WCZ}	Wait for XRO event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITXRL {WC/WZ/WCZ}	Wait for XRL event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
WAITATN {WC/WZ/WCZ}	Wait for ATN event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.

17.91.1) WAITINT {WC/WZ/WCZ} event INT flag (287)

Wait for INT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.

17.91.2) POLLCT1/WAITCT1 event CT1 flag (288)

Wait for CT1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
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.

Note: For Detailed discussion see [17.58\) ADDCT1\ADDCT2\ADDCT3](#)

17.91.3) POLLCT2/WAITCT2 event CT2 flag (289)

Wait for CT2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
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.

Note: For Detailed discussion see [17.58\) ADDCT1\ADDCT2\ADDCT3](#)

17.91.4) POLLCT3/WAITCT3 event CT3 flag (290)

Wait for CT3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
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.

Note: For Detailed discussion see [17.58\) ADDCT1\ADDCT2\ADDCT3](#)

17.92) Interrupt Instructions (302..312)

302	ALLOWI	Allow interrupts (default).
303	STALLI	Stall Interrupts.
304	TRGINT1	Trigger INT1, regardless of STALLI mode.
305	TRGINT2	Trigger INT2, regardless of STALLI mode.
306	TRGINT3	Trigger INT3, regardless of STALLI mode.
307	NIXINT1	Cancel INT1.
308	NIXINT2	Cancel INT2.
309	NIXINT3	Cancel INT3.
310	SETINT1 {#}D	Set INT1 source to D[3:0].
311	SETINT2 {#}D	Set INT2 source to D[3:0].
312	SETINT3 {#}D	Set INT3 source to D[3:0].

17.93) SETQ SetQ register prior to Instruction (313..314)

SETQ {#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set block transfer. Also used before MUXQ/COGINIT/QDIV/QFRAC/QROTATE/WAITxxx.
SETQ2 {#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set LUT block transfer.

17.93) SETQ CONSIDERATIONS (313..314))

Q is a hidden special purpose register inside the cog's processor core (ALU). The Program Counter (PC) is another one of these. Q also must have a couple of associated flags to tell subsequent instructions that Q has just been refreshed. At least two flags are needed for RDLONG/WRLONG to know if they should burst read/write to cogRAM or lutRAM. SETQ sets first flag and SETQ2 sets the second flag.

- TeluXORO32 executes - Q is set to the XORO32 result.
- RDLUT executes - Q is set to the data read from the lookup RAM.
- GETXACC executes - Q is set to the Goertzel sine accumulator value.
- CRCNIB executes - Q gets shifted left by four bits.
- COGINIT/QDIV/QFRAC/QROTATE executes without a preceding SETQ instruction - Q is set to zero.

Note: Following is what is guestimate for SETQ

SETQ/SETQ2 shields the next instruction from interruption to prevent an interrupt service routine.

A-SETQ works differently to the ALTx instructions. SETQ fills the Q register and sets a flag as future notification. ALTx instructions modify the already fetched next instruction inside the pipeline.

B- will have logic to detect the notification flag and change its data source for S[9:5] to Q[?:?:?]. Best guess is Q[4:0].

17.94) PUSH\POP\JMP Stack Instructions (315..317)

PUSH {#}D	Push D onto stack.
POP D {WC/WZ/WCZ}	Pop stack (K). D = K. C = K[31]. *
JMP D {WC/WZ/WCZ}	Jump to D. C = D[31], Z = D[30], PC = D[19:0].

17.95) CALL\RET Call subroutine using stack (318..319)

CALL D {WC/WZ/WCZ}	Call to D by pushing {C, Z, 10'b0, PC[19:0]} onto stack. C = D[31], Z = D[30], PC = D[19:0].
RET {WC/WZ/WCZ}	Return by popping stack (K). C = K[31], Z = K[30], PC = K[19:0].

17.96) CALLA\RETA Call using PTR A (320..321)

CALLA D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR A++. C = D[31], Z = D[30], PC = D[19:0].
RETA {WC/WZ/WCZ}	Return by reading hub long (L) at --PTR A. C = L[31], Z = L[30], PC = L[19:0].

RET <inst> <ops>	Execute <inst> always and return if no branch. If <inst> is not branching then return by popping stack[19:0] into PC.
--------------------	---

17.97) CALLB\RETB Call using PTR B (322..323)

CALLB D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR B++. C = D[31], Z = D[30], PC = D[19:0].
RETB {WC/WZ/WCZ}	Return by reading hub long (L) at --PTR B. C = L[31], Z = L[30], PC = L[19:0].

RET <inst> <ops>	Execute <inst> always and return if no branch. If <inst> is not branching then return by popping stack[19:0] into PC.
--------------------	---

17.98) JMPREL Jump ahead/back by D instructions (324)

JMPREL {#}D

Jump ahead/back by D instructions. For cogex, $PC += D[19:0]$. For hubex, $PC += D[17:0] \ll 2$.

17.99) SKIP\SKIPF Skip Instructions (325..326)

SKIP {#}D

Skip instructions per D. Subsequent instructions 0..31 get cancelled for each '1' bit in $D[0]..D[31]$.

SKIPF {#}D

Skip cog/LUT instructions fast per D. Like SKIP, but instead of cancelling instructions, the PC leaps over them.

17.100) EXECF Jump to D[9:0] in cog/LUT and set SKIPF pattern to D[31:10] (327)

EXECF {#}D

Jump to $D[9:0]$ in cog/LUT and set SKIPF pattern to $D[31:10]$. $PC = \{10'b0, D[9:0]\}$.

17.101) GERPTR D Get current FIFO hub pointer into D. (328)

GETPTR D

Get current FIFO hub pointer into D.

17.102) GETBRK Get breakpoint/cog status into D according to WC/WZ/WCZ. (329)

GETBRK D
WC/WZ/WCZ

Get breakpoint/cog status into D according to WC/WZ/WCZ. See documentation for details.

17.103) COGBRK If in debug ISR, trigger asynchronous breakpoint in cog (330)

COGBRK {#}D

If in debug ISR, trigger asynchronous breakpoint in cog $D[3:0]$. Cog $D[3:0]$ must have asynchronous breakpoint enabled.

17.104) BRK If in debug ISR (331)

BRK {#}D

If in debug ISR, set next break condition to D. Else, set BRK code to $D[7:0]$ and unconditionally trigger BRK interrupt, if enabled.

17.105) SETLUTS If $D[0] = 1$ then enable LUT sharing (332)

SETLUTS {#}D

If $D[0] = 1$ then enable LUT sharing, where LUT writes within the adjacent odd/even companion cog are copied to this cog's LUT.

17.106) SETCYx Set the colorspace converter (332..338)

SETCY {#}D	Set the colorspace converter "CY" parameter to D[31:0].
SETCI {#}D	Set the colorspace converter "CI" parameter to D[31:0].
SETCQ {#}D	Set the colorspace converter "CQ" parameter to D[31:0].
SETCFRQ {#}D	Set the colorspace converter "CFRQ" parameter to D[31:0].
SETCMOD {#}D	Set the colorspace converter "CMOD" parameter to D[8:0].
SETPIV {#}D	Set BLNPIX/MIXPIX blend factor to D[7:0].
SETPIX {#}D	Set MIXPIX mode to D[5:0].

17.107) COGATN Strobe "attention" of all cogs whose corresponding bits are high (340)

COGATN {#}D

Strobe "attention" of all cogs whose corresponding bits are high in D[15:0].

17.108) TESTP Test IN of pin (341..348)

TESTP {#}D WC/WZ	Test IN bit of pin D[5:0], write to C/Z. $C/Z = IN[D[5:0]]$.
TESTPN {#}D WC/WZ	Test !IN bit of pin D[5:0], write to C/Z. $C/Z = !IN[D[5:0]]$.
TESTP {#}D ANDC/ANDZ	Test IN bit of pin D[5:0], AND into C/Z. $C/Z = C/Z \text{ AND } IN[D[5:0]]$.
TESTPN {#}D ANDC/ANDZ	Test !IN bit of pin D[5:0], AND into C/Z. $C/Z = C/Z \text{ AND } !IN[D[5:0]]$.
TESTP {#}D ORC/ORZ	Test IN bit of pin D[5:0], OR into C/Z. $C/Z = C/Z \text{ OR } IN[D[5:0]]$.
TESTPN {#}D ORC/ORZ	Test !IN bit of pin D[5:0], OR into C/Z. $C/Z = C/Z \text{ OR } !IN[D[5:0]]$.
TESTP {#}D XORC/XORZ	Test IN bit of pin D[5:0], XOR into C/Z. $C/Z = C/Z \text{ XOR } IN[D[5:0]]$.
TESTPN {#}D XORC/XORZ	Test !IN bit of pin D[5:0], XOR into C/Z. $C/Z = C/Z \text{ XOR } !IN[D[5:0]]$.

17.109) DIR Direction Pin bits Instruction (349..357)

DIRL {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = 0. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRH {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = 1. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRC {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRNC {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = !C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRZ {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRNZ {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRRND {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
DIRNOT {#}D {WCZ}	Toggle DIR bits of pins D[10:6]+D[5:0]..D[5:0]. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.

17.110) OUTx Instruction (358..364)

OUTL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
OUTNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.

17.111) FLTx Out Bits of pins (365..372)

FLTL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
FLTNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.

17.112) DRVx OUT bits of pins(373..380)

DRVL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVNRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
DRVNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.

17.113) SPLIT\MERGE\SPLITW\SEUSSF\SEUSSR\RGBSQZ\RGBEXP (381..389)

SPLITB D	Split every 4th bit of D into bytes. $D = \{D[31], D[27], D[23], D[19], \dots D[12], D[8], D[4], D[0]\}$.
MERGE B D	Merge bits of bytes in D. $D = \{D[31], D[23], D[15], D[7], \dots D[24], D[16], D[8], D[0]\}$.
SPLITW D	Split odd/even bits of D into words. $D = \{D[31], D[29], D[27], D[25], \dots D[6], D[4], D[2], D[0]\}$.
MERGEW D	Merge bits of words in D. $D = \{D[31], D[15], D[30], D[14], \dots D[17], D[1], D[16], D[0]\}$.
SEUSSF D	Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Forward pattern.
SEUSSR D	Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Reverse pattern.
RGBSQZ D	Squeeze 8:8:8 RGB value in $D[31:8]$ into 5:6:5 value in $D[15:0]$. $D = \{15'b0, D[31:27], D[23:18], D[15:11]\}$.
RGBEXP D	Expand 5:6:5 RGB value in $D[15:0]$ into 8:8:8 value in $D[31:8]$. $D = \{D[15:11, 15:13], D[10:5, 10:9], D[4:0, 4:2], 8'b0\}$.
SPLITB D	Split every 4th bit of D into bytes. $D = \{D[31], D[27], D[23], D[19], \dots D[12], D[8], D[4], D[0]\}$.

17.114) 390 REV D Reverse D Bits (390)

REV D	Reverse D bits. $D = D[0:31]$.
-------	---------------------------------

17.114.1_Example_WRD_REV D_390

REV D Reverse D Bits. $D = D[0:31]$

17.115) RCZR\RCZL Rotate C,Z right through D. (391..392)

RCZR D {WC/WZ/WCZ}	Rotate C,Z right through D. $D = \{C, Z, D[31:2]\}$. $C = D[1]$, $Z = D[0]$.
--------------------	---

RCZL D {WC/WZ/WCZ}	Rotate C,Z left through D. $D = \{D[29:0], C, Z\}$. $C = D[31]$, $Z = D[30]$.
--------------------	--

17.116) WRC\WRNC\WRZ\WRNZ Write 0 or 1 according C\Z to D (393..396)

WRC D	Write 0 or 1 to D, according to C. D = {31'b0, C}.
WRNC D	Write 0 or 1 to D, according to !C. D = {31'b0, !C}.
WRZ D	Write 0 or 1 to D, according to Z. D = {31'b0, Z}.
WRNZ D	Write 0 or 1 to D, according to !Z. D = {31'b0, !Z}.

17.117) MODCZ\MODC\MODZ Modify C and Z according to cccc\zzzz (397..399)

MODCZ c,z {WC/WZ/WCZ}	Modify C and Z according to cccc and zzzz. C = cccc[{C,Z}], Z = zzzz[{C,Z}].
MODC c {WC}	Modify C according to cccc. C = cccc[{C,Z}].
MODZ z {WZ}	Modify Z according to zzzz. Z = zzzz[{C,Z}].

17.117) MODCZ Modify C or Z Flag

MODCZ c,z {WC/WZ/WCZ}	Math and Logic	EEEE 1101011 CZ1 0cccczzzz 001101111
MODC c {WC}	Math and Logic	EEEE 1101011 C01 0cccc0000 001101111
MODZ z {WZ}	Math and Logic	EEEE 1101011 0Z1 00000zzzz 001101111

MODCZ c,z {WC/WZ/WCZ} Modify C and Z according to cccc and zzzz. C = cccc[{C,Z}], Z = zzzz[{C,Z}].

MODC c {WC} Modify C according to cccc. C = cccc[{C,Z}].

MODZ z {WZ} Modify Z according to zzzz. Z = zzzz[{C,Z}].

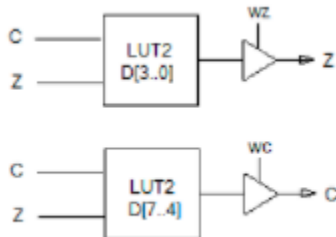
Operand MODCZ/MODC/MODZ

cccc and zzzz are the 4 bits that the constants define. They will be put into the D and S field.

MODCZ/MODC/MODZ is a "D-only" instruction, where D[7:4] = cccc and D[3:0] = zzzz.

S field is fixed and part of the opcode. D_xxxxxxx_xxxxxxxx_xxxxxxxx_CCCCC_ZZZZZ

Note: x indicates unknown



The block diagram is always the same for MODCZ/MODC/MODZ. You can replace D[3..0] by zzzz and D[7..4] by cccc, to match the bit names in the instruction encoding. You can see these LUTs as a 1bit wide ROM with 2 address inputs = 4 bits total. These address inputs are connected to the current state of C and Z. Depending on the state of C and Z one of 4 bits in the ROM is read and defines the new state of the C or Z flag, if the WC and/or WZ effect is set.

The instruction is very universal, but therefore also a bit complex. Like in the LUTs of an FPGA, you define in a truth table the resulting bit for any combination of C and Z.

Say you want to set Z to the state of C, then the truth table looks like that:

C	Z	zzzz
0	0	0

0	1		0
1	0		1
1	1		1

'-> %1100 = zzzz

when C is 1 the result is 1, if C is 0 the result is 0

so you can write the MODx instruction like that: `MODZ %1100 WZ` 'sets Z to C

but it is easier to understand with the constant: `MODZ _C WZ` 'the same with the named constant

MODCZ lets you affect the C and the Z flag in one instruction, you can for example swap the two flags:

MODCZ	<u> </u> Z,	<u> </u> C	WCZ	
	v	v		
	C	Z		'swap c and z

From the instruction encoding, that all 3 instructions are the same. Unused bits are just set to zero for MODC and MODZ. You can also affect only C or Z with MODCZ, but then the assembly syntax requires a dummy argument for the not used flag.

M.1) MODCZ constants From the instructions_v32.txt:

```

_CLR          =    %0000
_NC_AND_NZ    =    %0001
_NZ_AND_NC    =    %0001
_GT           =    %0001
_NC_AND_Z     =    %0010
_Z_AND_NC     =    %0010
_NC           =    %0011
_GE           =    %0011
_C_AND_NZ     =    %0100
_NZ_AND_C     =    %0100
_NZ           =    %0101
_NE           =    %0101
_C_NE_Z       =    %0110
_Z_NE_C       =    %0110
_NC_OR_NZ     =    %0111
_NZ_OR_NC     =    %0111
_C_AND_Z      =    %1000
_Z_AND_C      =    %1000
_C_EQ_Z       =    %1001
_Z_EQ_C       =    %1001
_Z            =    %1010
_E            =    %1010
_NC_OR_Z      =    %1011
_Z_OR_NC      =    %1011
_C            =    %1100
_LT           =    %1100
_C_OR_NZ      =    %1101
_NZ_OR_C      =    %1101
_C_OR_Z       =    %1110
_Z_OR_C       =    %1110
_LE           =    %1110
_SET          =    %1111

```

Examples:

```

MODCZ _CLR, _Z_OR_C W CZ  'C = 0, Z |= C
MODCZ _NZ, 0      WC  'C = !Z
MODCZ 0, _SET     WZ  'Z = 1
MODC  _NZ_AND_C   WC  'C = !Z & C
MODZ  _Z_NE_C     WZ  'Z = Z ^ C

```

17.117.1_Example_WRD_MODCZ_Operand

cccc and zzzz are the 4 bits that the constants define. They will be put into the D and S field.

MODCZ/MODC/MODZ is a "D-only" instruction, where D[7:4] = cccc and D[3:0] = zzzz.

S field is fixed and part of the opcode. D_XXXXXXXX_XXXXXXXX_XXXXXXXX_CCCCC_ZZZZZ

Note: x indicates unknown MODCZ useful for setting FFlag Bits CZ

17.118) SETSP\GETSCP Set four channel Scope (400..401)

SETSCP {#}D	Set four-channel oscilloscope enable to D[6] and set input pin base to D[5:2].
GETSCP D	Get four-channel oscilloscope samples into D. D = {ch3[7:0],ch2[7:0],ch1[7:0],ch0[7:0]}.

17.119) JMP\CALL\CALLA\CALLB\CALLD Jump or Call (402..406)

JMP #\}A	Jump to A. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
CALL #\}A	Call to A by pushing {C, Z, 10'b0, PC[19:0]} onto stack. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
CALLA #\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTRB++. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
CALLB #\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTRB++. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
CALLD PA/PB/PTRA/PTRB,#\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to PA/PB/PTRA/PTRB (per W). If R = 1 then PC += A, else PC = A. "\" forces R = 0.

17.120) LOC PA/PB/PTRA/PTRB,#{\}A (407)

LOC PA/PB/PTRA/PTRB,#{\}A

Get {12'b0, address[19:0]} into PA/PB/PTRA/PTRB (per W).
If R = 1, address = PC + A, else address = A. "\" forces R = 0.

17.121. AUGS\AUGD #N (408..409)

AUGS #n	Queue #n to be used as upper 23 bits for next #S occurrence, so that the next 9-bit #S will be augmented to 32 bits.
AUGD #n	Queue #n to be used as upper 23 bits for next #D occurrence, so that the next 9-bit #D will be augmented to 32 bits.

AUGS #n

Queue #n to be used as upper 23 bits for next #S occurrence, so that the next 9-bit #S will be augmented to 32 bits.

AUGD #n

Queue #n to be used as upper 23 bits for next #D occurrence, so that the next 9-bit #D will be augmented to 32 bits.

For purpose of this discussion the “Queue” will be termed 32 bit register with AUGSx for AUGS instruction and AUGDx for AUGD instruction.

Rules for Implementing AUGS\AUGD:

- ALTx/SCA/XORO32 instructions have to be hard prefixed. They can only operate on the very next instruction.
- AUGx normally affects the next instruction but will automatically extend over other AUGx and ALTx.
- SETQ normally affects the next instruction but will automatically extend over other AUGx and ALTx.
- Interrupts are postponed during a SETQ or AUGx or ALTx ... any instruction prefixing. Notably also for the duration of a REP loop and all WAITx instructions.

AUGS\AUGD has a Propeller IDE special directive ## which can be used to augment PASM instruction destination and source fields to expand the 9 bit (0-511) to 32 bit field.

Example

WrpIN ##P_LOW_15K,btN (WRPIN D/#,S/#) contains the spin compiler directive ## which results in the generation of code using the AUGD instruction to be used to expand the 9 bit D field to 32 bits. AUGD is selected because ## appears in the D field.

WRPIN ##P_LOW_15K,btN

Expands to:

AUGD #(P_LOW_15K>>9) ‘shift 9 bits leaving 23 upper bits

WRPIN #(P_LOW_15K & \$1FF),btN ‘mask off upper bits leaving 9 lower bits

The Destination field D is modified to create a 32 bit value adding the AUGD 23 bits with the WRPIN 9 bits and In the example (WRPIN) would be a 32 pin mode that is being written to the Pin(or Pins) defined by “btN”.

{=====}	
CON {Application Constants}	'user defined constants
VAR {Application Variables}	'user defined variables
OBJ {Application Objects}	'user defined objects
DAT {Application Data}	'user defined data
PUB mainApp()	'user Spin Program
repeat	'keep cog 0 running
DAT {Application PASM}	'user PASM Program

Appendix “B” DEBUG INTERRUPT

In addition to the 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.

Debug interrupts are enabled on a per-cog basis via HUBSET. Each debug-enabled cog will generate a debug interrupt on (re)start from each COGINIT exercised upon it. Within that initial debug ISR and within each subsequent debug ISR, multiple trigger conditions may be set for the next debug interrupt. If no trigger conditions are set before the debug ISR ends, no more debug interrupts will occur until the cog is restarted from another COGINIT.

The last 16KB of hub RAM, which is also mapped to \$FC000..\$FFFFFF, gets partially used as a buffer area for saving and restoring cog registers during debug ISR's. The initial debug ISR routines are also stored in this upper RAM. Once initialized with debug ISR code, this upper hub RAM can be write-protected, in which case it is mapped only to \$FC000..\$FFFFFF and it is only writable from within debug ISR's.

Each cog has an execute-only ROM in cog registers \$1F8..\$1FF which contains special debug-ISR-entry and -exit routines. These tiny routines perform seamless register-load and register-restore operations for your debugger program, which must be realized entirely within debug ISR's.

Execute-only ROM in cog registers \$1F8..\$1FF (%cccc = !CogNumber)
Debug ISR Entry - IJMP0 is initialized to \$1F8 on cog start
\$1F8 - SETQ #\$0F 'save registers \$000..\$00F \$1F9 - WRLONG 0,* '* = %1111_1111_1ccc_c000_0000 \$1FA - SETQ #\$0F 'load program into \$000..\$00F \$1FB - RDLONG 0,* '* = %1111_1111_1ccc_c100_0000

\$1FC - JMP #0 'jump to loaded program
Debug ISR Exit - Jump here to exit your debug ISR
\$1FD - SETQ #0 'restore registers \$000..\$00F
\$1FE - RDLONG 0,* '* = %1111_1111_1ccc_c000_0000
\$1FF - RETIO 'CALLD IRET0,IRET0 WCZ

During a debug ISR, INA and INB, normally read-only input-pin registers, become readable/writable RAM registers named IJMP0 and IRET0, and are used by the debug interrupt as jump and return addresses. On COGINIT, IJMP0 is initialized to \$1F8 which is the debug-ISR-entry routine's address.

When a debug interrupt occurs with IJMP0 pointing to \$1F8, the following sequence happens:

Cog registers \$000 to \$00F are saved to hub RAM starting at (\$FF800 + !CogNumber << 7), or %1111_1111_1ccc_c000_0000, where %cccc = !CogNumber.

Cog registers \$000 to \$00F are loaded from hub RAM starting at (\$FF840 + !CogNumber << 7), or %1111_1111_1ccc_c100_0000, where %cccc = !CogNumber.

A "JMP #\$000" executes to run the 16-instruction debugger program that was just loaded into registers \$000 to \$00F.

Your 16-instruction debugger program will likely want to determine if this debug interrupt was due to a COGINIT, in which case the debugger will probably want to note that a new program is now running in this cog. Depending on what the debugger must do next, it is likely that it will need to save more registers to the upper hub RAM and then load in more code from the upper hub RAM to facilitate more complex operations than the initial 16-instruction ISR can achieve. The ISR may then need to perform some communication between itself and a host system which may be serving as the debugger's user interface. It may be necessary to employ a LOCK to time-share P2-to-host communication channels among cogs, likely on P63 (serial Rx) and P62 (serial Tx). This scenario is somewhat hypothetical, but illustrates the design intent behind the debug interrupt mechanism.

When your debug ISR is complete, you can do a 'JMP #\$1FD' to execute the debug-ISR-exit routine which does the following:

Original cog registers \$000 to \$00F are restored from hub RAM starting at (\$FF800 + !CogNumber << 7), or %1111_1111_1ccc_c000_0000, where %cccc = !CogNumber.

A "RETIO" executes to return to the interrupted cog program.

Here is a table of the hub RAM locations used by each cog for register save/restore and ISR images during the debug interrupt when the register ROM routines are used for ISR entry and exit:

Cog	Save/Restore in Hub RAM for Registers \$000..\$00F	ISR image in Hub RAM for Registers \$000..\$00F
7	\$FFC00..\$FFC3F	\$FFC40..\$FFC7F
6	\$FFC80..\$FFCBF	\$FFCC0..\$FFCFF
5	\$FFD00..\$FFD3F	\$FFD40..\$FFD7F
4	\$FFD80..\$FFDBF	\$FFDC0..\$FFDFF
3	\$FFE00..\$FFE3F	\$FFE40..\$FFE7F
2	\$FFE80..\$FFEBF	\$FFEC0..\$FFEFF
1	\$FFF00..\$FFF3F	\$FFF40..\$FFF7F
0	\$FFF80..\$FFFBF	\$FFFC0..\$FFFFF

Though the first debug interrupt upon cog (re)start will always use the debug-ISR-entry routine at \$1F8, you may redirect IJMP0 during any debug ISR to point elsewhere for use by subsequent debug interrupts. This would mean that you would lose the initial register-saving function provided by the small ROM starting at \$1F8, so you would have to use some cog registers for debugger-state storage that don't interfere with the cog program that is being debugged. If no register saving/restoring or host communications are required, your debug ISR may execute very quickly.

What terminates a debug interrupt is not only RETI0 (CALLD INB,INB WCZ), but any D-register variant (CALLD anyreg,INB WCZ). For example RESI0 (CALLD INA,INB WCZ) may be used to resume next time from where this debug ISR left off, but this would imply that you are not using the debug-ISR-entry and -exit routines in the cog-register ROM and have, instead, permanently located debugger code into some cog registers, so that your debugger program is already present at the start of the debug interrupt.

This debug interrupt scheme was designed to operate stealthily, without any cooperation from the cog program being debugged. All control has been placed within the debug ISR. 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 the debug interrupt compromised as a standard debugging mechanism. Also, by executing the ISR strictly in cog register space, this scheme does not interfere with the hub FIFO state, which would be impossible to reconstruct if disturbed by hub execution within the debug ISR.

Below are the instructions which are used in the debugging mechanism:

BRK D/#

During normal program execution, the BRK instruction is used to generate a debug interrupt with an 8-bit code which can be read within the debug ISR. The BRK instruction interrupt must be enabled from within a prior debug ISR for this to work. Regardless of the execution condition, the BRK instruction will trigger a debug interrupt, if enabled. The execution condition only gates the writing of the 8-bit code:

D/# = %BBBBBBBB: 8-bit BRK code

During a debug ISR, the BRK instruction operates differently and is used to establish the next debug interrupt condition(s). It is also used to select INA/INB, instead of the IJMP0/IRET0 registers exposed during the ISR, so that the pins' inputs states may be read:

D/# = %aaaaaaaaaaaaaaaaeeee_LKJIHGFEDCBA

%aaaaaaaaaaaaaaaaeeee: 20-bit breakpoint address or 4-bit event code (%eeee)

%L: 1 = map INA/INB normally, 0 = map IJMP0/IRET0 at INA/INB (default during ISR) *

%K: 1 = enable interrupt on breakpoint address match

%J: 1 = enable interrupt on event %eeee

%I: 1 = enable interrupt on asynchronous breakpoint (via COGBRK on another cog)

%H: 1 = enable interrupt on INT3 ISR entry

%G: 1 = enable interrupt on INT2 ISR entry

%F: 1 = enable interrupt on INT1 ISR entry

%E: 1 = enable interrupt on BRK instruction

%D: 1 = enable interrupt on INT3 ISR code (single step)

%C: 1 = enable interrupt on INT2 ISR code (single step)

%B: 1 = enable interrupt on INT1 ISR code (single step)

%A: 1 = enable interrupt on non-ISR code (single step)

* If set to 1 by the debug ISR, %L must be reset to 0 before exiting the debug ISR, so that the RETI0 instruction is able to see IJMP0 and IRET0.

On debug ISR entry, bits A to L, are cleared to '0'. If a subsequent debug interrupt is desired, a BRK instruction must be executed before exiting the debug ISR, in order to establish the next breakpoint condition(s).

COGBRK D/#

The COGBRK instruction can trigger an asynchronous breakpoint in another cog. For this to work, the cog executing the COGBRK instruction must be in its own debug ISR and the other cog must have its asynchronous breakpoint interrupt enabled:

D/# = %CCCC: the cog in which to trigger an asynchronous breakpoint

GETBRK D WCZ

During normal program execution, GETBRK with WCZ returns various data about the cog's internal status:

C = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)

Z = 1 if cog started in hubexec or 0 if cog started in cogexec

D[31:23] = 0

D[22] = 1 if colorspace converter is active

D[21] = 1 if streamer is active

D[20] = 1 if WRFAST mode or 0 if RDFAST mode

D[19:16] = INT3 selector, established by SETINT3

D[15:12] = INT2 selector, established by SETINT2

D[11:08] = INT1 selector, established by SETINT1

D[07:06] = INT3 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[05:04] = INT2 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[03:02] = INT1 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[01] = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)

D[00] = 1 if cog started in hubexec or 0 if cog started in cogexec

During a debug ISR, GETBRK with WCZ returns additional data that is useful to a debugger:

C = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started

D[31:24] = 8-bit break code from the last 'BRK #/D' during normal execution

D[23] = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started

GETBRK D WC

GETBRK with WC always returns the following:

C = LSB of SKIP/SKIPF/EXECF/XBYTE pattern

D[31:28] = 4-bit CALL depth since SKIP/SKIPF/EXECF/XBYTE (skipping suspended if not %0000)

D[27] = 1 if SKIP mode or 0 if SKIPF/EXECF/XBYTE mode

D[26] = 1 if LUT sharing enabled (established by SETLUTS)

D[25] = 1 if top of stack = \$001FF, indicating XBYTE will execute on next _RET_/RET

D[24:16] = 9-bit XBYTE mode, established by '_RET_ SETQ/SETQ2' when top of stack = \$001FF

D[15:00] = 16 event-trap flags

GETBRK D WZ

GETBRK with WZ always returns the following:

Z = 1 if no SKIP/SKIPF/EXECF/XBYTE pattern queued (D = 0) or 1 if pattern queued (D <> 0)

D = 32-bit SKIP/SKIPF/EXECF/XBYTE pattern, used LSB-first to skip instructions in main code

Appendix “C” EVENTS

EVENTS are actions that Cogs individually monitor and track there are 16 different background events for each running Cog:

- 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.

C.1) Polled Events

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-01FF-read event flag	13
POLLATN	poll the attention-requested event flag	14
POLLQMT	Poll the CORDIC-read-but-no-results event flag	15

C.2) WAITxx Instructions

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
WAITXFI	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.

C.3) Selectable Events

Each cog can track up to four selectable pin, LUT, or hub lock events. This is accomplished by using the SETSEn instruction, where "n" is 1, 2, 3, or 4. In order for user code to detect the occurrence of the selected event, the following options are available:

The matched WAITSEn instruction will block until the event occurs

- The matched POLLEn instruction will check for the event without blocking
- The matches JSEn and JNSEn branch instructions will branch according to the polled event state
- As an interrupt (see [INTERRUPTS](#))

Each selected event is set or cleared according to the following rules:

- SEn is set whenever the configured event occurs.
- SEn is cleared on matched POLLEn / WAITSEn / JSEn / JNSEn.
- SEn is cleared when matched 'SETSEn D/#' is called.

SETSEn D/# accepts the following configuration values:

%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

C.3) Interrupt Jump Instructions

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

C.4) Details on Polled/Wait/Interrupt Instructions

Polled Events

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.

POLLPAT/WAITPAT event flag

Cleared on SETPAT

Set whenever $(INA \& D) \neq S$ 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 $(INB \& D) \neq S$ 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 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 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 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/WAITXRL 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.

WAITxx Instructions

'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.

Selectable Events

Each cog can track up to four selectable pin, LUT, or hub lock events. This is accomplished by using the SETSEn instruction, where "n" is 1, 2, 3, or 4. In order for user code to detect the occurrence of the selected event, the following options are available:

- The matched WAITSEn instruction will block until the event occurs
- The matched POLLSEn instruction will check for the event without blocking
- The matches JSEn and JNSEn branch instructions will branch according to the polled event state
- As an interrupt (see [INTERRUPTS](#))

Each selected event is set or cleared according to the following rules:

- SEn is set whenever the configured event occurs.
- SEn is cleared on matched POLLSEn / WAITSEn / JSEn / JNSEn.
- SEn is cleared when matched 'SETSEn D/#' is called.

SETSEn D/# accepts the following configuration values:

%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

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:

RET11	=	CALLD	INB,IRET1	WCZ
RET12	=	CALLD	INB,IRET2	WCZ
RET13	=	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:

RES11	=	CALLD	IJMP1,IRET1	WCZ
RES12	=	CALLD	IJMP2,IRET2	WCZ
RES13	=	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 during execution before the interrupt branch can happen:

ALTxx / CRCNIB / SCA / SCAS / 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 at the same priority level 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

In addition to the 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.

Debug interrupts are enabled on a per-cog basis via HUBSET. Each debug-enabled cog will generate a debug interrupt on (re)start from each COGINIT exercised upon it. Within that initial debug ISR and within each subsequent debug ISR, multiple trigger conditions may be set for the next debug interrupt. If no trigger conditions are set before the debug ISR ends, no more debug interrupts will occur until the cog is restarted from another COGINIT.

The last 16KB of hub RAM, which is also mapped to \$FC000..\$FFFFFF, gets partially used as a buffer area for saving and restoring cog registers during debug ISR's. The initial debug ISR routines are also stored in this upper RAM. Once initialized with debug ISR code, this upper hub RAM can be write-protected, in which case it is mapped only to \$FC000..\$FFFFFF and it is only writable from within debug ISR's.

Each cog has an execute-only ROM in cog registers \$1F8..\$1FF which contains special debug-ISR-entry and -exit routines. These tiny routines perform seamless register-load and register-restore operations for your debugger program, which must be realized entirely within debug ISR's.

Execute-only ROM in cog registers \$1F8..\$1FF (%cccc = !CogNumber)			
Debug ISR Entry - IJMP0 is initialized to \$1F8 on cog start			
\$1F8 -	SETQ	#\$0F	'save registers \$000..\$00F
\$1F9 -	WRLONG	0,*	'* = %1111_1111_1ccc_c000_0000
\$1FA -	SETQ	#\$0F	'load program into \$000..\$00F
\$1FB -	RDLONG	0,*	'* = %1111_1111_1ccc_c100_0000
\$1FC -	JMP	#0	'jump to loaded program

Debug ISR Exit - Jump here to exit your debug ISR			
\$1FD	-	SETQ	#\$0F 'restore registers \$000..\$00F
\$1FE	-	RDLONG	0,* '* = %1111_1111_1ccc_c000_0000
\$1FF	-	RETIO	'CALLD IRET0,IRET0 WCZ

During a debug ISR, INA and INB, normally read-only input-pin registers, become readable/writable RAM registers named IJMP0 and IRET0, and are used by the debug interrupt as jump and return addresses. On COGINIT, IJMP0 is initialized to \$1F8 which is the debug-ISR-entry routine's address.

When a debug interrupt occurs with IJMP0 pointing to \$1F8, the following sequence happens:

- Cog registers \$000 to \$00F are saved to hub RAM starting at (\$FF800 + !CogNumber << 7), or %1111_1111_1ccc_c000_0000, where %cccc = !CogNumber.
- Cog registers \$000 to \$00F are loaded from hub RAM starting at (\$FF840 + !CogNumber << 7), or %1111_1111_1ccc_c100_0000, where %cccc = !CogNumber.
- A "JMP #\$000" executes to run the 16-instruction debugger program that was just loaded into registers \$000 to \$00F.

Your 16-instruction debugger program will likely want to determine if this debug interrupt was due to a COGINIT, in which case the debugger will probably want to note that a new program is now running in this cog. Depending on what the debugger must do next, it is likely that it will need to save more registers to the upper hub RAM and then load in more code from the upper hub RAM to facilitate more complex operations than the initial 16-instruction ISR can achieve. The ISR may then need to perform some communication between itself and a host system which may be serving as the debugger's user interface. It may be necessary to employ a LOCK to time-share P2-to-host communication channels among cogs, likely on P63 (serial Rx) and P62 (serial Tx). This scenario is somewhat hypothetical, but illustrates the design intent behind the debug interrupt mechanism.

When your debug ISR is complete, you can do a 'JMP #\$1FD' to execute the debug-ISR-exit routine which does the following:

- Original cog registers \$000 to \$00F are restored from hub RAM starting at (\$FF800 + !CogNumber << 7), or %1111_1111_1ccc_c000_0000, where %cccc = !CogNumber.
- A "RETIO" executes to return to the interrupted cog program.

Here is a table of the hub RAM locations used by each cog for register save/restore and ISR images during the debug interrupt when the register ROM routines are used for ISR entry and exit:

Cog	Save/Restore in Hub RAM for Registers \$000..\$00F	ISR image in Hub RAM for Registers \$000..\$00F
7	\$FFC00 .. \$FFC3F	\$FFC40 .. \$FFC7F
6	\$FFC80 .. \$FFCBF	\$FFCC0 .. \$FFCFF
5	\$FFD00 .. \$FFD3F	\$FFD40 .. \$FFD7F
4	\$FFD80 .. \$FFDBF	\$FFDC0 .. \$FFDDF
3	\$FFE00 .. \$FFE3F	\$FFE40 .. \$FFE7F
2	\$FFE80 .. \$FFEBF	\$FFEC0 .. \$FFEFF
1	\$FFF00 .. \$FFF3F	\$FFF40 .. \$FFF7F
0	\$FFF80 .. \$FFFBF	\$FFFC0 .. \$FFFFF

Though the first debug interrupt upon cog (re)start will always use the debug-ISR-entry routine at \$1F8, you may redirect IJMP0 during any debug ISR to point elsewhere for use by subsequent debug interrupts. This would mean that you would lose the initial register-saving function provided by the small ROM starting at \$1F8, so you would have to use some cog registers for debugger-state storage that don't interfere with the cog program that is being debugged. If no register saving/restoring or host

communications are required, your debug ISR may execute very quickly.

What terminates a debug interrupt is not only RETIO (CALLD INB,INB WCZ), but any D-register variant (CALLD anyreg,INB WCZ). For example RESIO (CALLD INA,INB WCZ) may be used to resume next time from where this debug ISR left off, but this would imply that you are not using the debug-ISR-entry and -exit routines in the cog-register ROM and have, instead, permanently located debugger code into some cog registers, so that your debugger program is already present at the start of the debug interrupt.

This debug interrupt scheme was designed to operate stealthily, without any cooperation from the cog program being debugged. All control has been placed within the debug ISR. 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 the debug interrupt compromised as a standard debugging mechanism. Also, by executing the ISR strictly in cog register space, this scheme does not interfere with the hub FIFO state, which would be impossible to reconstruct if disturbed by hub execution within the debug ISR.

Below are the instructions which are used in the debugging mechanism:

BRK D/#

During normal program execution, the BRK instruction is used to generate a debug interrupt with an 8-bit code which can be read within the debug ISR. The BRK instruction interrupt must be enabled from within a prior debug ISR for this to work. Regardless of the execution condition, the BRK instruction will trigger a debug interrupt, if enabled. The execution condition only gates the writing of the 8-bit code:

D/# = %BBBBBBBB: 8-bit BRK code

During a debug ISR, the BRK instruction operates differently and is used to establish the next debug interrupt condition(s). It is also used to select INA/INB, instead of the IJMP0/IRETO registers exposed during the ISR, so that the pins' inputs states may be read:

D/# = %aaaaaaaaaaaaaaaaeeee_LKJIHGFE DCBA

%aaaaaaaaaaaaaaaaaaaae: 20-bit breakpoint address or 4-bit event code (**%eeee**)

%L: 1 = map INA/INB normally, 0 = map IJMP0/IRET0 at INA/INB (default during ISR) *

%K: 1 = enable interrupt on breakpoint address match

%J: 1 = enable interrupt on event **%eeee**

%I: 1 = enable interrupt on asynchronous breakpoint (via COGBRK on another cog)

%H: 1 = enable interrupt on INT3 ISR entry

%G: 1 = enable interrupt on INT2 ISR entry

%F: 1 = enable interrupt on INT1 ISR entry

%E: 1 = enable interrupt on BRK instruction

%D: 1 = enable interrupt on INT3 ISR code (single step)

%C: 1 = enable interrupt on INT2 ISR code (single step)

%B: 1 = enable interrupt on INT1 ISR code (single step)

%A: 1 = enable interrupt on non-ISR code (single step)

* If set to 1 by the debug ISR, **%L** must be reset to 0 before exiting the debug ISR, so

that the RETI0 instruction is able to see IJMP0 and IRET0.

On debug ISR entry, bits A to L, are cleared to '0'. If a subsequent debug interrupt is desired, a BRK instruction must be executed before exiting the debug ISR, in order to establish the next breakpoint condition(s).

COGBRK D/#

The COGBRK instruction can trigger an asynchronous breakpoint in another cog. For this to work, the cog executing the COGBRK instruction must be in its own debug ISR and the other cog must have its asynchronous breakpoint interrupt enabled:

D/# = %CCCC: the cog in which to trigger an asynchronous breakpoint

GETBRK D WCZ

During normal program execution, GETBRK with WCZ returns various data about the cog's internal status:

C = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)

Z = 1 if cog started in hubexec or 0 if cog started in cogexec

D[31:23] = 0

D[22] = 1 if colorspace converter is active

D[21] = 1 if streamer is active

D[20] = 1 if WRFAST mode or 0 if RDFAST mode

D[19:16] = INT3 selector, established by SETINT3

D[15:12] = INT2 selector, established by SETINT2

D[11:08] = INT1 selector, established by SETINT1

D[07:06] = INT3 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[05:04] = INT2 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[03:02] = INT1 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing

D[01] = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)

D[00] = 1 if cog started in hubexec or 0 if cog started in cogexec

During a debug ISR, GETBRK with WCZ returns additional data that is useful to a debugger:

C = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started

D[31:24] = 8-bit break code from the last 'BRK #/D' during normal execution

D[23] = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started

GETBRK D WC

GETBRK with WC always returns the following:

C = LSB of SKIP/SKIPF/EXECF/XBYTE pattern

D[31:28] = 4-bit CALL depth since SKIP/SKIPF/EXECF/XBYTE (skipping suspended if not %0000)

D[27] = 1 if SKIP mode or 0 if SKIPF/EXECF/XBYTE mode

D[26] = 1 if LUT sharing enabled (established by SETLUTS)

D[25] = 1 if top of stack = \$001FF, indicating XBYTE will execute on next `_RET_/RET`

D[24:16] = 9-bit XBYTE mode, established by '`_RET_ SETQ/SETQ2`' when top of stack = \$001FF

D[15:00] = 16 event-trap flags

GETBRK D WZ

GETBRK with WZ always returns the following:

Z = 1 if no SKIP/SKIPF/EXECF/XBYTE pattern queued (D = 0) or 1 if pattern queued (D <> 0)

D = 32-bit SKIP/SKIPF/EXECF/XBYTE pattern, used LSB-first to skip instructions in main code

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/WAITXRL 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.

Note: the .loop operator is cleared after compiling instruction and loop can be re-used for writing jump code. Remember to include #(immediate) directive

Appendix “D” P2 Edge

D.1) Edge Specifications

Features

- Compact module with Propeller 2 P2X8C4M64P multicore microcontroller
 - 6-layer, low noise, system-on-board module
 - Integrated thermal planes for low temperature rise characteristics at high speed operation
 - Double-sided 80 way 0.05” (1.27mm) edge connector
 - Orientation / module locking hole
 - Two mounting holes connected to the module ground planes
 - 20 MHz crystal
 - Adjustable operating frequency; recommended maximum 180 MHz clock
 - Overclocking possible beyond 300 MHz
 - 16 MB SPI Flash memory
 - 64 Smart I/O pins brought out to the Edge Connector
 - Buffered LEDs on I/O pins P56 and P57, visible from both sides of the module PCB
 - Onboard LED feature enable/disable switch
 - Onboard 1.8 V 2-Amp switching regulator with short-circuit, over-current fault and brownout detection protection for the P2 core (VDD)
 - Onboard low-noise LDO 3.3 V regulators for the P2 smart-pins (VIO), with short-circuit and over-current fault protection
 - Dual power inputs via the edge connector or optional header pads on the back of the module, with reverse polarity protection
 - Compatible with the Parallax Prop-Plug #32201 for system programming
- Key Specifications
- Voltage input requirements: 5 VDC; absolute maximum 5.5 VDC

Input Current requirements:

- Recommended minimum 100 mA
- Typical experimentation 500–1000 mA
- Maximum according to customer application
- Voltage input protection: reverse voltage
- Propeller 2 chip: P2X8C4M64P (8 cogs, 512 KB shared hub RAM, 64 smart pins)
- Non-volatile Memory: 16 MB (128 Mb) SPI Flash
- Crystal: 20 MHz SMT
- Smart I/O pins: 64 accessible, 56 fully free, grouped in 8 sets of 8 I/Os
- Smart I/O pin logic voltage: 3.3 V
- Internal VDD Power Supply: 1.8 V up to 2 A, 1 MHz nominal switching frequency
- VIO Power Supplies: 3.3V up to 300 mA per 8 I/O pins
- Edge Connector: Double sided 80 way 0.05" (1.27mm) pitch edge slot
- Programming: Serial up to 2 MBaud
- Operating temperature: -40 to +185 °F (-40 to +85 °C)
- PCB Dimensions: 1.45 in x 2.04 in (37mm x 52mm)

Appendix “E” Miscelanous Variable Type Definition

E.0) Byte/Word/Long Declaration

Propeller has only 3 types of variables Byte\Word\Long.

Propeller/Spin endianness is very simple:

The Propeller is a little-endian processor. The less significant bytes of a word or long are stored in the lower memory locations. VARiables and DATa are stored little-endian once compiled. However, since we (i.e. humans who are used to LTR languages) are used to writing numbers in big-endian order, the **Spin compiler**, for convenience, lets us write "byte \$76543210" when we want "byte \$10, \$32, \$54, \$76".

The variables are arranged long, word, byte at compile time. There aren't empty bytes between each byte variable. Individual bytes are addressable in hub RAM. Cog RAM is addressable only by longs.

E.O.1) Byte Declaration 8 bits

Syntax 1 Var (variable declaration)

VAR

byte Temp 'Temp is a byte variable

byte Str[25] 'Str is a byte array Str[0]-Str[24]

Syntax 2 Dat (data declaration)

DAT

MyData byte 41," A", \$2A

MyString byte "Hello", 0 'zero terminated string

Hub Memory DAT Block Access

Pub GetData|Temp

Temp := MyData 'reads first byte of MyData

<more code>

Pub GetData |Index,Temp

Index := 0

Repeat

Temp := MyString[Index++]

<more code>

While temp > 0 'check after loop

Syntax 3 Byte Hub Memory DAT Block Access

Pub GetData|Temp

Temp :=Byte[@MyData]

<more code>

Pub GetData |Index,Temp

Index := 0

Repeat

Temp := Byte[@MyString][Index++]

<more code>

While temp > 0

E.O.1.1_Example_WRD_Byte_Access from Memory

```
{{E.O.1.1_Example_WRD_Byte_Access from Memory}}
```

```
{{
```

```
Temp := Byte[@MyString][Index++]
```

```
Temp := MyData
```

```
Example
```

```
Demonstrate Byte data access
```

```
}}
```

```
CON
```

```
_clkfreq = 200_000_000 "Debug must be enabled clock must be greater than 10MHZ for Debug
```

```
P0 = 0 , P1 = 1 , P2 = 2
```

```
VAR
```

```
byte Temp 'Temp is a byte variable
```

```
byte Str[25] 'Str is a byte array Str[0]-Str[24]
```

```
PUB main()
```

```
debug("-----")
```

```
debug("Example")
```

```
debug("Demonstrate Byte data access")
```

```
debug("-----")
```

```
GetData1()
```

```
GetData2()
```

```
GetData3()
```

```
GetData4()
```

```
Repeat
```

```
Pub GetData1()|Temp1
```

```
Temp1 := MyData 'reads first byte of MyData
```

```
debug("MyData ",udec_byte(Temp1))
```

```
Pub GetData2() |Index2,Temp2
```

```
Index2 := 0
```

```
Repeat
```

```
Temp2 := MyString[Index2++]
```

```
debug("MyString ",udec_byte(Temp2))
```

```
While Temp2 > 0 'check after loop entry
```

```
Pub GetData3()|Index3,Temp3
```

```
Index3 := 0
```

```
Repeat
```

```
Temp3 := Byte[@MyData][Index3++]
```

```
debug("MyData ",udec_byte(Temp3))
```

```
while Temp3 > 0 'check after loop entry
```

```
Pub GetData4()|Index4,Temp4
```

```
Index4 := 0
```

```
Repeat
```

```
Temp4 := Byte[@MyString][Index4++]
```

```
debug("MyString ",udec_byte(Temp4))
```

```
While Temp4 > 0 'check after loop entry
```

DAT

MyData byte 41,"A","B","C", \$2A,0

MyString byte "Hello", 0 'zero terminated string

E.0.2) Word Declaration 16 bits

Syntax 1 Var (variable declaration)

VAR

word var01 'var01 Temp is a word variable

word List[25] 'List is a word array Str[0]-Str[24]

Syntax 2 Dat (data declaration)

DAT

MyList word \$FFFF, 41," A", \$2A

E.0.3) Long Declaration 32 bits

Syntax 1 Var (variable declaration)

VAR

Long Temp ' Temp is a Long variable

Long List[25] 'List is a long array

Syntax 2 Dat (data declaration)

DAT

MyData Long 640_000, \$BB50 'Long-aligned/sized data

MyList Byte Long \$FF995544 , Long 1_000 ' Byte-aligned/long sized

Syntax 3 PUB/PRI

Pub method| Index, var01 'declares local method variable of type Long

E.0.3.1_Example_WRD_Long_Aligned_Memory

E.0.4) Address Convention

$B[7:0] = B_7B_6B_5B_4B_3B_2B_1B_0$

$W[15:0] = B_{15}B_{14}B_{13}B_{12}B_{11}B_{10}B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0$

$L[31:0] = B_{31}B_{30}B_{29}B_{28}B_{27}B_{26}B_{25}B_{24}B_{23}B_{22}B_{21}B_{20}B_{19}B_{18}B_{17}B_{16}B_{15}B_{14}B_{13}B_{12}B_{11}B_{10}B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0$

Byte Addressing

$D_B[3:0]$

$= D_3D_2D_1D_0$

Bit Addressing

$D[31:0]$

$= d_{31}d_{30}d_{29}d_{28}d_{27}d_{26}d_{25}d_{24}d_{23}d_{22}d_{21}d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$

Byte Bit Addressing

$D_{BB}[37:00]$

$= d_{37}d_{36}d_{35}d_{34}d_{33}d_{32}d_{31}d_{30}d_{27}d_{26}d_{25}d_{24}d_{23}d_{22}d_{21}d_{20}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_7d_6d_5d_4d_3d_2d_1d_0$

Nibble Addressing

$D_N[7:0] = D_7D_6D_5D_4D_3D_2D_1D_0$

$D_7 = d_{73}d_{72}d_{71}d_{70}$ $D_6 = d_{63}d_{62}d_{61}d_{60}$ $D_5 = d_{53}d_{52}d_{51}d_{50}$ $D_4 = d_{43}d_{42}d_{41}d_{40}$

$D_3 = d_{33}d_{32}d_{31}d_{30}$ $D_2 = d_{23}d_{22}d_{21}d_{20}$ $D_1 = d_{13}d_{12}d_{11}d_{10}$ $D_0 = d_{03}d_{02}d_{01}d_{00}$

Byte Addressing

$S_B[3:0]$

$= S_3S_2S_1S_0$

Bit Addressing

$S[31:0]$

$= s_{31}s_{30}s_{29}s_{28}s_{27}s_{26}s_{25}s_{24}s_{23}s_{22}s_{21}s_{20}s_{19}s_{18}s_{17}s_{16}s_{15}s_{14}s_{13}s_{12}s_{11}s_{10}s_9s_8s_7s_6s_5s_4s_3s_2s_1s_0$

Byte Bit Addressing

$S_{BB}[31:0]$

$= s_{37}s_{36}s_{35}s_{34}s_{33}s_{32}s_{31}s_{30}s_{27}s_{26}s_{25}s_{24}s_{23}s_{22}s_{21}s_{20}s_{17}s_{16}s_{15}s_{14}s_{13}s_{12}s_{11}s_{10}s_7s_6s_5s_4s_3s_2s_1s_0$

Nibble Addressing

$S_N[7:0] = S_7S_6S_5S_4S_3S_2S_1S_0$

$S_7 = s_{73}s_{72}s_{71}s_{70}$ $S_6 = s_{63}s_{62}s_{61}s_{60}$ $S_5 = s_{53}s_{52}s_{51}s_{50}$ $S_4 = s_{43}s_{42}s_{41}s_{40}$

$S_3 = s_{33}s_{32}s_{31}s_{30}$ $S_2 = s_{23}s_{22}s_{21}s_{20}$ $S_1 = s_{13}s_{12}s_{11}s_{10}$ $S_0 = s_{03}s_{02}s_{01}s_{00}$

B[31:0]

= B₃₁B₃₀B₂₉B₂₈B₂₇B₂₆B₂₅B₂₄B₂₃B₂₂B₂₁B₂₀B₁₉B₁₈B₁₇B₁₆B₁₅B₁₄B₁₃B₁₂B₁₁B₁₀ B₉B₈B₇B₆B₅ B₄B₃B₂B₁B₀

S[31:0]

= S₃₁S₃₀S₂₉S₂₈S₂₇S₂₆S₂₅S₂₄S₂₃S₂₂S₂₁S₂₀S₁₉S₁₈S₁₇S₁₆S₁₅S₁₄S₁₃S₁₂S₁₁S₁₀S₀₉S₀₈S₀₇S₀₆S₀₅S₀₄S₀₃S₀₂S₀₁S₀₀

D[31:0]

= d₃₁d₃₀d₂₉d₂₈d₂₇d₂₆d₂₅d₂₄d₂₃d₂₂d₂₁d₂₀d₁₉d₁₈d₁₇d₁₆d₁₅d₁₄d₁₃d₁₂d₁₁d₁₀d₀₉d₀₈d₀₇d₀₆d₀₅d₀₄d₀₃d₀₂d₀₁d₀₀

N[7:0] = N₇N₆N₅N₄N₃N₂N₁N₀

N₇ = n₇₃n₇₂n₇₁n₇₀ N₆ = n₆₃n₆₂n₆₁n₆₀ N₅ = n₅₃n₅₂n₅₁n₅₀ N₄ = n₄₃n₄₂n₄₁n₄₀

N₃ = n₃₃n₃₂n₃₁n₃₀ N₂ = n₂₃n₂₂n₂₁n₂₀ N₁ = n₁₃n₁₂n₁₁n₁₀ N₀ = n₀₃n₀₂n₀₁n₀₀

S_N[7:0] = S₇S₆S₅S₄S₃S₂S₁S₀

S₇ = s₇₃s₇₂s₇₁s₇₀ S₆ = s₆₃s₆₂s₆₁s₆₀ S₅ = s₅₃s₅₂s₅₁s₅₀ S₄ = s₄₃s₄₂s₄₁s₄₀

S₃ = s₃₃s₃₂s₃₁s₃₀ S₂ = s₂₃s₂₂s₂₁s₂₀ S₁ = s₁₃s₁₂s₁₁s₁₀ S₀ = s₀₃s₀₂s₀₁s₀₀

D_N[7:0] = D₇D₆D₅D₄D₃D₂D₁D₀

D₇ = d₇₃d₇₂d₇₁d₇₀ D₆ = d₆₃d₆₂d₆₁d₆₀ D₅ = d₅₃d₅₂d₅₁d₅₀ D₄ = s₄₃s₄₂s₄₁s₄₀

D₃ = d₃₃d₃₂d₃₁d₃₀ D₂ = d₂₃d₂₂d₂₁d₂₀ D₁ = d₁₃d₁₂d₁₁d₁₀ D₀ = d₀₃d₀₂d₀₁d₀₀

W[31:0] = W₁W₀

= W₃₁W₃₀W₂₉W₂₈W₂₇W₂₆W₂₅W₂₄W₂₃W₂₂W₂₁W₂₀W₁₉W₁₈W₁₇W₁₆_W₁₅W₁₄W₁₃W₁₂W₁₁W₁₀W₉W₈_W₇W₆W₅W₄W₃W₂W₁W₀

W₁ = w₀₁₅w₀₁₄w₀₁₃w₀₁₂w₀₁₁w₀₁₀w₀₀₉w₀₀₈w₀₀₇w₀₀₆w₀₀₅w₀₀₄w₀₀₃w₀₀₂w₀₀₁w₀₀₀

W₂ = w₁₁₅w₁₁₄w₁₁₃w₁₁₂w₁₁₁w₁₁₀w₁₀₉w₁₀₈w₁₀₇w₁₀₆w₁₀₅w₁₀₄w₁₀₃w₁₀₂w₁₀₁w₁₀₀

Example

D[BH:BL] = D[S[9:5] + S[4:0] : S[4:0]] = 0 Defines a range of bits

S[9:5] = %10001 = 17 S[4:0] = %00001 = 1

D[BH:BL] = D[%1001 + %00001 : %00001] = D[17 + 1: 1] = D[18:1] = 0

D[BH:BL] = D[S[9:5] + S[4:0] : S[4:0]] = 0 Defines a range of bits

S[31:0] = S[31:10] + S[9:5] + S[4:0] Defines Special function range

E.1) Number Types Signed, Unsigned, Float, Modular

Unsigned 32 bit number has a range from 0 ---- 4_294_967_295

Signed 32 bit number has a range from -2_147_483_648 ---- 2_147_483_647

Note: When possible integer operations should be used they are faster and simpler Float values can be scaled up: $5.6/7.8 = 56/78$

E.1.1_Example_WRD_Signed_Unsigned_Numbers

```
{{E.1_Example_WRD_Signed_Unsigned_Numbers}}
```

```
"unsigned,signed
```

```
CON
```

```
_clkfreq = 200_000_000 "Debug must be enabled clock must be greater than 10MHZ for Debug
```

```
VAR
```

```
Byte cogRunning 'cog ID started is returned or -1 if not started
```

```
PUB main()
```

```
  cogRunning := COGINIT(COGEXEC_NEW,@NumTypes,0)
```

```
  debug(udec(cogRunning))
```

```
  debug(ubin(UnSignedMax),uhex(UnSignedMax),udec(UnSignedMax))
```

```
  debug(ubin(UnSignedMin),uhex(UnSignedMin),udec(UnSignedMin))
```

```
  debug(sbin(SignedMax),shex(SignedMax),sdec(SignedMax))
```

```
  debug(sbin(SignedMin),shex(SignedMin),sdec(SignedMin))
```

```
  repeat
```

```
DAT
```

```
  ORG 0
```

```
NumTypes
```

```
_Loop
```

```
  NOP
```

```
  JMP #_Loop
```

```
UnSignedMax    long    $FFFF_FFFF
```

```
UnSignedMIN    long    0
```

```
SignedMax      long    %01111111_11111111_11111111_11111111
```

```
SignedMin      long    %10000000_00000000_00000000_00000000
```

```
UnSignedBinMax long    %11111111_11111111_11111111_11111111
```

```
UnSignedBinMin long    %00000000_00000000_00000000_00000000
```

```
UnSignedHexMax long    $FFFF_FFFF
```

```
UnSignedHexMin long    $0000_0000
```

```
UnSignedDecMax long    4_294_967_295
```

```
UnSignedDecMin long    0
```

```
SignedBinMax   long    %01111111_11111111_11111111_11111111
```

```
SignedBinMin   long    %10000000_00000000_00000000_00000000
```

```
SignedHexMax   long    $7FF_FFFF
```

```
SignedHexMin   long    -$800_0000
```

```
SignedDecMax   long    2_147_483_647
```

```
SignedDecMin   long    -2_147_483_648
```


E.1.2) Floating Point Numbers

$$\text{Float Base 10 } 12.120 = 1 \times 10^1 + 2 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2}$$

$$\text{Float Base 2 } 101.101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

Convert Base 2 Float 101.101 to Base 10 Float

$$(101.101)_2$$

$$(101)_2 = 4 + 0 + 1 = 5$$

$$(.101)_2 = 1 \times 1/2^1 + 0 \times 1/2^2 + 1 \times 1/2^3$$

$$(.101)_2 = 1 \times .5 + 0 \times .25 + 1 \times .125 = (.625)_{10}$$

$$(101.101)_2 = (5.625)_{10}$$

Convert Base 10 Float 5.625 to Base 2 Float

$$5.625$$

$$(5)_{10} = (4 + 0 + 1)_{10} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (101)_2$$

$$.625 \times 2 = 1.25 \quad 1$$

$$.25 \times 2 = .5 \quad 0$$

$$.5 \times 2 = 1.0 \quad 1$$

$$(.625)_{10} = (.101)_2$$

$$(5.625)_{10} = (101.101)_2$$

E.1.3) Modular Arithmetic

An Introduction to Modular Math

When we divide two integers we will have an equation that looks like the following:

$$\frac{A}{B} = Q \text{ remainder } R$$

A is the dividend

B is the divisor

Q is the quotient

R is the remainder

Sometimes, we are only interested in what the **remainder** is when we divide A by B.

For these cases there is an operator called the modulo operator (abbreviated as mod).

Using the same A, B, Q, and R as above, we would have: $A \bmod B = R$

We would say this as **A modulo B is equal to R**. Where B is referred to as the modulus.

$$\text{Fo } \frac{13}{5} = 2 \text{ remainder } 3$$

E.3.1) Time as Modular Arithmetic

Clock Arithmetic or a Circle as a Number Line One way to turn a circle into a number line is to divide it into twelve equal parts. In this case, one step is usually called one hour.



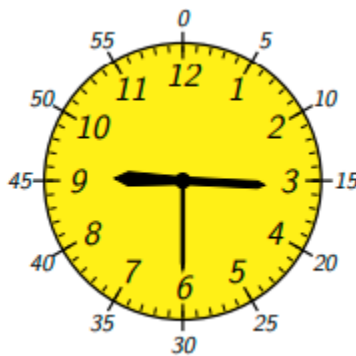
The hour hand moves from 0 to 1, from 1 to 2, ... from 11 to 12 just as it would have on the straight number line. However, 12 equals 0 on this circle, so there it goes 2 Notice that 0 coincides with 12, and as the hour hand moves to the right, 1 coincides with 13, 2 with 14, and so on. The hour hand rotates clockwise which corresponds with numbers increasing when moving to the right on a number line. However, 12 is equivalent to 0 on this circle, which can be written as follows:

$$12 \equiv 0 \pmod{12}$$

This can be read as 12 is congruent to 0 modulo 12. The usual "=" sign is reserved for the straight number line; we use " \equiv " on the circle instead. The symbol "mod 12" tells us that the circle is divided into 12 equal parts, so that 12 coincides with 0, 13 with 1, etc. In the new notation we have:

$$12 \equiv 0 \pmod{12}, \quad 13 \equiv 1 \pmod{12}, \quad \dots \quad 23 \equiv 11 \pmod{12}$$

The 24-Hour Clock There are 24 hours in a day, so one more standard way to turn a circle into a number line is to divide it into 24 equal parts. The US military uses the 24 hour clock. Since 60 is not a multiple of 24, we can't use the same marks on the face of a 24 hour clock for minutes and hours (look at the minute marks on the face of the 24 hour clock).



Modular Arithmetic

In addition to clock analogy, one can view modular arithmetic as arithmetic of remainders. For example, in mod 12 arithmetic, all the multiples of 12 (i.e., all the numbers that give remainder 0 when divided by 12) are equivalent to 0. In the modular arithmetic notation, this can be written as:

$$12 \times n \equiv 0 \pmod{12} \text{ for any whole number } n.$$

Similarly, all numbers that give remainder 1 when divided by 12 are equivalent to 1. In other words:

$$12 \times n + 1 \equiv 1 \pmod{12} \text{ for any whole number } n.$$

Recall that any whole number a can be uniquely written in the form:

$$a = 12 \times n + r$$

where r is one of the numbers 0, 1, ..., 11. Notice that r is the remainder of the division of a by 12. Therefore, $a \equiv r \pmod{12}$. For example:

$$\begin{aligned} 50 &= 5 \times 12 + 10, \text{ which implies } 50 \equiv 10 \pmod{12}, \\ 40 &= 3 \times 12 + 4, \text{ which means } 40 \equiv 4 \pmod{12}. \end{aligned}$$

E3.2) Addition /Subtraction property of modular arithmetic:

$$(A + B) \bmod C = (A \bmod C + B \bmod C) \bmod C$$

$$(A - B) \bmod C = (A \bmod C - B \bmod C) \bmod C$$

Exa

Let $A=14$

$LH =$

$RH = LHS$

14

$31 \bmod$

$LHRHS$

14

4

$RHLS = RHS = 1$

E3.3) Multiplication property of modular arithmetic:

(A ExaLet $A \equiv B \pmod{m}$ $A * B \pmod{m}$ LHS= RHS= ti LHS $(A \pmod{m} LHS (4 \pmod{m} LHS 28 \pmod{m} LHS 4RHS A \pmod{m} * B \pmod{m})$
 $RHS 4 * 7 \pmod{m} RHS 4 \pmod{m} RHS 4LS = RHS = 4$

E.1.4) Binary Operations

E.2.3) CRC8 Dallas/Maxim Algorithm

Binary Multiplication

$$\begin{array}{r}
 101 \quad 5 \quad 111001 + 1 = 26 \quad 26 \bmod 5 = 1 \\
 \underline{\times 101} \quad \underline{\times 5} \\
 101 \quad 25 \\
 0000 \\
 \underline{10100} \\
 11001 = 25
 \end{array}$$

Binary Division

$$\begin{array}{r}
 \quad \quad \quad \underline{101} \quad \underline{\hspace{1cm}} \\
 101 \mid 11010 \\
 \underline{101} \\
 10 \\
 \underline{00} \\
 100 \\
 \underline{101} \\
 1 \text{ Remainder}
 \end{array}$$

$$26 \bmod 5 = 1$$

Modulo 2 Division XOR

Modulo-2 division is performed similarly to “normal” arithmetic division. The only difference is that we use modulo-2 subtraction (**XOR**) instead of arithmetic subtraction for calculating the remainders in each step. The quotient is not of interest.

$$\begin{array}{r}
 \quad \quad \quad \underline{111} \quad \underline{\hspace{1cm}} \\
 101 \mid 11010 \\
 \underline{101} \\
 111 \\
 \underline{101} \\
 100 \\
 \underline{101} \\
 1 \text{ Remainder}
 \end{array}$$

$$26 \bmod 5 = 1$$

E.2) CRC8 Cycle Redundancy Check

CRC stands for Cyclic Redundancy Check. It is an error-detecting code used to determine if a block of data has been corrupted. The idea is given a block of N bits, let's compute a checksum of a sort to see if the N bits were damaged in some way, for instance by transit over a network. The extra data we transmit with this checksum is the "Redundancy" part of CRC, and the second C just means this is a "Check" to see if the data are corrupted (as opposed to an ECC code, which both detects and corrects errors).

Simple Parity is another method for error checking for example the number of 1's and zero's are even or odd parity for example 10101010 to be even parity a 1 would be required to append 10101010_1 to obtain 4 one bits for even parity if 0 odd parity used a 0 bit would be added. If two bits are switched or lost the error checking will not detect it. Only single bit errors can be detected.

Ce is another method for error detection. Unfortunately sometimes a CRC value is termed a CheckSum. **CRC**, treats the message as a big number, we choose a special number to divide the message by (referred to as the "CRC generation polynomial" divisor in the literature), and the remainder of the division is the CRC. Intuitively, it should be obvious that we can detect more than single bit errors with this scheme. Additionally, I think it is obvious that some divisors are better than others at detecting errors. Most implementations do not use division in the normal sense but use Module 2 arithmetic which eliminates the need for the borrowing operation. Modulus 2 arithmetic is XOR exclusive OR operation. For CRC calculations, no normal subtraction is used, but all calculations are done modulo 2. In that situation you ignore carry bits and in effect the subtraction will be equal to an exclusive or operation. This looks strange, the resulting remainder has a different value, but from an algebraic point of view the functionality is equal. A discussion of this would need university level knowledge of algebraic field theory.

The CRC is a predetermined number of bits to be used for the error detection. 8, 16, 32 or 64 bits are commonly used. This set of notes will concentrate on **C** 1 Wire usage errors. The number of bits in the error code is n and with CRC8 Dallas/Maxim n = 8.

Wyusator polynomial? Farca.Sufvehe
Binary Numbers can be represented as a Polynomial:

$$\begin{aligned}\text{BinaryNumber} = B[7:0] &= B_7X^7 + B_6X^6 + B_5X^5 + B_4X^4 + B_3X^3 + B_2X^2 + B_1X^1 + B_0X^0 \\ &= B_72^7 + B_62^6 + B_52^5 + B_42^4 + B_32^3 + B_22^2 + B_12^1 + B_02^0\end{aligned}$$

$$B_02^0 = B_0 * 1$$

$$B_12^1 = B_1 * 2$$

$$B_22^2 = B_2 * 4$$

$$B_32^3 = B_3 * 8$$

$$B_42^4 = B_4 * 16$$

$$B_52^5 = B_5 * 32$$

$$B_6 2^6 = B_6 * 64$$

$$B_7 2^7 = B_7 * 128$$

$$G(X) = B[8:0] = G_8 X^8 + G_7 X^7 + G_6 X^6 + G_5 X^5 + G_4 X^4 + G_3 X^3 + G_2 X^2 + G_1 X^1 + G_0 X^0$$

$$G(2) = x^8 + x^5 + x^4 + x^0 = 2^8 + 2^5 + 2^4 + 2^0 = G_7 G_6 G_5 G_4 G_3 G_2 G_1 G_0 = 100110001$$

$$\text{CRC8Maxim Divisor} = \%1_0011_0001 = 131_{16} = 305_{10}$$

Polynomial Generator bits 0-8 (9 actual bits $B_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0$)

$$\text{CRC8Dallas\Maxim} = X^8 + X^5 + X^4 + X^0$$

$$X^8 + X^5 + X^4 + X^0 = 1 * X^8 + 0 * X^7 + 0 * X^6 + 1 * X^5 + 1 * X^4 + 0 * X^3 + 0 * X^2 + 0 * X^1 + 1 * X^0$$

$$= \%1_0011_0001 \text{ this is the CRC8 9 bit divisor (Coefficients)}$$

Note: Polynomial is a shift left multiplier of base 2 = $\%10$

Endianness

The *endianness* is the order of bytes with which data words are stored. We distinguish the following types:

Little-endian: The *least* significant byte is stored at the smallest memory address. In terms of data transmission, the *least* significant byte is transmitted first.

Big-endian: The *most* significant byte is stored at the smallest memory address. In terms of data transmission, the *most* significant byte is transmitted first.

Note: Parallax propeller is Little Endian processor LSBytes stored in lowest memory address to MSBytes in increasing memory value.

The same conventions can be used in the ordering of the Polynomials. Typically Big Endian convention is mostly used for CRC calculations but little Endian convention can be used.

End_1 $X^8 + X^7 + X^6 + X^5 + X^4 + X^3 + X^2 + X + 1$ $B_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0$

Big Endian

$G(X) = B_8 X^8 + B_7 X^7 + B_6 X^6 + B_5 X^5 + B_4 X^4 + B_3 X^3 + B_2 X^2 + B_1 X^1 + B_0 X^0$
 100000111 $\rightarrow B[8:0] = 100000111$

Little Endian

$G(X) = B_0 X^8 + B_1 X^7 + B_2 X^6 + B_3 X^5 + B_4 X^4 + B_5 X^3 + B_6 X^2 + B_7 X^1 + B_8 X^0$
 100000111 $\rightarrow B[8:0] = 111000001$

Endian Example_2

$X^8 + X^5 + X^4 + 1$ $B[8:0] = B_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0$

Big Endian

$G(X) = B_8 X^8 + B_7 X^7 + B_6 X^6 + B_5 X^5 + B_4 X^4 + B_3 X^3 + B_2 X^2 + B_1 X^1 + B_0 X^0$
 100110001 $\rightarrow B[8:0] = 100110001$

Little Endian

$G(X) = B_0 X^8 + B_1 X^7 + B_2 X^6 + B_3 X^5 + B_4 X^4 + B_5 X^3 + B_6 X^2 + B_7 X^1 + B_8 X^0$
 100110001 $\rightarrow B[8:0] = 100011001$

Note: That most polynomial specifications either drop the [MSB](#) or [LSB](#), since they are always 1.
 CRC8Dallas/Maxim = $\$8C = \%10001100$ or $\%100011001$ adding 1 to LSB

The most commonly used generation polynomials are as follows:

$$\text{CRC8} = X^8 + X^5 + X^4 + X^0$$

$$\text{CRC-CCITT} = X^{16} + X^{12} + X^5 + X^0$$

$$\text{CRC16} = X^{16} + X^{15} + X^2 + X^0$$

$$\text{CRC12} = X^{12} + X^{11} + X^3 + X^2 + X^0$$

$$\text{CRC32} = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + X^0$$

Polynomial Name	Polynomial	Use
Custom	User defined	General
CRC-1	$x + 1$	Parity
CRC-4-ITU	$x^4 + x + 1$	ITU G.704
CRC-5-ITU	$x^5 + x^4 + x^2 + 1$	ITU G.704
CRC-5-USB	$x^5 + x^2 + 1$	USB
CRC-6-ITU	$x^6 + x + 1$	ITU G.704
CRC-7	$x^7 + x^3 + 1$	Telecom systems, MMC
CRC-8-ATM	$x^8 + x^2 + x + 1$	ATM HEC
CRC-8-CCITT	$x^8 + x^7 + x^3 + x^2 + 1$	1-Wire bus
CRC-8-Maxim	$x^8 + x^5 + x^4 + 1$	1-Wire bus

Most Hobbyist usage of CRC values use an 8 bit CRC value and for the remainder of this discussion CRC-8Maxim will be used. The manufacturer "MAXIM Integrated" now part of Analog devices originally used this for their 1 wire devices which has the CRC8Maxim registers built into their devices.

E.2.0) CRC Transmission Process

1) Create DataStream = Data + Checksum(CRC)

Divisor is CRC8Dallas\Maxim = $X^8 + X^5 + X^4 + X^0 = 305_{10} = 131_{16} = 100110001_2$

Data to be Transferred Let \$7778797A = DATA the bytes to be transferred

Dividend = DataStream + 8 zeroes (k = number bits in Divisor = n+1 = 9bits)

ADD "n" zero's (CRC8 is "8+1 = n+1") so add 8 zero's to stream to be transferred:

Dividend \$7778797A00 = 111_0111_0111_1000_0111_1001_0111_1010_0000_0000

Calculate the CRC (See CRC8 Dallas/Maxim Algorithm)

$CRC = 102_{10} = 0x66_{16} = 100110001_2$ from CRC calculator n = number bits in CRC = 8

Data Stream = \$7778797A66

2) Transmit Data From Sender to Receiver Device

Both Transmitter and receiver must be aware of "Generation Polynomial" in this case

CRC8Dallas\Maxim = $X^8 + X^5 + X^4 + X^0 = 305_{10} = 131_{16} = 100110001_2$

3) Receive data and create a receiving end CRC of Data Stream

When Checking CRC from the receiving end the , the generated CRC is appended to the Data since CRC is a linear function with a property that $CRC(x \oplus y \oplus z) = CRC(x) \oplus CRC(y) \oplus CRC(z)$.

DataStream XOR CRC = 0 in above example $0x66 \oplus 0x66 = 0$. Doing a CRC on the DataStream if data is good will have a CRC of 0.

E.2.1) Maxim 1-Wire CRC

The error detection scheme most effective at locating errors in a serial-data stream with a minimal amount of hardware is the CRC. The operation and properties of the CRC function used in Maxim products is presented without going into the mathematical details of proving the statements and descriptions. The mathematical concepts behind the properties of the CRC are described in detail in the references. The CRC can be most easily understood by considering the function as it would actually be built in hardware, usually represented as a shift register arrangement with feedback as shown in **Figure 2**. Alternatively, the CRC is sometimes referred to as a polynomial expression in a dummy variable X , with binary coefficients for each of the terms. The coefficients correspond directly to the feedback paths shown in the shift register implementation. The number of stages in the shift register for the hardware description, or the highest order coefficient present in the polynomial expression, indicate the magnitude of the CRC value that is computed. CRC codes that are commonly used in digital data communications include the CRC-16 and the CRC-CCITT, each of which computes a 16-bit CRC value. The Maxim 1-Wire CRC magnitude is 8 bits, which is used for checking the 64-bit ROM code written into each 1-Wire product. This ROM code consists of an 8-bit family code written into the least significant byte, a unique 48-bit serial number written into the next 6 bytes, and a CRC value that is computed based on the preceding 56 bits of ROM and then written into the most significant byte. The location of the feedback paths represented by the exclusive-OR gates in Figure 2, or the presence of coefficients in the polynomial expression, determine the properties of the CRC and the ability of the algorithm to locate certain types of errors in the data. For the 1-Wire CRC, the types of errors that are detectable are:

1. Any odd number of errors anywhere within the 64-bit number.
2. All double-bit errors anywhere within the 64-bit number.
3. Any cluster of errors that can be contained within an 8-bit "window" (1-8 bits incorrect).
4. Most larger clusters of errors.

The input data is exclusive-OR'ed with the output of the eighth stage of the shift register in Figure 2. The shift register can be considered mathematically as a dividing circuit. The input data is the dividend, and the shift register with feedback acts as a divisor. The resulting quotient is discarded, and the remainder is the CRC value for that particular stream of input data, which resides in the shift register after the last data bit has been shifted in. From the shift register implementation it is obvious that the final result (CRC value) is dependent, in a very complex way, on the past history of the bits presented. Therefore, it would take an extremely rare combination of errors to escape detection by this method.

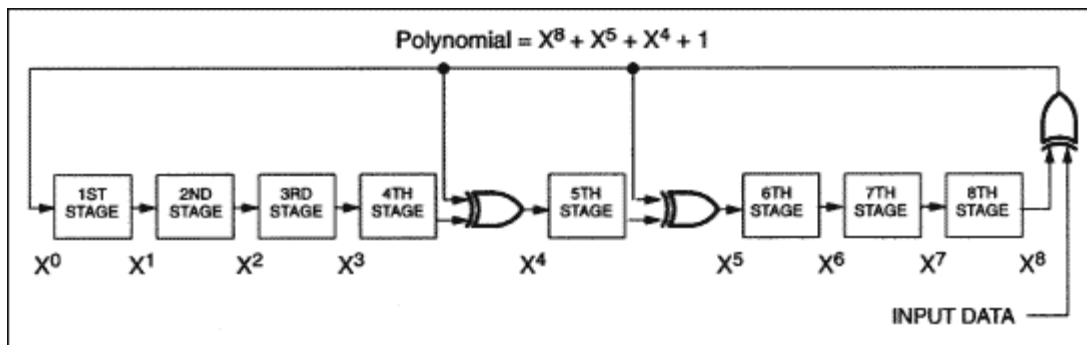


Figure 2. Maxim 1-Wire 8-bit CRC.

E.2.1) Modul 2 Binary Division Vs Traditional Division

The basic idea of CRC algorithm is to treat the transmitted data as a very long number of digits. Divide this number by another number. The resulting remainder is appended to the original data as check data. Also take the data from the above example:

6, 23, 4 can be seen as a binary number: 0000011000010111 00000010

If 9 is chosen by dividing, the binary representation is: 1001

Then the division operation can be expressed as:

```

      1010, 1101, 0011, 1001
1001 ) 0000, 0110, 0001, 0111, 0000, 0010
      100, 1
      ----
        1, 100
        1, 001
        ----
          111, 0
          100, 1
          ----
            10, 11
            10, 01
            ----
              1011
              1001
              ----
                10, 000
                1, 001
                ----
                  1110
                  1001
                  ----
                    101, 0
                    100, 1
                    ----
                      1010
                      1001
                      ----
                        0001

```

As you can see, the last remaining number is 1. If we use this remainder as a checksum, the data transferred is: 6, 23, 4, 1.

The CRC algorithm is a bit similar to this process, but it does not use the usual division in the example above. In the CRC algorithm, binary data streams are used as coefficients of the polynomial, followed by the multiplication and division of the polynomial. Let's give an example.

For example, we have two binary numbers: 1101 and 1011.

1101 is associated with the following polynomial: $1x^3+1x^2+0x^1+1x^0=x^3+x^2+x^0$

1011 is associated with the following polynomial: $1x^3+0x^2+1x^1+1x^0=x^3+x^1+x^0$

Multiplication of two polynomials: $(x^3+x^2+x^0)(x^3+x^1+x^0)=x^6+x^5+x^4+x^3+x^3+x^2+x^1+x^0$

When the result is obtained, the modulo 2 operation is used to merge the same items. That is, multiplication and division use normal polynomial multiplication and division, while addition and subtraction use modulo 2 operations. The so-called modulo 2 operation is to divide the result by 2 and take the remainder. For example, $3 \bmod 2 = 1$. Therefore, the resulting polynomial above is: $x^6+x^5+x^4+x^3+x^2+x^1+x^0$, corresponding to the binary number: 111111

Addition and subtraction with modulo 2 is actually an operation, which is what we usually call XOR:

$0+0=0$	$0-0=0$
$0+1=1$	$1-0=1$
$1+0=1$	$0-1=1$
$1+1=0$	$1-1=0$

As mentioned above, half-day polynomials, in fact, even without introducing the concept of polynomial multiplication and division, can explain the particularity of these operations. Only polynomials are mentioned in almost all the literature explaining the CRC algorithm, so a few basic concepts are simply written here. However, it is very tedious to always use this polynomial representation, and the following instructions will try to use a more concise way of writing.

The division operation is similar to the multiplication concept given above, or the addition and subtraction are replaced by XOR. Here is an example:

The data to be transferred is: 1101011011

The divisor is set to 10011

Before calculating, four 0:11010110000 are added to the back of the original data, so the reason for adding 0 is explained later.

```

      1100001010
10011 ) 11010110110000
      10011,.,.,.,.
      -----
      ,.,.,.,.
      10011,.,.,.,.
      10011,.,.,.,.
      -----
      ,.,.,.,.
      10110...
      10011...
      -----
      ...
      10100.
      10011.
      -----
      1110 = Remainder

```

From this example, it can be seen that after the addition and subtraction of module 2, the problem of borrowing does not need to be considered, so division becomes simpler. The final remainder is the CRC checkword. In order to perform the CRC operation, that is, this special division operation, a dividend must be specified. In the CRC algorithm, this divider has a special name called "Generate Polynomial". Selection of the resulting polynomial is a very difficult problem. If not, the probability of detecting errors will be much lower. Fortunately, this problem has been studied by experts for a long time. For those of us users, we just need to use the ready-made results.

E.2.2) CRC8Maxim $x^8+x^5+x^4 + 1$ Lookup Table

//8 bit CRC lookup table

const unsigned char crc_table[256] =

```
{ 0, 94, 188, 226, 97, 63, 221, 131, 194, 156, 126, 32, 163, 253, 31, 65, 157, 195, 33, 127, //19
  252, 162, 64, 30, 95, 1, 227, 189, 62, 96, 130, 220, 35, 125, 159, 193, 66, 28, 254, 160, //39
  225, 191, 93, 3, 128, 222, 60, 98, 190, 224, 2, 92, 223, 129, 99, 61, 124, 34, 192, 158, //59
  29, 67, 161, 255, 70, 24, 250, 164, 9, 121, 155, 197, 132, 218, 56, 102, 229, 187, 89, 7, //79
  219, 133, 103, 57, 186, 228, 6, 88, 25, 71, 165, 251, 120, 38, 196, 154, 101, 59, 217, 135, //99
  4, 90, 184, 230, 167, 249, 27, 69, 198, 152, 122, 36, 248, 166, 68, 26, 153, 199, 37, 123, //119
  58, 100, 134, 216, 91, 5, 231, 185, 140, 210, 48, 110, 237, 179, 81, 15, 78, 16, 242, 172, //139
  47, 113, 147, 205, 17, 79, 173, 243, 112, 46, 204, 146, 211, 141, 111, 49, 178, 236, 14, 80, //159
  175, 241, 19, 77, 206, 144, 114, 44, 109, 51, 209, 143, 12, 82, 176, 238, 50, 108, 142, 208, //179
  83, 13, 239, 177, 240, 174, 76, 18, 145, 207, 45, 115, 202, 148, 118, 40, 171, 245, 23, 73, //199
  8, 86, 180, 234, 105, 55, 213, 139, 87, 9, 235, 181, 54, 104, 138, 212, 149, 203, 41, 119, //219
  244, 170, 72, 22, 233, 183, 85, 11, 136, 214, 52, 106, 43, 117, 151, 201, 74, 20, 246, 168, //239
  116, 42, 200, 150, 21, 75, 169, 247, 182, 232, 10, 84, 215, 137, 107, 53 };
```

CRC linear f w $CRC(x \oplus y \oplus z) = CRC(x) \oplus CRC(y) \oplus CRC(z)$

E.2.3) CRC8 Dallas/Maxim Algorithm

Binary Multiplication

$$\begin{array}{r}
 101 \quad 5 \quad 111001 + 1 = 26 \quad 26 \bmod 5 = 1 \\
 \underline{\times 101} \quad \underline{\times 5} \\
 101 \quad 25 \\
 0000 \\
 \underline{10100} \\
 11001 = 25
 \end{array}$$

Binary Division

$$\begin{array}{r}
 \overline{101} \\
 101 \overline{) 11010} \\
 \underline{101} \\
 10 \\
 \underline{00} \\
 100 \\
 \underline{101} \\
 1 \text{ Remainder}
 \end{array}$$

$$26 \bmod 5 = 1$$

Modulo 2 Division XOR

Modulo-2 division is performed similarly to “normal” arithmetic division. The only difference is that we use modulo-2 subtraction (**XOR**) instead of arithmetic subtraction for calculating the remainders in each step. The quotient is not of interest.

$$\begin{array}{r}
 \overline{111} \\
 101 \overline{) 11010} \\
 \underline{101} \\
 111 \\
 \underline{101} \\
 100 \\
 \underline{101} \\
 1 \text{ Remainder}
 \end{array}$$

$$26 \bmod 5 = 1$$

E.2.4) Types of CRCs

There are different types of CRCs. They are categorized by the degree of the polynomial they use. As the first exponent of a polynomial of degree n is always present by definition (otherwise it would have a lower degree), its binary representation always begins with a 1.

In other words, the first bit of a binary polynomial representation doesn't carry any information about the polynomial when we agree on a fixed degree.

For that reason, the first bit of a binary polynomial representation is always dropped when computing a CRC in software. So the bit size of the resulting binary is always n for a polynomial of degree n .

It is apparent there is a myriad of CRC implementations and the sending and receiving devices must be using the same methodology. It is because of this non-standardization complexity (obfuscation) rules.

Example:

Polynomial	Binary Representation	Binary (1 st bit dropped)	Bit Size
$x^4 + x^2 + x + 1$	10111	0111	4
$x^4 + x^3 + x^2 + 1$	11101	1101	4
$x^8 + x^4 + x^2 + 1$	100010101	00010101	8

CRCs types are named by their bit size. Here are the most common ones:

CRC-8

CRC-16

CRC-32

CRC-64

CRC-1 (parity bit) is a special case

Generally, we can refer to a CRC as CRC- n , where n is the number of CRC bits and the number of bits of the polynomial's binary representation with a dropped first bit. Obviously, different CRCs are possible for the same n as multiple polynomials exist for the same degree.

E.2.4.1) Error Detection

How do we choose a suitable CRC and a respective polynomial? There are three things we need to consider:

Random Error Detection Accuracy

Burst Error Detection Accuracy

The Redundancy Factor

E.2.4.2) Random Error Detection Accuracy

Random errors are errors that can occur randomly in any data. For example, a single bit is flipped when transmitting data or a few bits are lost during the transmission.

Depending on the bit size of the CRC we use, we can detect most of these random errors. However, for a CRC- n , $1/2^n$ of these errors cannot be detected. The following table shows the percentage of the possible random errors that remain undetected for each CRC type:

CRC Type	Undetected Errors	% Undetected
CRC-8	$1/2^8$	0.39
CRC-16	$1/2^{16}$	0.0015
CRC-32	$1/2^{32}$	0.00000002
CRC-64	$1/2^{64}$	5.4×10^{-20}

E.2.4.3) Burst Error Detection Accuracy

Errors in data transmission are oftentimes not random but produced over a consecutive sequence of bits. Such errors are called *burst* errors. They are the most common errors in data communication. It's one of the CRC's strongest properties to detecting burst errors reliably.

A CRC- n can detect single burst errors with a maximum length of n bits. However, this depends a lot on the polynomial used for computing the CRC. Some polynomials are able to detect multiple bursts of errors in the transmitted data.

CRC Type	Burst Error Detection
CRC-8	at least a single burst of ≤ 8 bits
CRC-16	at least a single burst of ≤ 16 bits
CRC-32	at least a single burst of ≤ 32 bits
CRC-64	at least a single burst of ≤ 64 bits

E.2.4.4) The Redundancy Factor

Using a CRC for error detection comes at the cost of extra (non-meaningful) data. When we use a CRC-32 (4 bytes), we need to transmit two more bytes of "unnecessary" data as compared to a CRC-16. CRCs with a lower bit size are obviously cheaper with respect to storage space.

Based on these three factors, we can decide which CRC type to choose for our application. However, the polynomial you choose for your CRC also affects the efficiency and quality of your error detection. But that's a topic for itself and we won't cover it in this article. Fortunately, there are a couple of standard polynomials used for a particular CRC type and in most cases it makes sense to just use one of these.

E.2.5) CRC8Dallas\Maxim Algorithm (Rayman thanks for info)

- 1) Divisor is 10011001 n = 9 bit divisor (CRC bits = n-1)
- 2) Data bits are to be reversed ordered and add n-1 zeroes (8)
- 3) Perform Modul 2 division
- 4) Reverse order of remainder is the CRC8Dallas\Maxim

E.2.5.1_Example 1 \$C2

Data = \$C2 = %11000010 **ReverseC2** = > %01000011 = \$43 **Divisor** = 10011001

10011001	0100001100000000
	<u>10011001</u>
	111101000
	<u>10011001</u>
	110110010
	<u>10011001</u>
	100000110
	<u>10011001</u>
	01101110 => 0111011 reverse order 8 bit

CRCMaxim(\$C2) = %01110110 = \$76 = 118

E.2.5.2_Example \$BC

Data = \$BC = %10111100 **ReverseBC** = > 00111101 = \$3E **Divisor** = 100110001

```

100110001 | 0011110100000000
          100110001
          110110010
          100110001
          100000110
          100110001
          110111000
          100110001
          10001001 => 10010001 reverse order 8 bit
    
```

CRCMaxim(\$BC) = %10010001 = \$91 = 145

E.2.5.4 Example 4 \$7778797A

$$a) \text{CRC}(\$7778) = \text{CRC}(\text{CRC}(\$77) \oplus \$78)$$

$$\text{CRC}(\$77) \oplus \$78 = \$7B \oplus \$78 = 01111011$$

$$\begin{array}{r} 01111000 \\ \underline{} \end{array}$$

$$00000011 = \$03$$

$$\text{CRC}(\$7778) = \text{CRC}(\$03) = \$E2 = 226$$

$$b) \text{CRC}(\$777879) = \text{CRC}(\text{CRC}(\$7778) \oplus \$79)$$

$$\text{CRC}(\$7778) \oplus \$79 = \$E2 \oplus \$79 = 11100010$$

$$\begin{array}{r} 01111001 \\ \underline{} \end{array}$$

$$10011011 = \$9B$$

$$\text{CRC}(\$777879) = \text{CRC}(\$9B) = \$31 = 49$$

$$c) \text{CRC}(\$7778797A) = \text{CRC}(\text{CRC}(\$777879) \oplus \$7A)$$

$$\text{CRC}(\$777879) \oplus \$7A = \$31 \oplus \$7A = 00110001$$

$$\begin{array}{r} 01111010 \\ \underline{} \end{array}$$

$$01001011 = \$4B$$

$$\text{CRC}(\$7778797A) = \text{CRC}(\$4B) = \$66 = 102$$

E.2.6) Example CRC Calculator (source Chris Gadd)

```

{E.2.6.1_Example_CRC_Calculator}
{{
    CRC calculator
}}
CON
    _CLKFREQ = 20_000_000

VAR
    long ptr

DAT
    ' Family code      CRC
    ' |  └─Serial number┘ |
    '  ? ?              ? ?
ds1822_1 byte  "22_B6_1B_3F_00_00_00_FE", $00
ds1822_2 byte  "22_5A_0F_3F_00_00_00_C9", $00

test_data byte  "77_78_79_7A", $00

PUB main() | poly, crc

    debug(" ")
    debug("The find_poly method is used to find the polynomial used to create a CRC")
    find_poly(@ds1822_2)
    crc := get_crc(@ds1822_2, $8C)
    debug(" ")
    debug("The get_crc method is used to verify a valid message: ")
    debug(uhex_byte(crc))

    debug(" ")
    debug("Test_data does not contain a valid CRC, and no eight-bit polynomial is able to match the
existing bytes")
    find_poly(@test_data)

    debug(" ")
    debug("A CRC can be found using the get_crc method")
    crc := get_crc(@test_data, $8C)
    debug(uhex_byte(crc))

    debug(" ")
    debug("Appending this value to test_data results in a valid message")
    ' find_poly(string("77_78_79_7A_66"))
    crc := get_crc(string("77_78_79_7A_66"), $8C)
    debug(uhex_byte(crc))

PUB find_poly(strPtr) : result | poly, crc, data    " Use this method when you have a complete message
with CRC, but don't know the polynomial

```

```

poly := %1000_0000          ' Initialize the polynomial
repeat until poly == %1111_1111 ' Repeat for all possible 8-bit polynomials
if (crc := get_crc(strPtr,poly)) == 0 ' A valid polynomial will result in crc being 0
    debug(uhex_byte(poly))
    result++
    poly += 1

if result == 0
    debug("None found")

PUB get_crc(strPtr,poly) : crc | data          " Use this method to find or verify a CRC
ptr := strPtr
repeat until (data := get_byte()) == -1      ' read all data bytes from string
    repeat 8                                ' repeat for all bits in each byte
        if crc & 1 <> data & 1                ' if lsb of crc not equal to lsb of data
            crc >>= 1                        ' shift crc right one
            crc ^= poly                      ' xor crc with polynomial
        else                                ' otherwise
            crc >>= 1                        ' just shift crc without xor
        data >>= 1                          ' put next bit of data into lsb (bytes are processed lsb 1st)

'    debug(ubin_byte(crc)," ",ubin_byte(data))
'    debug(" ")

PRI get_byte() : result | char                ' Convert two ASCII characters into a byte, ignores
underscore, returns -1 if null-terminator detected
repeat 2
    result <<= 4
    repeat until (char := byte[ptr++]) <> "_"
    if char == $00
        return -1
case char
    "0".. "9" : result |= char - "0"
    "A".. "F" : result |= char - "A" + $0A
    "a".. "f" : result |= char - "a" + $0A

```

E.2.7) Websites for CRC:

<https://rndtool.info/CRC-step-by-step-calculator/> dividend/divisor steps

<https://crccalc.com> CRC Calculator different values

<https://www.youtube.com/watch?v=izG7qT0EpBw> provides overview of how CRC derived

<https://quickbirdstudios.com/blog/validate-data-with-crc/> CRC8 polynomial generation dropping bit

Appendix “F” Hardware and Constants

F.1) Cog CPU

Each Cog CPU has a PC program Counter,

Each Cog CPU Has ALU Arithmetic Logic Unit

ALU has a result register C Carry Flag or Z Zero Flag

Each Cog CPU Has a “Q” register with 2 associated flags that can be set with SetQ and SetQ2

Q register has RDLONG\WRLONG to identify burts read/write to CogRam or LutRam

Each Cog CPU has AUGSx registers. For augmenting source field S (9 bit field +23 bits)

Each Cog CPU has AUGDx register. For augmenting destination D 9 bit field + 23 bits)

Note: It would be useful to have block diagram of the Cog CPU

F.2) Hardware Register

Variables (all LONG)	Variable Name	Address or Offset	Description	Useful in Spin2	Useful in Spin2- PASM	Useful in PASM- Only
Hub Locations	CLKMODE	\$00040	Clock mode value	Yes	Yes	No
	CLKFREQ	\$00044	Clock frequency value	Yes	Yes	No
Hub VAR	VARBASE	+0	Object base pointer, @VARBASE is VAR base, used by method-pointer calls	Maybe	No	No
Cog Registers	PR0	\$1D8	Spin2 <-> PASM communication	Yes	Yes	No
	PR1	\$1D9		Yes	Yes	No
	PR2	\$1DA		Yes	Yes	No
	PR3	\$1DB		Yes	Yes	No
	PR4	\$1DC		Yes	Yes	No
	PR5	\$1DD		Yes	Yes	No
	PR6	\$1DE		Yes	Yes	No
	PR7	\$1DF		Yes	Yes	No
	IJMP3	\$1F0	Interrupt JMP's and RET's	No	Yes	Yes
	IRET3	\$1F1		No	Yes	Yes
	IJMP2	\$1F2		No	Yes	Yes
	IRET2	\$1F3		No	Yes	Yes
	IJMP1	\$1F4	Pointer registers Data pointer passed from COGINIT	No	Yes	Yes
	IRET1	\$1F5		No	Yes	Yes
	PA	\$1F6		No	Yes	Yes
	PB	\$1F7		No	Yes	Yes
	PTRA	\$1F8	Output enables for P31..P0 Output enables for P63..P32 Output states for P31..P0 Output states for P63..P32 Input states from P31..P0 Input states from P63..P32	No	Yes	Yes
	PTRB	\$1F9		No	Yes	Yes
	DIRA	\$1FA		Yes	Yes	Yes
	DIRB	\$1FB		Yes	Yes	Yes
	OUTA	\$1FC		Yes	Yes	Yes
	OUTB	\$1FD		Yes	Yes	Yes
	INA	\$1FE		Yes	Yes	Yes
	INB	\$1FF		Yes	Yes	Yes

F.3) HUB Memory

Hub Memory is located in (and managed by) the Hub and is accessible to each cog, in a time-shared, round-robin fashion. It consists of Hub RAM and Hub ROM.

Hub RAM is **512 KB**, accessible as bytes, words, and longs. It holds your program, data, global variables, and stack space, which collectively make up your Propeller Application. Hub RAM is also used to share information between cogs or process larger blocks of data than will fit into Cog RAM.

The Hub ROM is 16 KB and holds read-only system resources such as the Boot Loader. It is loaded into the last **16 KB** of Hub RAM upon boot-up.

F.4) HUB Memory Spin2 Stack.

The Spin Interpreter implements a call stack to facilitate Spin method calling, parameter passing, expression evaluation, and returning method results.

The Propeller Application (if Spin2-based) has an automatically allocated stack located in Hub RAM immediately following the application's global variable memory. It expands and collapses as needed; growing towards higher addresses and shrinking towards lower addresses.

Spin methods that are manually launched into other cogs store their stack starting at the `StkAddr` address given by the `COGSPIN` command that launched them (usually inside a long array in variable space). Their stacks expand and contract in the same manner as with the Propeller Application stack. In both cases, the capacity of the stack (method nesting-depth, parameter list length, expression complexity, and return result length) is limited only by the amount of free memory available (for the application) or memory provided (by the developer).

F.5) DAT Blocks

DAT block symbols exist in Hub RAM, but if they are part of PASM2 code that is launched, they are also in Cog RAM where they are manipulated independently.

The DAT block itself is stored in the application image in Hub RAM. Spin2-based references to DAT symbols access the corresponding location and data in Hub RAM.

When a cog is launched with assembly code, any DAT symbols within 504 longs of the launch point are copied into Register RAM. Unlike with Spin2 code, PASM2 code that references those symbols accesses the corresponding Register RAM* locations (its local copy) instead of Hub RAM. In addition, those symbolic references are addressed as longs of Register RAM memory, regardless of how the symbol was actually declared.

* Or Lookup RAM, if the code launched into Register RAM manually loads PASM2+symbol code into, and executes code from, Lookup RAM.

The DAT block's purpose is to hold fixed data and Propeller 2 Assembly code for the application. Symbols may be included to reference this data and code.

DAT blocks are stored in the application image in Hub RAM. Just like with code in PUB and PRI blocks, there is only one instance of each DAT block in the running application, regardless of how many instances of the containing object there are. This means that Spin-based references to DAT symbols each access the same corresponding location and data in Hub RAM, regardless of which instance of that object is making the reference. This is handy to share memory between multiple instances of a Spin2 object.

When a cog is launched with assembly code, any DAT symbols within 504 longs of the launch point are copied into Register RAM. Unlike with Spin2 code, PASM2 code that references those symbols accesses the corresponding Register RAM* locations (its local copy) instead of Hub RAM. In addition, those symbolic references are addressed as longs of Register RAM memory, regardless of how the symbol was actually declared. In PASM2, no Hub RAM references can be made by simply using the declared symbolic name; instead, the absolute address of that symbol must be passed from the Spin2 object and used along with instructions like RDLONG and WRLONG.

There's nothing preventing the contents of DAT from being modified at runtime. This naturally leads to a special use—; "special values" may be defined in a DAT block that are easily referenced by every Spin2 object instance (and every new launch of PASM2 code) and can be modified at runtime to instantly change what each Spin2 instance (and future new PASM2 launched cogs) sees.

* Or Lookup RAM locations, if symbolic data were initially loaded in by the code running in Register RAM.

F.6) Propeller Electrical Specifications

Absolute Maximum Electrical Ratings Stresses in excess of the absolute maximum ratings can cause permanent damage to the device. These are absolute stress ratings only. Functional operation of the device is not implied at these or any other conditions in excess of those given. Exposure to absolute maximum ratings for extended periods can adversely affect device reliability.

Absolute Maximum Ratings	
Ambient temperature under bias	-40 °C to +125 °C
Storage temperature	-40 °C to +150 °C
Voltage on VDD with respect to GND	-0.3 V to +2.2 V
Voltage on Vxxyy with respect to GND	-0.3 V to +4.0 V
Voltage on all other pins with respect to GND ¹	-0.3 V to (Vxxyy + 0.3 V)
Total power dissipation	2.5 W
Max. current out of GND	4 A
Max. current into VDD pins	120 mA per pin
Max. current into Vxxyy pins	120 mA per pin
Max DC current into an input pin with internal protection diode forward biased	±10 mA
Max. allowable current per I/O pin	±30 mA
ESD Human Body Model (JS-001)	4 kV
ESD Charged Device Model (JS-002)	1 kV

¹ Note: I/O pin voltages in respect to GND may be exceeded if the internal protection diode forward bias current is not exceeded.

F.7) Built In Numeric Constants

Symbol Value	Symbol Name	Details
\$0000_0000	FALSE	Same as 0
\$FFFF_FFFF	TRUE	Same as -1
\$8000_0000	NEGX	Negative-extreme integer, -2_147_483_648 (\$8000_0000)
\$7FFF_FFFF	POSX	Positive-extreme integer, +2_147_483_647 (\$7FFF_FFFF)
\$4049_0FDB	PI	Single-precision floating-point value of Pi, 3.14159265

Appendix “G” Table of Operators

Operators

Below is a table of all the operators available for use in Spin2 methods. Compile-time expressions can use the unary, binary, ternary and float operators.

Var-Prefix Operators	Term (method only)	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Expr
++ (pre)	++var	1	++var	1	Pre-increment	
-- (pre)	--var	1	--var	1	Pre-decrement	
?? (pre)	??var	1	??var	1	Iterate long per XOR032, return pseudo-random	

Var-Postfix Operators	Term (method only)	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Expr
(post) ++	var++	1	var++	1	Post-increment	
(post) --	var--	1	var--	1	Post-decrement	
(post) !!	var!!	1	var!!	1	Post-logical NOT (0 → -1, non-0 → 0)	
(post) !	var!	1	var!	1	Post-bitwise NOT	
(post) \	var\x	1	var\x	1	Post-assign x	
(post) ~	var~	1	var~	1	Post-clear all bits	
(post) ~~	var~~	1	var~~	1	Post-set all bits	

Address Operators	Term (method only)	Priority (term)			Description	Float Expr
@	@symbol	1			Hub address of VAR/PUB/PRI variable or DAT symbol	
@	@method	1			Pointer to method, may be @object{[[]]}.method	
@@	@@x	1			Hub address of object + x, 'DAT x long @dat_symbol'	
#	#reg_symbol	1			Register address of cog/LUT DAT symbol	

Unary Operators	Term	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Expr
!!, NOT	!!x	12	!!= var	1	Logical NOT (0 → -1, non-0 → 0)	
!	!x	2	!= var	1	Bitwise NOT (1's complement)	
-	-x	2	-= var	1	Negate (2's complement)	✓
ABS	ABS x	2	ABS= var	1	Absolute value	✓
ENCOD	ENCOD x	2	ENCOD= var	1	Encode MSB, 0..31	
DECOD	DECOD x	2	DECOD= var	1	Decode, 1 << (x & \$1F)	
BMASK	BMASK x	2	BMASK= var	1	Bitmask, (2 << (x & \$1F)) - 1	
ONES	ONES x	2	ONES= var	1	Sum all '1' bits, 0..32	
SQRT	SQRT x	2	SQRT= var	1	Square root of unsigned value	
QLOG	QLOG x	2	QLOG= var	1	Unsigned value to logarithm {5'whole, 27'fraction}	
QEXP	QEXP x	2	QEXP= var	1	Logarithm to unsigned value	

Binary Operators	Term	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Expr
>>	x >> y	3	var >>= y	17	Shift x right by y bits, insert 0's	
<<	x << y	3	var <<= y	17	Shift x left by y bits, insert 0's	
SAR	x SAR y	3	var SAR= y	17	Shift x right by y bits, insert MSB's	
ROR	x ROR y	3	var ROR= y	17	Rotate x right by y bits	
ROL	x ROL y	3	var ROL= y	17	Rotate x left by y bits	
REV	x REV y	3	var REV= y	17	Reverse y LSBs of x and zero-extend	
ZEROX	x ZEROX y	3	var ZEROX= y	17	Zero-extend above bit y	
SIGNX	x SIGNX y	3	var SIGNX= y	17	Sign-extend from bit y	
&	x & y	4	var &= y	17	Bitwise AND	
^	x ^ y	5	var ^= y	17	Bitwise XOR	
	x y	6	var = y	17	Bitwise OR	
*	x * y	7	var *= y	17	Signed multiply	✓
/	x / y	7	var /= y	17	Signed divide, return quotient	✓
+/	x +/ y	7	var +/= y	17	Unsigned divide, return quotient	
//	x // y	7	var //= y	17	Signed divide, return remainder	
++	x ++ y	7	var ++= y	17	Unsigned divide, return remainder	
SCA	x SCA y	7	var SCA= y	17	Unsigned scale, (x * y) >> 32	
SCAS	x SCAS y	7	var SCAS= y	17	Signed scale, (x * y) >> 30	
FRAC	x FRAC y	7	var FRAC= y	17	Unsigned fraction, (x << 32) / y	
+	x + y	8	var += y	17	Add	✓
-	x - y	8	var -= y	17	Subtract	✓
#>	x #> y	9	var #>= y	17	Force x > y, signed	✓
<#	x <# y	9	var <#= y	17	Force x <= y, signed	✓
ADDBITS	x ADDBITS y	10	var ADDBITS= y	17	Make bitfield, (x & \$1F) (y & \$1F) << 5	
ADDPINS	x ADDPINS y	10	var ADDPINS= y	17	Make pinfield, (x & \$3F) (y & \$1F) << 6	
<	x < y	11			Signed less than (returns 0 or -1)	✓
+<	x +< y	11			Unsigned less than (returns 0 or -1)	
<=	x <= y	11			Signed less than or equal (returns 0 or -1)	✓
+<=	x +<= y	11			Unsigned less than or equal (returns 0 or -1)	
=	x = y	11			Equal (returns 0 or -1)	✓
<>	x <> y	11			Not equal (returns 0 or -1)	✓
>=	x >= y	11			Signed greater than or equal (returns 0 or -1)	✓
+>=	x +>= y	11			Unsigned greater than or equal (returns 0 or -1)	
>	x > y	11			Signed greater than (returns 0 or -1)	✓
+>	x +> y	11			Unsigned greater than (returns 0 or -1)	
<=>	x <=> y	11			Signed comparison (<,> returns -1,0,1)	✓
&&, AND	x && y	13	var &&= y	17	Logical AND (x <> 0 AND y <> 0, returns 0 or -1)	
^^, XOR	x ^^ y	14	var ^^= y	17	Logical XOR (x <> 0 XOR y <> 0, returns 0 or -1)	
, OR	x y	15	var = y	17	Logical OR (x <> 0 OR y <> 0, returns 0 or -1)	

Ternary Operator	Term	Priority (term)			Description	Float Expr
<code>? :</code>	<code>x ? y : z</code>	16			If <code>x <> 0</code> then choose <code>y</code> , else choose <code>z</code>	
Assign Operator			Assign (method only)	Priority (assign)	Description	Float Expr
<code>:=</code>			<code>var := x</code> <code>v1, v2 := x, y</code>	17	Set <code>var</code> to <code>x</code> Set <code>v1</code> to <code>x</code> , set <code>v2</code> to <code>y</code> , etc. (<code>'_'</code> on left = ignore)	
Equate Operator			Assign (CON block only)	Priority (equate)	Description	Float Expr
<code>=</code>			<code>symbol = x</code>	17	Set <code>symbol</code> to <code>x</code> in CON block	
Float Operators	Term (constant only)				Description	Float Expr
<code>FLOAT ()</code>	<code>FLOAT (x)</code>				Convert integer <code>x</code> to float	✓
<code>ROUND ()</code>	<code>ROUND (x)</code>				Convert float <code>x</code> to rounded integer	✓
<code>TRUNC ()</code>	<code>TRUNC (x)</code>				Convert float <code>x</code> to truncated integer	✓

Appendix “H” Table of Built In Methods

Hub Methods	Details
HUBSET(Value)	Execute HUBSET instruction using Value
CLKSET(NewCLKMODE, NewCLKFREQ)	Safely establish new clock settings, updates CLKMODE and CLKFREQ
COGSPIN(CogNum, Method({Pars}), StkAddr)	Start Spin2 method in a cog, returns cog's ID if used as an expression element, -1 = no cog free
COGINIT(CogNum, PASMaddr, PTRaValue)	Start PASM code in a cog, returns cog's ID if used as an expression element, -1 = no cog free
COGSTOP(CogNum)	Stop cog CogNum
COGID() : CogNum	Get this cog's ID
COGCHK(CogNum) : Running	Check if cog CogNum is running, returns -1 if running or 0 if not
LOCKNEW() : LockNum	Check out a new LOCK from inventory, LockNum = 0..15 if successful or < 0 if no LOCK available
LOCKRET(LockNum)	Return a certain LOCK to inventory
LOCKTRY(LockNum) : LockState	Try to capture a certain LOCK, LockState = -1 if successful or 0 if another cog has captured the LOCK
LOCKREL(LockNum)	Release a certain LOCK
LOCKCHK(LockNum) : LockState	Check a certain LOCK's state, LockState[31] = captured, LockState[3:0] = current or last owner cog
COGATN(CogMask)	Strobe ATN input(s) of cog(s) according to 16-bit CogMask
POLLATN() : AtnFlag	Check if this cog has received an ATN strobe, AtnFlag = -1 if ATN strobed or 0 if not strobed
WAITATN()	Wait for this cog to receive an ATN strobe

Pin Methods	Details
PINW PINWRITE(PinField, Data)	Drive PinField pin(s) with Data
PINL PINLOW(PinField)	Drive PinField pin(s) low
PINH PINHIGH(PinField)	Drive PinField pin(s) high
PINT PINTOGGLE(PinField)	Drive and toggle PinField pin(s)
PINF PINFLOAT(PinField)	Float PinField pin(s)
PINR PINREAD(PinField) : PinStates	Read PinField pin(s)
PINSTART(PinField, Mode, Xval, Yval)	Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1
PINCLEAR(PinField)	Clear PinField smart pin(s): DIR=0, then WRPIN=0
WRPIN(PinField, Data)	Write 'mode' register(s) of PinField smart pin(s) with Data
WXPIN(PinField, Data)	Write 'X' register(s) of PinField smart pin(s) with Data
WYPIN(PinField, Data)	Write 'Y' register(s) of PinField smart pin(s) with Data
AKPIN(PinField)	Acknowledge PinField smart pin(s)
RDPIN(Pin) : Zval	Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN data
RQPIN(Pin) : Zval	Read Pin smart pin without acknowledge, Zval[31] = C flag from RQPIN, other bits are RQPIN data

Timing Methods	Details
GETCT() : Count	Get 32-bit system counter
POLLCT(Tick) : Past	Check if system counter has gone past 'Tick', returns -1 if past or 0 if not past
WAITCT(Tick)	Wait for system counter to get past 'Tick'
WAITUS(Microseconds)	Wait Microseconds, uses CLKFREQ
WAITMS(Milliseconds)	Wait Milliseconds, uses CLKFREQ
GETSEC() : Seconds	Get seconds since booting, uses 64-bit system counter and CLKFREQ, rolls over every 136 years.
GETMS() : Milliseconds	Get milliseconds since booting, uses 64-bit system counter and CLKFREQ, rolls over every 49.7 days.

PASM interfacing	Details
CALL(RegOrHubAddr)	CALL PASM code at Addr, PASM code should avoid registers \$130..\$1D7 and LUT
REGEEXEC(HubAddr)	Load a self-defined chunk of PASM code at HubAddr into registers and CALL it. See REGEEXEC description.
REGLOAD(HubAddr)	Load a self-defined chunk of PASM code or data at HubAddr into registers. See REGLOAD description.

Math Methods	Details
ROTXY(x, y, angle32bit) : rotx, roty	Rotate (x,y) by angle32bit and return rotated (x,y)
POLXY(length, angle32bit) : x, y	Convert (length, angle32bit) to (x,y)

XYPOL(x, y) : length, angle32bit	Convert (x,y) to (length, angle32bit)
QSIN(length, angle, twopi) : y	Rotate (length,0) by (angle / twopi) * 2Pi and return y. Use 0 for twopi = \$1_0000_0000. Twopi is unsigned.
QCOS(length, angle, twopi) : x	Rotate (length,0) by (angle / twopi) * 2Pi and return x. Use 0 for twopi = \$1_0000_0000. Twopi is unsigned.
MULDIV64(mult1,mult2,divisor) : quotient	Divide the 64-bit product of 'mult1' and 'mult2' by 'divisor', return quotient (unsigned operation)
GETRND() : Rnd	Get random long (from xoroshiro128** PRNG, seeded on boot with thermal noise from ADC)

Memory Methods	Details
GETREGS(HubAddr, CogAddr, Count)	Move Count registers at CogAddr to longs at HubAddr
SETREGS(HubAddr, CogAddr, Count)	Move Count longs at HubAddr to registers at CogAddr
BYTEMOVE(Dest, Source, Count)	Move Count bytes from Source to Dest
WORDMOVE(Dest, Source, Count)	Move Count words from Source to Dest
LONGMOVE(Dest, Source, Count)	Move Count longs from Source to Dest
BYTEFILL(Dest, Value, Count)	Fill Count bytes at Dest with Value
WORDFILL(Dest, Value, Count)	Fill Count words at Dest with Value
LONGFILL(Dest, Value, Count)	Fill Count longs at Dest with Value

String Methods	Details
STRSIZE(Addr) : Size	Count bytes in zero-terminated string at Addr, return string size, not including zero terminator
STRCOMP(AddrA,AddrB) : Match	Compare zero-terminated strings at AddrA and AddrB, return -1 if match or 0 if mismatch
STRING("Text",9) : StringAddress	Compose a zero-terminated string (quoted characters and values 1..255 allowed), return address of string

Index ↔ Value Methods	Details
LOOKUP (Index: v1, v2..v3, etc) : Value	Lookup value (values and ranges allowed) using 1-based index, return value (0 if index out of range)
LOOKUP2 (Index: v1, v2..v3, etc) : Value	Lookup value (values and ranges allowed) using 0-based index, return value (0 if index out of range)
LOOKDOWN (Value: v1, v2..v3, etc) : Index	Determine 1-based index of matching value (values and ranges allowed), return index (0 if no match)
LOOKDOWN2 (Value: v1, v2..v3, etc) : Index	Determine 0-based index of matching value (values and ranges allowed), return index (0 if no match)

Appendix “I” Hub Operation

I.1) Hub RAM

The globally-accessible Hub RAM can be read and written as bytes, words, and longs, in little-endian format. Specifically, **little-endian** is when the least significant bytes are stored before the more significant bytes, and **big-endian** is when the most significant bytes are stored before the less significant bytes. **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 starting at any hub address, as well as execute **PASM2** instructions (longs) from any hub address starting at **\$400**. The last 16 KB of Hub RAM is normally addressable at both its normal address range, as well as at **\$FC000..\$FFFFFF**. This provides a stable address space (regardless of future Propeller 2 variations) for the **16 KB of internal ROM** which gets cached into the **last 16 KB of Hub RAM** on startup.

This **upper 16 KB** mapping is also used by the **cog debugging** scheme. The last 16 KB of RAM can be hidden from its normal address range and made read-only at **\$FC000..\$FFFFFF**. This is useful for making the last 16 KB of RAM persistent, **like ROM**. It is also how debugging is realized, as the RAM mapped to **\$FC000..\$FFFFFF** can still be written to while **executing** code 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.

Cog-to-Hub RAM Interface **Hub RAM** consists of 32-bit-wide single-port RAMs with byte-level write controls. This RAM is split into slices (one per cog) that are multiplexed among all cogs. On the Propeller 2 (P2X8C4M64P), each RAM **slice** holds **every 8th long** in the composite **Hub RAM**. Upon **every clock** cycle, each cog can access the **"next" RAM slice**, allowing for **continuous bidirectional streaming** of sequential Hub RAM longs.

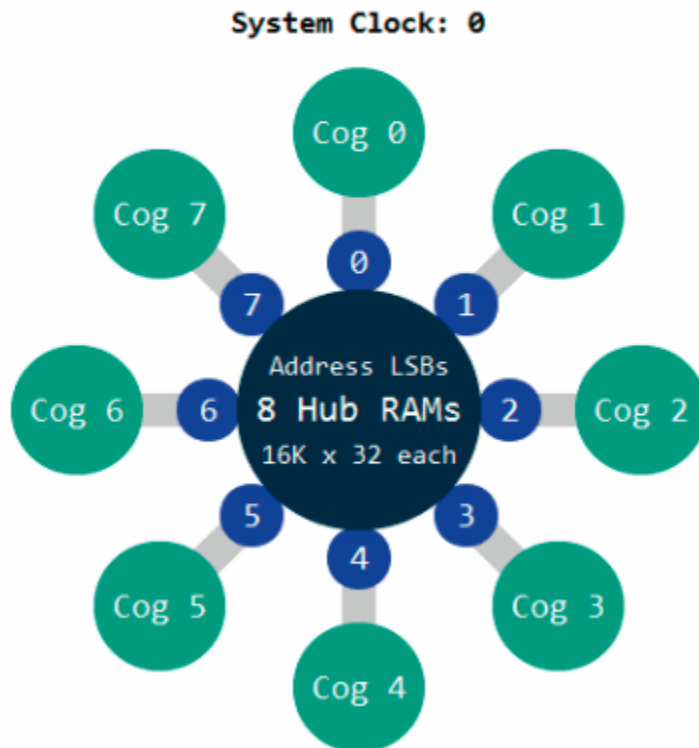
The Hub RAM Interface diagram illustrates this process conceptually as the collective of RAM slices rotates around, each facing a new cog every clock cycle. When a cog wants to read or write the Hub RAM, it must wait up to #cogs-1 clocks to access the initial RAM slice of interest. Once that occurs, subsequent locations (**slices**) can be accessed on **every clock**, thereafter, for continuous reading or writing of 32-bit longs. Normally, if the cog chooses not to access the next available location upon the next clock, it must once again wait up to 7 clocks to re-align with the desired slice.

However, each cog has an optional hub FIFO interface that smooths out data flow for less than 32-bits-per-clock access. This hub FIFO interface can be set for hub-RAM-read or hub-RAM-write operation to allow 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. Regardless of the transfer frequency or the word size, the FIFO will ensure that the cog's reads or writes are all properly conducted from/to the composite Hub RAM.

1.2) COG HUB Access

P2 Hub RAM Interface

Every cog can read/write 32 bits per clock



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..\$FFFF)
- Streamer usage (background transfers from Hub RAM → pins/DACs, or from pins/ADCs → Hub RAM)
- Software usage (fast sequential-reading or sequential-writing instructions)

For streamer or software usage, FIFO operation must be established by a RFAST or WFAST instruction executed from Cog RAM (Register/Lookup, \$00000..\$003FF). After that, and while remaining in Cog RAM, the streamer can be enabled to begin moving data in the background, or the two-clock RFxxx/WFxxx instructions can be used to manually read and write sequential data.

The FIFO contains (#cogs+11) stages. When in read mode, the FIFO loads continuously whenever less than (#cogs+7) stages are filled, after which point, up to 5 more longs may stream in, potentially filling all stages. These metrics ensure that the FIFO never underflows, under all potential reading scenarios.

1.3) HUB RAM Slice

The hub is made of 8 contiguously-mapped blocks of 16K longs (4 bytes, 2 words, or 32 bits wide). Each cog can access the next block on each clock, so that after 8 clocks the same block is again accessible. This allows for an ever-ascending address sequence for reading and writing longs on each clock, for each cog. Addressing is arranged so that the lowest 2 bits of address can select a byte within a block's long, the next higher 3 bits account for the block number, and the remaining upper 15 bits are fed to the address bus of each block. This makes a total of 20 bits, which allows for 1MB of hub address space (assuming 32K-long blocks), although the current P2 only implements 512KB, hence the use of 16k-long blocks.

Here is the block sequence, in terms of hub address, that each cog can hop onto and off of for contiguous long reading and writing:

```
%xxxx_xxxx_xxxx_xxx0_00xx
%xxxx_xxxx_xxxx_xxx0_01xx
%xxxx_xxxx_xxxx_xxx0_10xx
%xxxx_xxxx_xxxx_xxx0_11xx
%xxxx_xxxx_xxxx_xxx1_00xx
%xxxx_xxxx_xxxx_xxx1_01xx
%xxxx_xxxx_xxxx_xxx1_10xx
%xxxx_xxxx_xxxx_xxx1_11xx
<repeat>
```

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 hidden 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.

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..\$FFFFFF)
- 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 DDDDDDDDDD SSSSSSSSS   RDFAST D/#,S/#
EEEE 1100100 0LI DDDDDDDDDD SSSSSSSSS   WRFAST D/#,S/#
EEEE 1100100 1LI DDDDDDDDDD 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 RFAST 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_CBBBBBBB_BAAAAAAA %SSSSSSSS_SSSSCCCC_CBBBBBBB_BAAAAAAA
%1AAAAAAA	%1BBBBBBB	%1CCCCCCC	%SDDDDDDD	%000SDDDD_DDDCCCCC_CBBBBBBB_BAAAAAAA %SSSSDDDD_DDDCCCCC_CBBBBBBB_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 SSSSSSSS	RDBYTE D,S/#/PTRx {WC/WZ/WCZ}
EEEE 1010111 CZI DDDDDDDDD SSSSSSSS	RDWORD D,S/#/PTRx {WC/WZ/WCZ}
EEEE 1011000 CZI DDDDDDDDD SSSSSSSS	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 SSSSSSSS	WRBYTE D/#,S/#/PTRx
EEEE 1100010 1LI DDDDDDDDD SSSSSSSS	WRWORD D/#,S/#/PTRx
EEEE 1100011 0LI DDDDDDDDD SSSSSSSS	WRLONG D/#,S/#/PTRx
EEEE 1010011 11I DDDDDDDDD SSSSSSSS	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..\$FFFFFF - 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 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
##%0000000000000000AAAAAAAAAA_AAAAAAAAAA	- AUGS, 20-bit immediate address
##%000000001SUPNNNNNNNNNNNN_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 PTRx, 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	

10000000	PTRA	'use PTRA
11000000	PTRB	'use PTRB
10110001	PTRA++	'use PTRA, PTRA += SCALE
11110001	PTRB++	'use PTRB, PTRB += SCALE
10111111	PTRA--	'use PTRA, PTRA -= SCALE
11111111	PTRB--	'use PTRB, PTRB -= SCALE
10100001	++PTRA	'use PTRA + SCALE, PTRA += SCALE
11100001	++PTRB	'use PTRB + SCALE, PTRB += SCALE
10101111	--PTRA	'use PTRA - SCALE, PTRA -= SCALE
11101111	--PTRB	'use PTRB - SCALE, PTRB -= SCALE
100NNNNN	PTRA[INDEX]	'use PTRA + INDEX*SCALE
110NNNNN	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 PTRB into D

```
1111 1010110 001 DDDDDDDDDD 100000000 RDBYTE D,PTRB
```

Write lower word in D to PTRB - 7*2

```
1111 1100010 101 DDDDDDDDDD 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 PTRB into D, PTRB -= 1*2

```
1111 1010111 001 DDDDDDDDDD 101111111 RDWORD D,PTRB--
```

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

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

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

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

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

```
1111 1100010 001 DDDDDDDDDD 101011111 WRBYTE D,PTRB++[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 follows the SETQ/SETQ2 instruction, but will remain single-long scaled.

Appendix “J” System Clock

The system clock is the time base for all internal components and can be configured in several ways.

- Direct from internal slow clock (RCSLOW); a ~20 kHz oscillator is intended for low-power operation
- Direct from internal fast clock (RCFAST); a 20 MHz+ oscillator designed for minimum 20 MHz operation
- Direct from XI pin; driven externally via a clock oscillator or a crystal oscillator
- PLL-modified XI pin; driven externally via a clock oscillator or a crystal oscillator and the signal internally modified by the PLL (phase-locked loop), usually to multiple to a much higher frequency

The system clock is configured by the running Propeller 2 application using the HUBSET instruction in this format:

HUBSET ##%0000_000E_DDDD_DDMM_MMMM_MMMM_PPPP_CCSS 'set clock mode

The bit fields (E, D, M, P, C, and S) are described in the following tables.

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 <i>multiplying</i> the divided XI frequency (per %DDDDDD) inside the VCO. The VCO frequency should be kept within 100MHz to 350MHz.
%PPPP	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	VCO / 2 VCO / 4 VCO / 6 VCO / 8 VCO / 10 VCO / 12 VCO / 14 VCO / 16 VCO / 18 VCO / 20 VCO / 22 VCO / 24 VCO / 26 VCO / 28 VCO / 30 VCO / 1	This divided VCO frequency 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 pin
%01	RCSLOW	~20 kHz, can be switched to at any time, low-power
%00	RCFAST	20 MHz+, can be switched to at any time, used on boot-up.

WARNING: Incorrectly switching away from the PLL setting (%SS = %11) can cause a glitch which will hang the clock circuit. In order to safely switch, always start by switching to an internal oscillator using either HUBSET #F0 (for RCFAST) or HUBSET #F1 (for RCSLOW).

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

$$VCO = \frac{Freq(XI) \times (\%MMMMMMMM + 1)}{(\%DDDDDD + 1)}$$

$$PLL = if(\%PPPP = 15) \Rightarrow VCO$$

$$PLL = if(\%PPPP \neq 15) \Rightarrow \frac{VCO}{(\%PPPP + 1) \times 2}$$

Let's say you have a 20 MHz crystal attached to XI and XO and you want to run the Prop2 at 148.5 MHz. You could divide the crystal by 40 (%DDDDDD = 39) to get a 500 kHz reference, then multiply that by 297 (%MMMMMMMM = 296) in the VCO to get 148.5 MHz. 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 15 pf crystal mode and the PLL. In order to realize this clock setting, though, it must be done over a few steps:

HUBSET #F0	'set 20 MHz+ (RCFAST) mode
HUBSET ##%1_100111_0100101000_1111_10_00	'enable crystal+PLL, stay in RCFAST 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 148.5 MHz

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.

Appendix “K” Locks (HUB Cog Memory Access)

Locks For application-defined cog coordination, the hub provides a pool of 16 semaphore bits, called locks. Cogs may use locks, for example, to manage exclusive access of a resource or to represent an exclusive state, shared among multiple cogs. What a lock represents is completely up to the application using it; they are a means allowing one cog at a time the exclusive status of 'owner' of a particular lock ID. In order to be useful, all participant cogs must agree on a lock's ID and what purpose it serves. The LOCK instructions are:

LOCKNEW D {WC}

LOCKRET {#}D

LOCKTRY {#}D {WC}

LOCKREL {#}D {WC}

Lock Usage In order to use a lock, one cog must first allocate a lock with LOCKNEW and communicate that lock's ID with other cooperative cogs. Cooperative cogs then use LOCKTRY and LOCKREL to respectively take or release ownership of the state which that lock represents. If the lock is no longer needed by the application, it may be returned to the unallocated lock pool by executing LOCKRET. A cog may allocate more than one lock.

At any time, a cog may attempt to own a lock (ie: the state that lock represents) by using LOCKTRY. The Hub grants or denies ownership in response, ensuring that, at most, one cog owns the lock at any time. If a cog is granted ownership, it can perform the task defined for that lock and then use LOCKRET to release ownership, allowing any other cog to attempt ownership. Only the cog that has taken ownership of the lock can release it; however, a lock will also be implicitly released if the owner cog is stopped (COGSTOP) or restarted (COGINIT).

Appendix “L” Cordic Solver (HUB contains “Coordinate Rotation Digital Computer”)

The Hub contains a 54-stage pipelined CORDIC solver (Coordinate Rotation Digital Computer) 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 unsigned value with 32-bit result
- 32-bit signed (X, Y) rotation around (0, 0) by a 32-bit angle with 32-bit signed (X, Y) results
- 32-bit signed (X, Y) to 32-bit (length, angle) cartesian to polar operation
- 32-bit (length, angle) to 32-bit signed (X, Y) polar to cartesian operation
- 32-bit unsigned integer to 5:27-bit logarithm
- 5:27-bit logarithm to 32-bit unsigned integer

Each cog can issue one CORDIC instruction per its hub access window (which occurs once every eight clocks) and retrieve the result 55 clocks later via the GETQX and GETQY instructions. For faster throughput cogs can take advantage of the hub access window and CORDIC pipeline to issue a stream of CORDIC instructions interleaved with retrieving corresponding results.

Multiply Use the QMUL instruction to multiply two unsigned 32-bit numbers together and retrieve the CORDIC results with the GETQX and GETQY instructions (for lower and upper long, respectively).

Divide Use the QDIV or QFRAC instruction (either with optional preceding SETQ instruction) to divide a 64-bit numerator by a 32-bit denominator, then retrieve the CORDIC results with the GETQX and GETQY instructions (for quotient and remainder, respectively).
Square Root Use the QSQRT instruction on a 64-bit number and retrieve the square root CORDIC result with the GETQX instruction.

(X, Y) Rotation Use the SETQ instruction followed by the QROTATE instruction to rotate a 32-bit signed Y and X point pair by an unsigned 32-bit angle and retrieve the CORDIC results with the GETQX and GETQY instructions for X and Y, respectively.

(X, Y) to (length, angle) Use the QVECTOR instruction to convert a (X, Y) cartesian coordinate into (length, angle) polar coordinate and retrieve the CORDIC results with the GETQX and GETQY instructions (for length and angle, respectively).

(length, angle) to (X, Y) Use the QROTATE instruction to convert a (length, angle) polar coordinate into (X, Y) cartesian coordinate and retrieve the CORDIC results with the GETQX and GETQY instructions (for X and Y, respectively).

Logarithm Use the QLOG instruction on an unsigned 32-bit integer and retrieve the 5:27-bit logarithm CORDIC result (5-bit exponent and 27-bit mantissa) with the GETQX instruction.

Exponent Use the QEXP instruction on a 5:27-bit logarithm and retrieve the unsigned 32-bit integer CORDIC result with the GETQX instruction.

Multiply

Use the `QMUL` instruction to multiply two unsigned 32-bit numbers together and retrieve the CORDIC result with the `GETQX` and `GETQY` instructions (for lower and upper long, respectively). `QMUL` will wait for the hub access window and `GETQX` / `GETQY` will wait for the CORDIC results.

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

To get the results (these instructions wait for the CORDIC results):

```
GETQX   lower_long
GETQY   upper_long
```

Divide

Use the `QDIV` or `QFRAC` instruction (either with optional preceding `SETQ` instruction) to divide a 64-bit numerator by a 32-bit denominator, then retrieve the CORDIC results with the `GETQX` and `GETQY` instructions (for quotient and remainder, respectively). `QDIV` / `QFRAC` will wait for the hub access window and `GETQX` / `GETQY` will wait for the CORDIC results.

```
QDIV    D/#,S,#           - Divide {S: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
```

...and to get the results:

```
GETQX   quotient
GETQY   remainder
```

Square Root

Use the `QSQRT` instruction on a 64-bit number and retrieve the square root CORDIC result with the `GETQX` instruction. `QSQRT` will wait for the hub access window and `GETQX` will wait for the CORDIC results.

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

Rotation

Use the `SETQ` instruction followed by the `QROTATE` instruction to rotate a 32-bit signed Y and X point pair by an unsigned 32-bit angle and retrieve the CORDIC results with the `GETQX` and `GETQY` instructions for X and Y, respectively. For the angle (in S), `$00000000..$FFFFFFFF = 0..359.9999999` degrees. `QROTATE` will wait for the hub access window and `GETQX` / `GETQY` will wait for the CORDIC results.

```
SETQ    Q/#               - Set Y
QROTATE D/#,S,#           - Rotate (D,Q) by S
GETQX   X
GETQY   Y
```

Cartesian to Polar

Use the `QVECTOR` instruction to convert a (X, Y) cartesian coordinate into (length, angle) polar coordinate and retrieve the CORDIC results with the `GETQX` and `GETQY` instructions (for length and angle, respectively). `QVECTOR` will wait for the hub access window and `GETQX` / `GETQY` will wait for the CORDIC results.

```
QVECTOR D/#, S, #      - (X=D, Y=S) cartesian into (length, angle) polar
GETQX   length
GETQY   angle
```

Polar to Cartesian

Use the `QROTATE` instruction to convert a (length, angle) polar coordinate into (X, Y) cartesian coordinate and retrieve the CORDIC results with the `GETQX` and `GETQY` instructions (for X and Y, respectively). For the angle (in S), $\$00000000..\$FFFFFFF = 0..359.9999999$ degrees. `QROTATE` will wait for the hub access window and `GETQX` / `GETQY` will wait for the CORDIC results.

```
QROTATE D/#, S, #      - Rotate (D, $00000000) by S
GETQX   X
GETQY   Y
```

Note this is just like an [X,Y Rotation](#), but with Y set to 0 (by omitting the leading `SETQ`).

Integer to Logarithm

Use the `QLOG` instruction on an unsigned 32-bit integer and retrieve the 5:27-bit logarithm CORDIC result (5-bit exponent and 27-bit mantissa) with the `GETQX` instruction. `QLOG` will wait for the hub access window and `GETQX` will wait for the CORDIC results.

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

Logarithm to Integer

Use the `QEXP` instruction on a 5:27-bit logarithm and retrieve the unsigned 32-bit integer CORDIC result with the `GETQX` instruction. `QEXP` will wait for the hub access window and `GETQX` will wait for the CORDIC results.

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


Appendix “M” Pixel Operations (DVI/HDMI)

M.0.1) DVI/HDMI

M.0.1.) DVI Digital Visual Interface is a video display interface developed by the [Digital Display Working Group](#) (DDWG). The [digital](#) interface is used to connect a video source, such as a [video display controller](#), to a [display device](#), such as a [computer monitor](#). It was developed with the intention of creating an industry standard for the transfer of digital video content.

This interface is designed to transmit [uncompressed](#) digital video and can be configured to support multiple modes such as DVI-A (analog only), DVI-D (digital only) or DVI-I (digital and analog). Featuring support for analog connections, the DVI specification is compatible with the [VGA](#) interface.^[4] This compatibility, along with other advantages, led to its widespread acceptance over competing digital display standards [Plug and Display](#) (P&D) and [Digital Flat Panel](#) (DFP).^[2] Although DVI is predominantly associated with computers, it is sometimes used in other consumer electronics such as [television sets](#) and [DVD players](#).

M.0.2) HDMI High-Definition Multimedia Interface is a [proprietary](#) audio/video [interface](#) for transmitting [uncompressed video](#) data and compressed or uncompressed [digital audio](#) data from an HDMI-compliant source device, such as a [display controller](#), to a compatible [computer monitor](#), [video projector](#), [digital television](#), or [digital audio](#) device.^[4] HDMI is a digital replacement for [analog video](#) standards.

HDMI implements the [EIA/CEA-861](#) standards, which define video formats and waveforms, transport of compressed and uncompressed [LPCM](#) audio, auxiliary data, and implementations of the [VESA EDID](#).^{[5][6]: p. III} CEA-861 signals carried by HDMI are electrically compatible with the CEA-861 signals used by the [Digital Visual Interface](#) (DVI). No signal conversion is necessary, nor is there a loss of video quality when a DVI-to-HDMI adapter is used.^{[6]: §C} The [CEC](#) (Consumer Electronics Control) capability allows HDMI devices to control each other when necessary and allows the user to operate multiple devices with one handheld [remote control](#) device.^{[6]: §6.3}

Several versions of HDMI have been developed and deployed since the initial release of the technology, but all use the same cable and connector. Other than improved audio and video capacity, performance, resolution and color spaces, newer versions have optional advanced features such as [3D](#), [Ethernet](#) data connection, and CEC (Consumer Electronics Control) extensions.

Production of consumer HDMI products started in late 2003.^[7] In Europe, either DVI-[HDCP](#) or HDMI is included in the [HD ready](#) in-store labeling specification for TV sets for HDTV, formulated by [EICTA](#) with [SES Astra](#) in 2005. HDMI began to appear on [consumer HDTVs](#) in 2004 and [camcorders](#) and [digital still cameras](#) in 2006.^{[8][9]} As of January 6, 2015 (twelve years after the release of the first HDMI specification), over 4 billion HDMI devices have been sold.^[10]

"DVI is the accepted standard for transferring serially uncompressed data at high speeds between a PC host and digital display such as an LCD monitor. DVI enables a video signal to be transferred from a PC source to a digital display in its native digital form, simplifying the way PCs communicate with displays and improving display image quality." — Digital Visual Interface and TMDS Extensions White Paper by Silicon Image, Oct. 2004

M.0.3) **TMDS** stands for Transition Minimized Differential Signaling.

WHAT IS TMDS AND WHY IS IT IN MY HDMI?

One of the problems with transmitting digital signals over distances has always been the susceptibility of those signals to noise, interference and signal loss. Digital transmissions are very low current, low voltage signals, which make it easy for all these types of transmission gremlins to get in the midst of a byte.

Remember, when we're talking about digital signal, we're talking about "ones" and "zeros." A one is equal to 5 VDC (volts direct current) in most cases and a zero is equal to 0 VDC. Each one or zero is called a "bit" and eight of them together are called a "byte" (it's possible to have longer bytes, but let's not confuse the issue; most of them are eight bits).

One of the things that makes all digital systems work is the presence of a clock signal. This clock signal, which is what is being referred to when your computer says it operates as 1.8 GHz (1.8 billion cycles per second). This clock signal functions like the baton of the orchestra conductor, telling everyone when the time is, or when they should play their notes.

The earliest attempt at getting rid of transmission gremlins, back in the stone age of computers, before high-speed Internet access existed, was called parity checking. With parity checking, each 8-bit "byte" had a parity bit added to the end of it. The number of "ones" in the byte was counted, and the "parity bit" was made a one or a zero to make the whole byte have an even or odd number of ones (depending upon whether they were using even or odd parity).

If one of those transmission gremlins happened to get into a byte, it generally had to travel solo. The odds of two of them managing to hop the train and get into the same byte was pretty darn slim. So, the piece of hardware that was receiving the information would count the ones in the bytes received, verifying that they were okay. If one came through wrong, such as an even number of ones in a byte that was supposed to be odd parity, the receiving device would ask for that data to be sent again.

This system worked fine, as long as we were talking about a relatively low quantity of low speed transmissions. However, the faster the transmissions have become and the greater amount of data that is transmitted over those lines, the greater the need to insure that the data is accurate. When video started being transmitted digitally, both the need and the difficulty of insuring the accuracy of that data grew once again.

Transition Minimized Differential Signaling (TMDS) was developed by Silicon Image Inc. as a two-part system to reduce the possibility of transmission gremlins in serial data, specifically video data sent by serial connection. The system consists of two parts; a physical connectivity part and a software algorithm that codes and decodes the information.

TMDS is the result of lots of very interesting theories being put together in the same place to accomplish the same goal. Let me just briefly give you an idea of some of those theories:

- It uses **Differential Signaling** - That means that the signal is sent over two separate lines, out of phase with each other (the positive and negative reversed). When it gets to the other end, the signals are merged back into one, eliminating any static gremlins, which won't have its corresponding out of phase signal on the other line.
- It travels over **Twisted Pairs** - Twisted pairs, rather than coaxial cables actually provide for lower electrical interference. Any interference picked up at a point along the way (say from being too close to an electrical power line) will only get onto one of the wires, allowing it to be eliminated by the differential signaling I just mentioned.
- Instead of having the signals compared to ground, as is done in most digital equipment, it uses **Low-Voltage Differential Signaling (LVDS)**. What that means is that the two signals are compared to each other instead of to ground. By doing this, it really doesn't matter if some spurious noise or signal gets onto one of the lines, making "ground" not really be ground; it's going to be compared to the other anyway. The comparing circuitry is just going to look for differences between highs and lows.
- The data being sent over the line should be **DC balanced**. This means that there should be as many bits that are ones as there are zeroes. DC balancing reduces the "charge" (think battery) on the line, which resists further changes from ones to zeroes.
- At the same time that the data signal needs to be DC balanced, it also needs to be **Transition Minimized**. This means that the number of transitions from one to zero is reduced, making the likelihood of data loss by a transition being slow from the "ramp up" from zero to one.

To put all these theories to working together takes some very specific cable and connector design, along with pretty fancy manipulation of the data by the algorithms that control the TMDS.

Let's Look at the Connectivity Requirements

When we're talking about digital video signals, we've got to realize that we're talking about massive amounts of data being moved around. 1080p television resolution is actually 1,920 x 1080 pixels. That means that the data for 2,073,600 individual pixels has to be transmitted for every screen shot. Since television works at a rate of 30 "frames" (screen shots) per second, that makes for 62,208,000 pixels of information that has to be transmitted every second. Oh, I almost forgot to mention, each pixel is more than 1 byte of information.

Because of the massive amount of data included in these video signals, the three main color components of video signals, red, green and blue, are broken out and sent separately. Each of these signals is sent out over a shielded pair of twisted pair of wires, to lower the possibility of any interference. So, between the **clock signal** and the **three shielded pairs**, that's **12 pins out of the 19 pin** HDMI connector.

What's the Algorithm Doing?

A computer algorithm is just a series of instructions for a computer to execute. It's not even a complete computer program, but a piece of one. Any program on your computer, such as a word processing program or a browser, is made up of thousands of algorithms, each of which is responsible for performing one particular function. When you click a button on the toolbar, it tells the program which algorithms to use to accomplish the function that you want.

The TMDS algorithm manipulates the data that is in the byte, with the goal of making it the most easily transmitted and least likely to be damaged data possible. There are two stages of this process, each of which the algorithm selects automatically. When transmitting data, TMDS adds two control bits (a one or zero) to the data byte, making it a **10 bit** byte. These two control bits tell the receiving piece of equipment what manipulations were done to that particular byte, so that the algorithm can decode the byte.

What the TMDS algorithm is attempting to accomplish with all this manipulation is to create a "perfect" byte of information for transmission; or at least as close to perfect as possible. The theoretical ideal byte would look like 11111111, followed by another byte that is 00000000. This creates data that is both transition minimized and DC balanced.

The first stage of this process is the transition minimization. This is done by comparing the bits in the byte to the first bit and determining if a logical XOR or logical XNOR operation would make the byte have the least number of transitions.

- Logical XOR = (exclusive or) one or the other, but not both is a one
- Logical XNOR = (not exclusive or) putting it simply, it means both are one or zero; but is actually the opposite of XOR, the "N" referring to 'not'

The second stage of this process is the DC balancing, which means that the entire byte may or may not be inverted, to balance it with the byte before it.

Whether or not these two operations are accomplished is transmitted to the receiving piece of equipment, or more accurately to the algorithm in that equipment by the two control bits that have been added to the end of the byte.

All right, so what does all that complicated computer mumbo-jumbo mean to you? Really, all it means is that TMDS is an incredibly complicated system, which works automatically in the background, to insure that you get the crispest, clearest image, without any noise, snow, static lines or other transmission gremlins showing up on your screen.

While I won't go as far as to say that there's no possible way that those transmission gremlins can get into your signal and cause the occasional white or black pixel, where it should be blue; the chance of any of those gremlins messing your football game, movie or favorite sitcom up are drastically minimized.

This doesn't mean that you can't have any problems whatsoever. You can still spend a fortune on the latest LED backlit LCD or plasma TV, and Blue-Ray player and even buy gold-plated wires, but if you don't have a good signal, you won't have a good image. A crummy antenna is still a crummy antenna. If you're expecting a good image, make sure you have a good signal, whether from cable, satellite or broadcast.

M.0.4) Video Type Comparison

Component video is a video signal that has been split into two or more component channels. In popular use, it refers to a type of component analog video (CAV) information that is transmitted or stored as three separate signals. Component video can be contrasted with *composite video* (NTSC, PAL or SECAM) in which all the video information is combined into a single line-level signal that is used in analog television. Like composite, component-video cables do not carry audio and are often paired with audio cables.

When used without any other qualifications the term *component video* generally refers to analog YPbPr component video with sync on luma.

Analog component video

Reproducing a video signal on a display device (for example, a Cathode ray tube) (CRT) is a straightforward process complicated by the multitude of signal sources. DVD, VHS, computers and video game consoles all store, process and transmit video signals using different methods, and often each will provide more than one signal option. One way of maintaining signal clarity is by separating the components of a video signal so that they do not interfere with each other. A signal separated in this way is called "component video". S-Video, RGB and YPbPr signals comprise two or more separate signals: hence, all are component-video signals. For most consumer-level applications, analog component video is used. Digital component video is slowly becoming popular in both computer and home-theatre applications. Component video is capable of carrying signals such as 480i, 480p, 576i, 576p. Many new high definition TVs support the use of component video up to their native resolution.

The various RGB (red, green, blue) analog component video standards (e.g., RGBS, RGBHV, RG&SB) use no compression and impose no real limit on color depth or resolution, but require large bandwidth to carry the signal and contain much redundant data since each channel typically includes the same black and white image. Most modern computers offer this signal via the VGA port. Many televisions,

especially in Europe, utilize RGB via the SCART connector. All arcade games, excepting early vector and black and white games, use RGB monitors.

Analog RGB is slowly falling out of favor as computers obtain better clarity using DisplayPort or Digital Visual Interface (DVI) digital connections, while home theater moves towards High-Definition Multimedia Interface (HDMI). Analog RGB has been largely ignored, despite its quality and suitability, as it cannot easily be made to support digital rights management. RGB was never popular in North America for consumer electronics as S-Video was considered sufficient for consumer use, although RGB was used extensively in commercial, professional and high-end installations.

RGB requires an additional signal for synchronizing the video display. Several methods are used:

- composite sync, horizontal and vertical signals are mixed together on a separate wire (the S in RGBS)
- separate sync, where the horizontal and vertical are each on their own wire (the H and V in RGBHV)
- sync on green, where a composite sync signal is overlaid on the green wire (SoG or RGsB).
- sync on red or sync on blue, where a composite sync signal is overlaid on either the red or blue wire

RGB is a color space describing the image in the percentage of red (R), green (G), and blue (B), the primary colors. Various standards define the 100% levels of these colors, although slightly different.

YPbPr signals come from the red, green and blue colors of RGB. They are converted into two-color difference signals called B-Y and R-Y and brightness for TV or video.

It is the analog counterpart of YCbCr, which is used in digital. Manufacturers usually call YPbPr component video but know there are various types of component video. Some of them are in different forms of RGB, the raw format. Unlike YCbCr that uses only a single cable, YPbPr uses three cables. These separate cables are in reference to the three components of YPbPr after it's converted from RGB.

The three components of YPbPr are Y, Pb, and Pr. They are:

Y

The Y component carries the luma, also known as brightness or luminance. It also carries the sync information. In color television, Y represents intensity but that of the component colors' composite.

Pb

The Pb component transmits the difference between blue and luma.

Pr

The Pr component transmits the difference between red and luma.

As for the green signal, there is no need since this signal can come from the red, blue, and luma information.

M.0.5) TMDs Details

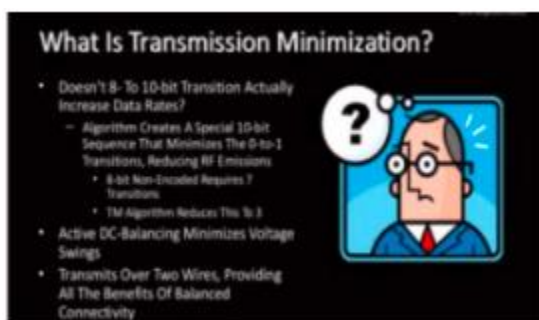
TMDs was developed by Silicon Image Inc., a member of the Digital Display Working Group, as a method for transmitting high speed digital data. It incorporates a very unique and very clever algorithm that reduces electromagnetic interference (EMI) and enables the clock recovery at prodigious distances, up to 100ft at 1920x1200. It also enables high skew tolerance on cables that are really complex, and based on their original design, should not be able to produce video images from one end to the other. It does all of this with a very high level of confidence.

TMDs is a lot like RGBHV, and much like the analog world we live in today, in that it uses four channels: Red, Green, Blue and Clock. So, if someone said to you, *"I have four coaxial cables instead of five so since I can't use RGBHV to connect my video source to my projector or my video source to my display, what can I use?"* Well, you would probably respond that they could use SRGB, where we composite the horizontal and vertical sync and then multiplex them on a single cable. That is exactly what is happening on TMDs. So now you can begin to see that we are not in a foreign land. This looks very familiar — Red, Green, Blue and Clock — and it is a two stage process. This algorithm converts the input of an 8-bit video word into a 10-bit video word. By doing that, it does something very counterintuitive; it makes the signal smaller.

M.0.5.1) Transition Minimization

That is very unusual. Why would it do that? How does it do that? How did these computer guys get so clever? TMDs signaling uses twisted pair, hence the term "differential" in TMDs. Twisted pair means we are going to have common mode noise reduction, or interference rejection. This means we are going to have a higher head room. It also operates at current modal logic, which tells us that we're talking about something that is operating under 5 volts. In fact, the way that HDMI operates between the transmitter and the receiver is to operate at a 5 volt handshake, while 3.3 volts is the current mode logic where the actual data is transferred.

There are three twisted pairs for Red, Green and Blue, plus a fourth twisted pair for sync and, once again, that unique 8-bit to 10-bit conversion capability. So, what does that mean? That is the **transition minimized** part of Transition Minimized Differential Signaling. Here is what happens. Pretend the image below is the image you see on your computer screen right now. You see a black screen image with white letters on it. We know that each section of that screen is described by a digital word. Black, also known in digital as zero (as defined by IRE, Institute of Radio Engineers), is described by an 8-bit digital word that is 00000000. As we get to the letter "W" in the word "What" in the headline, it changes to white.



The signal goes to 100 IRE, or full saturation, full white, where all three colors are at maximum output. This is indicated by a digital word of 11111111. Every time you transition from a digital zero to a digital

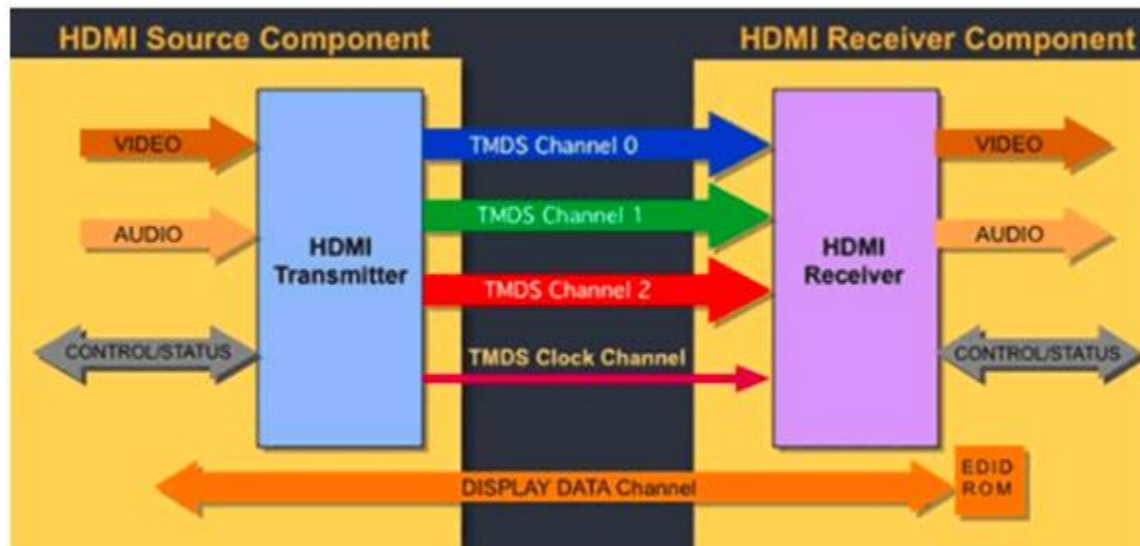
one, or transition from one bit to the next, it is described by an electrical square wave. Square wave, as you may remember from your engineering classes, is a fundamental sine wave plus all of the odd harmonics, all of the very high frequencies. This means that when we have to make eight transitions we have a tremendous amount of high frequency going through this cable. It becomes very difficult for us to be able to encompass all of this bandwidth on any kind of a practical transmission.

So what our computer brethren did was to take a look at this and come up with a better solution. Most of what happens in video happens in shades of gray, but even as we look at the other colors we realize that those colors are described by shades of gray going through a red, green or blue filter, then combining to actually make the color. All of these variations happen in the four middle bits. So the computer guys said, *"What if we take the four middle bits and where there are all ones we inverted them and made them all zeroes and then added a one to the very end? That way we eliminate a lot of these transitions, allowing us to carry less high frequency material."* They then went one step further and said, *"What if the least significant bit and the most significant bit are ones, and how about we invert those and make those zeroes and then add another bit to the end?"* By going from eight bits to ten bits, what they have actually done is minimize the transitions from zero to one so that the maximum amount of transitions being applied is five rather than eight. This eliminates a lot of high frequency material that needs to be processed and transmitted. So it is rather counterintuitive to go from eight bits to ten bits, to go from a smaller word to a larger word, and then end up with a smaller amount of information. That is the beauty of data truncation or algorithms. This algorithm uses a special ten bit sequence to minimize the zero to one transitions. Hence, TRANSITION MINIMIZED DIFFERENTIAL (because it's going over a twisted pair) Signaling. So now we understand TMDS and all of the benefits it provides, including providing up to a D4k level of transmission.

D4K resolution refers to a horizontal [display resolution](#) of approximately 4,000 [pixels](#).^[1] [Digital television](#) and [digital cinematography](#) commonly use several different 4K resolutions. In television and consumer media, 3840 × 2160 (4K [UHD](#)) is the dominant 4K standard, whereas the [movie projection](#) industry uses 4096 × 2160 ([DCI 4K](#)).

M.0.5.2) Hardware Communication

Let us now take a look at how this hardware communication is played out. The block diagram you see below could represent, for instance, the output of a Blu-ray DVD player and the input of an LCD panel in your living room. It could also represent the output of a codec, a medical imaging device or other such device with an HDMI output, and the HDMI input on something like a projector or LCD panel in a digital signage installation.



What you see here should make you feel pretty good because it is very familiar. In the middle, you see the following TMDS channels: 0 (Blue), 1 (Green) and 2 (Red). This is the same Red, Green and Blue that you have been used to seeing all along. The fourth pair is the TMDS Clock Channel; there is your sync. So you see that we are really not in foreign territory here. These things are very familiar to those of us who have been working with analog audio and video for a while. What you see in that fifth connection, *Digital Display Data Channel (DDC)*, also known as EDID - carries a tremendous amount of information. This is where things start to get really interesting.

M.0.5.3) Extended Display Identification Data (EDID) is a metadata format for display devices to describe their capabilities to a video source (e.g. graphics card or set-top box). The data format is defined by a standard published by the Video Electronics Standards Association (VESA).

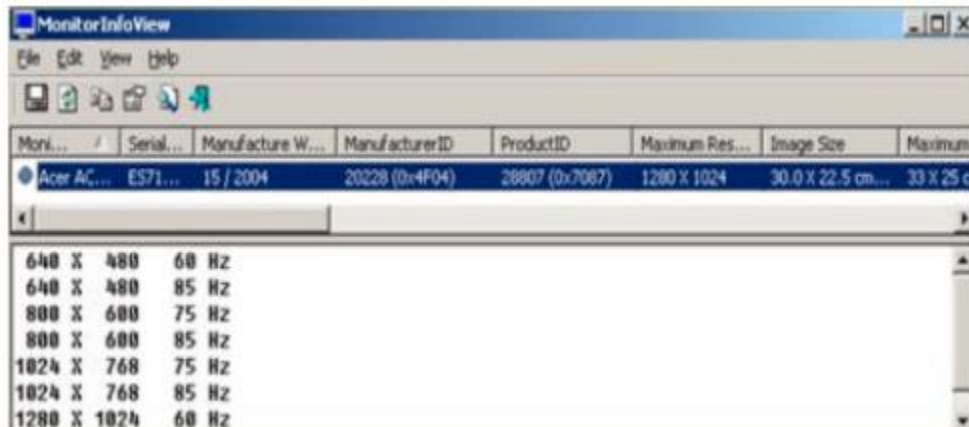
The EDID data structure includes manufacturer name and serial number, product type, phosphor or filter type (as chromaticity data), timings supported by the display, display size, luminance data and (for digital displays only) pixel mapping data.

You know that if you connect your computer up to a monitor using a VGA cable you have DDC channel information there. That is what tells your computer to switch from 1280x800 to 1024x768 to present on a particular projector. That process is highly automated.

In the digital world it is a little bit more sophisticated. In the digital environment we move from Display Data Channel (DDC), which is a digital communications protocol between a display and a source that allows these devices to understand at what resolution they can operate.

This moves us into something a little more sophisticated called the Extended Display Identification Data, or EDID. EDID, now on version 1.3, is a 256-byte structure that provides a tremendous amount of information such as: Monitor name, identification number, model number, serial number, display size, aspect ratio, etc. etc.

The screenshot you see below is really kind of interesting. It might be a little bit small, but this shows you that the DDC channel EDID information is not simply transmitting the resolution the device can operate on. Rather, it is actually transmitting the serial number, the build date, the firmware date, the manufacturing number, the manufacturing identification number, the maximum resolution, the color depth, and a table of all the resolutions that the device can resolve. So there is a lot of information being transmitted there which makes these monitors compatible with a number of digital devices. Here in lies the very first problem that we experienced.



Back when HDMI first was released in the market, we already had DVI. However, what we really had was a failure to properly write EDID information on all of these devices. In the early days manufacturers actually had something called "plug fests". These were events where display manufacturers brought their displays and source manufacturers brought their sources. They would get together and start plugging these devices together and would take notes until they found where the incompatibilities existed. What we discovered was that a lot of the incompatibilities had to do with the EDID information being improperly coded resulting in the devices being unable to talk to one another. I'm very happy to say that most of these issues now reside only in legacy equipment, so the only time you are really going to experience any type of EDID information issues is if you are trying to incorporate devices that are four to six years old into a contemporary installation. Most EDID issues have been resolved and, in fact almost every digital device, whether it's an LCD panel, plasma panel, DLP panel or Blu-ray player, have firmware that constantly updates the EDID information to make sure these devices are compatible with all contemporary technologies.

Now, I would like to point out one other thing about the block diagram on the previous page. Although you will notice there is Red, Green, Blue and Clock as well as DDC data, what we do not see there is a pair for audio. In the digital world audio is embedded into the Red, Green and Blue digital video signal. So, you cannot possibly have a cable that has HDMI on one end and DVI on the other with 3.5mm audio. This is electronically combined and electronically separated at both ends, so it is part of the video signal. Moreover, audio truly is encompassed within the TMDS environment. A lot of people do not realize that even DVI-D is capable of supporting TCM digital audio in this video information. It was just never implemented at the time, and it took HDMI to get us there.

Prior to the DDC, the [VGA](#) standard had reserved four pins in the analog [VGA connector](#), known as ID0, ID1, ID2 and ID3 (pins 11, 12, 4 and 15) for identification of monitor type. These ID pins, attached to resistors to pull one or more of them to ground (GND), allowed for the definition of the monitor type, with all open (n/c, not connected) meaning "no monitor".

In the most commonly documented scheme, the ID3 pin was unused and only the 3 remaining pins were defined. The ID0 was pulled to GND by color monitors, while the monochrome monitors pulled ID1 to GND. Finally, the ID2 pulled to GND signaled a monitor capable of 1024×768 resolution, such as [IBM 8514](#). In this scheme, the input states of the ID pins would encode the monitor type as follows:^{[1][2][3]}

ID2 (pin 4)	ID0 (pin 11)	ID1 (pin 12)	monitor type
n/c	n/c	n/c	no monitor connected
n/c	n/c	GND	< 1024×768, monochrome
n/c	GND	n/c	< 1024×768, color
GND	GND	n/c	≥ 1024×768, color

DDC changed the purpose of the ID pins to incorporate a [serial link interface](#). However, during the transition, the change was not backwards-compatible and video cards using the old scheme could have problems if a DDC-capable monitor was connected.^[5] The DDC signal can be sent to or from a video graphics array (VGA) monitor with the I²C protocol using the master's serial clock and serial data pins.

Consumer Electronics Control (CEC) is an HDMI feature designed to allow the user to command and control up to ten CEC-enabled devices connected through HDMI by using just one of their remote controls (for example by controlling a television set, set-top box, PVR/DVR, and DVD player using only the remote control of the TV). CEC also allows for individual CEC-enabled devices to command and control each other without user intervention.

HDMI-CEC is a one-wire bidirectional serial bus that uses the industry-standard AV.link protocol to perform remote control functions. CEC wiring is mandatory, although implementation of CEC in a product is optional. It was defined in HDMI Specification 1.0 and updated in HDMI 1.2, HDMI 1.2a and HDMI 1.3a (the last added timer and audio commands to the bus). USB to CEC Adapters exist that allow a computer to control CEC-enabled devices.

HDCP stands for **High-Bandwidth Digital Content Protection**. It's a coding scheme developed by Intel used to protect audio and video signals traveling through DVI, HDMI, and DisplayPort from being copied and illegally intercepted during a streaming session. It shields the transfer of digital content from a video source like a computer or DVD player to a receiver like a monitor or TV screen. This technology was officially approved by the Federal Communications Commission in 2004.

M.1) Digital Video Output (DVI/HDMI)

The streamer can serialize its internal 32 pin output data P[31:0] into 8-pin/10-bit digital video format, where the 32-pin output becomes \$000000xx with \$xx being a reversible pattern of RED, GRN, BLU, and CLK differential pairs.

The SETCMOD instruction is used to write bits 8:7 of the CMOD register to set digital video mode:

CMOD[8:7]	Mode	Pin +31:8	Pin +7	Pin +6	Pin +5	Pin +4	Pin +3	Pin +2	Pin +1	Pin +0
%0x	Normal	P[31:8]	P[7]	P[6]	P[5]	P[4]	P[3]	P[2]	P[1]	P[0]
%10	DVI fwd	\$000000	RED+	RED-	GRN+	GRN-	BLU+	BLU-	CLK+	CLK-
%11	DVI rev	\$000000	CLK-	CLK+	BLU-	BLU+	GRN-	GRN+	RED-	RED+

Eight-bit red, green, and blue pixel data are encoded into 10-bit TMDS patterns for transmission, while control data, such as horizontal and vertical syncs, are transmitted literally. P[1] in the internal pin output data selects whether data will be TMDS-encoded or sent out literally:

P[31:0]	RED+/- serial	GRN+/- serial	BLU+/- serial
%RRRRRRRRR_GGGGGGGG_BBBBBBBB_XXXXXX0x	%RRRRRRRRR gets encoded	%GGGGGGGGG gets encoded	%BBBBBBBBB gets encoded
%rrrrrrrrrr_gggggggggg_bbbbbbbbbb_1x	%rrrrrrrrrr is sent literally	%gggggggggg is sent literally	%bbbbbbbbb is sent literally

Digital video output mode requires that the P2 clock frequency be 10x the pixel rate. For 640x480 digital video, which has a pixel rate of 25MHz, the P2 chip must be clocked at 250MHz.

The NCO frequency must be set to 1/10 of the main clock using the value \$0CCCCCCC+1, where the +1 forces initial NCO rollover on the 10th clock.

The following program displays a 16bpp image in 640x480 HDMI mode:

```

'*****
'*  VGA 640 x 480 x 16bpp 5:6:5 RGB - HDMI  *
'*****

CON          hdmi_base = 16          'must be a multiple of 8

DAT          org

'
'
'  Setup
'
          hubset  ##%1_000001_0000011000_1111_10_00      'config PLL, 20MHz/2*25*1 = 250MHz
          waitx   ##20_000_000 / 200                    'allow crystal+PLL 5ms to stabilize
          hubset  ##%1_000001_0000011000_1111_10_11      'switch to PLL

          rdfast  ##640*350*2/64,##$1000  'set rdfast to wrap on bitmap

          setxfrq ##$0CCCCCCC+1          'set streamer freq to 1/10th clk

          setcm0d ##$100                  'enable HDMI mode

          drvl    #7<<6 + hdmi_base      'enable HDMI pins

          wrpin   ##%100100_00_00000_0,#7<<6 + hdmi_base 'set 1mA drive on HDMI pins
'
'
'  Field loop
'
field      mov     hsync0, sync_000        'vsync off
          mov     hsync1, sync_001

          callpa  #90, #blank              'top blanks

          mov     x, #350                  'set visible lines
line       call   #hsync                  'do horizontal sync

```

```

        xcont    m_rf,#0                'do visible line
        djnz     x,#line                'another line?

        callpa   #83,#blank             'bottom blanks

        mov      hsync0, sync_222       'vsync on
        mov      hsync1, sync_223

        callpa   #2,#blank              'vertical sync blanks

        jmp      #field                 'loop
'
'
' Subroutines
'
blank      call   #hsync                'blank lines
          xcont   m_vi,hsync0
          _ret_   djnz   pa,#blank

hsync      xcont   m_bs,hsync0          'horizontal sync
          xzero   m_sn,hsync1
          _ret_   xcont   m_bv,hsync0
'
'
' Initialized data
'
sync_000   long    %1101010100_1101010100_1101010100_10    '
sync_001   long    %1101010100_1101010100_0010101011_10    '          hsync
sync_222   long    %0101010100_0101010100_0101010100_10    'vsync
sync_223   long    %0101010100_0101010100_1010101011_10    'vsync + hsync

m_bs       long    $70810000 + hdmi_base<<17 + 16          'before sync
m_sn       long    $70810000 + hdmi_base<<17 + 96          'sync
m_bv       long    $70810000 + hdmi_base<<17 + 48          'before visible
m_vi       long    $70810000 + hdmi_base<<17 + 640          'visible

m_rf       long    $B0850000 + hdmi_base<<17 + 640          'visible rword rgb16 (5:6:5)

```

```

'
'
' Uninitialized data
'
x          res      1

hsync0     res      1
hsync1     res      1
'
'
' Bitmap
'
          orgh      $1000 - 70          'justify pixels at $1000
          file      "birds_16bpp.bmp"    'rayman's picture (640 x 350)

```

M.2) ColorSpace Converter

Each cog has a colorspace 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 colorspace converter is configured via the following instructions:

```

SETCY {#}D - Set colorspace converter CY parameter to D[31:0]
SETCI {#}D - Set colorspace converter CI parameter to D[31:0]
SETCQ {#}D - Set colorspace converter CQ parameter to D[31:0]
SETCFRQ {#}D - Set colorspace converter CFRQ parameter to D[31:0]
SETCMOD {#}D - Set colorspace 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 colorspace 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:

$$PHS[31:0] = PHS[31:0] - CFRQ[31:0]$$

$$IQ[7:0] = Q \text{ of } (I, Q) \text{ after being rotated by PHS and multiplied by } 1.646$$

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:

$$\begin{aligned}
 FY[7:0] &= CY[7:0] + (DAC0 \& \{8\{CMOD[3]\}\}) + Y[7:0] && \text{(VGA R / HDTV Y)} \\
 FI[7:0] &= CI[7:0] + (DAC0 \& \{8\{CMOD[2]\}\}) + I[7:0] && \text{(VGA G / HDTV Pb)} \\
 FQ[7:0] &= CQ[7:0] + (DAC0 \& \{8\{CMOD[1]\}\}) + Q[7:0] && \text{(VGA B / HDTV Pr)} \\
 FS[7:0] &= \{8\{DAC0[0] \wedge CMOD[0]\}\} && \text{(VGA H-Sync)} \\
 FIQ[7:0] &= CQ[7:0] + IQ[7:0] && \text{(Chroma)} \\
 FYS[7:0] &= DAC0[1] \quad ? \quad 8'b0 && \text{(1x = Luma Sync)} \\
 &: DAC0[0] \quad ? \quad CI[7:0] && \text{(01 = Luma Blank/Burst)} \\
 &: \quad CY[7:0] + Y[7:0] && \text{(00 = Luma Visible)} \\
 FYC[7:0] &= FYS[7:0] + IQ[7:0] && \text{(Composite Luma+Chroma)}
 \end{aligned}$$

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)

M.3) 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

```

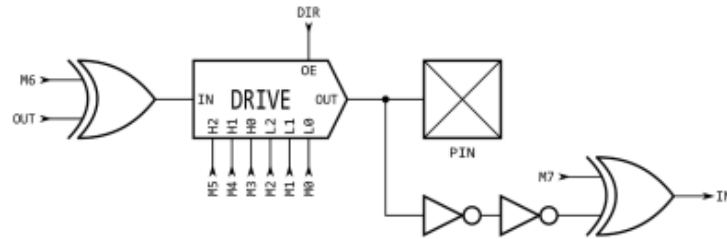
Here are the DMIX and SMIX terms, according to each instruction:

	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]

Appendix “N” PIN Logic Diagrams

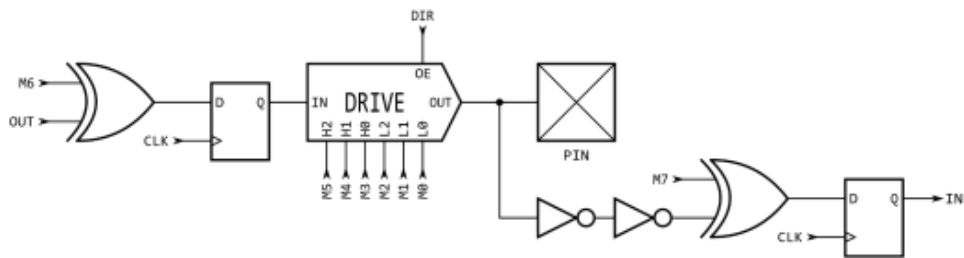
Equivalent Schematics for Each Unique I/O Pin Configuration

%00000MMMMMMMM - Logic

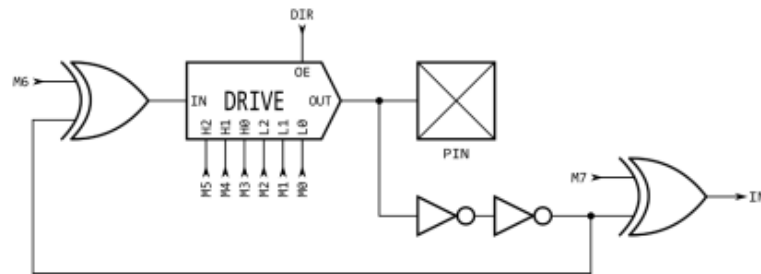


H/L	DRIVE
000	Digital
001	1.5k
010	15k
011	150k
100	1mA
101	100uA
110	10uA
111	Float

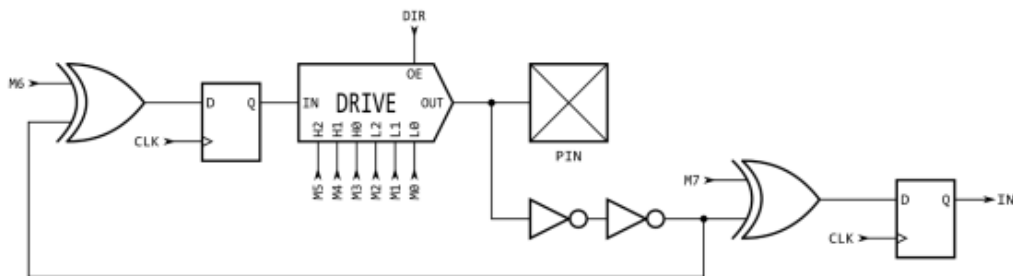
%00001MMMMMMMM - Logic, Clocked



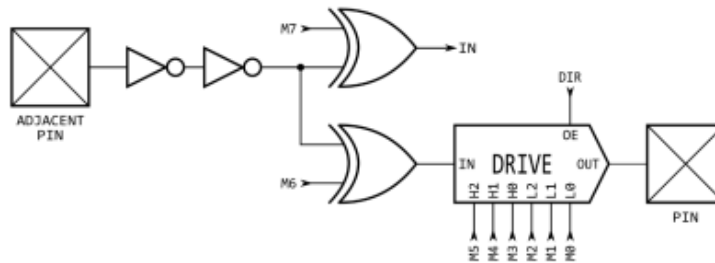
%00010MMMMMMMM - Logic with Feedback



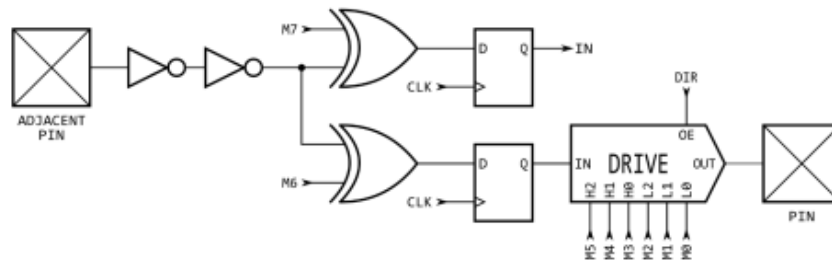
%00011MMMMMMMM - Logic with Feedback, Clocked



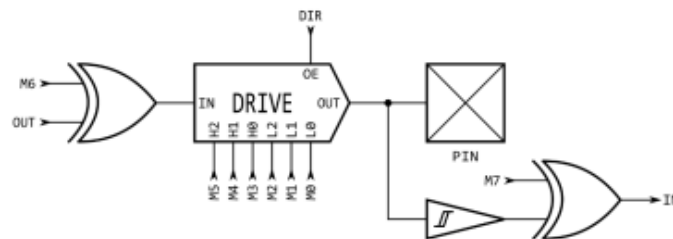
%00100MMMMMMMM - Logic with Adjacent-Pin Feedback



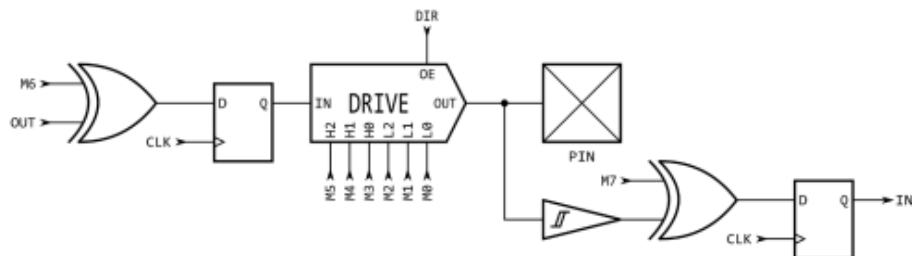
%00101MMMMMMMM - Logic with Adjacent-Pin Feedback, Clocked



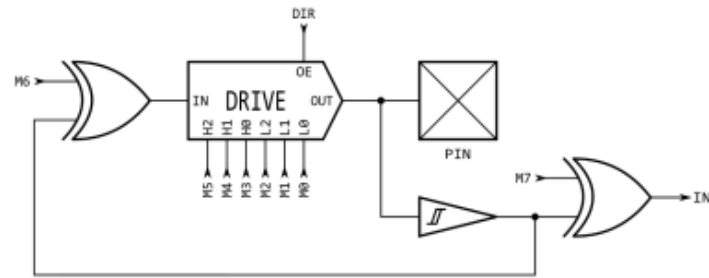
%00110MMMMMMMM - Schmitt



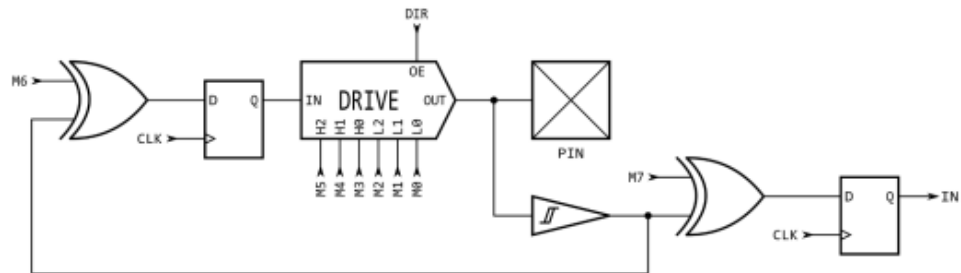
%00111MMMMMMMM - Schmitt, Clocked



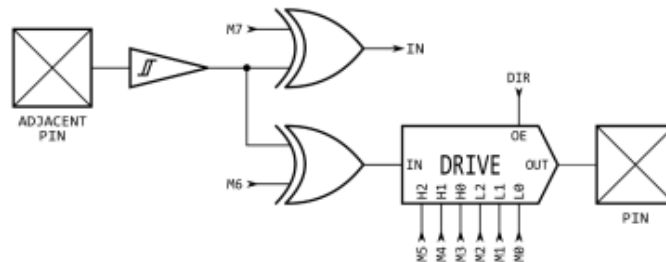
%01000MMMMMMMM - Schmitt with Feedback



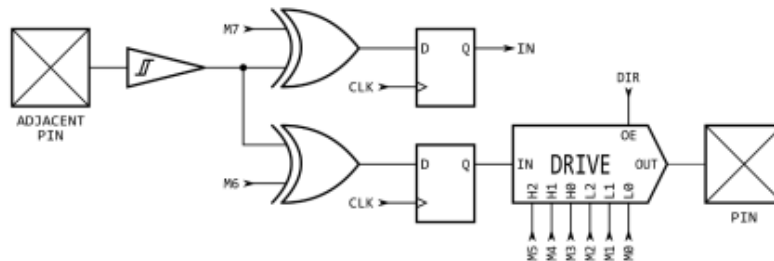
%01001MMMMMMMM - Schmitt with Feedback, Clocked



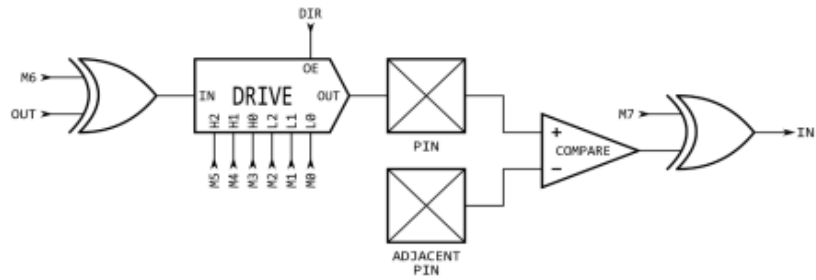
%01010MMMMMMMM - Schmitt with Adjacent-Pin Feedback



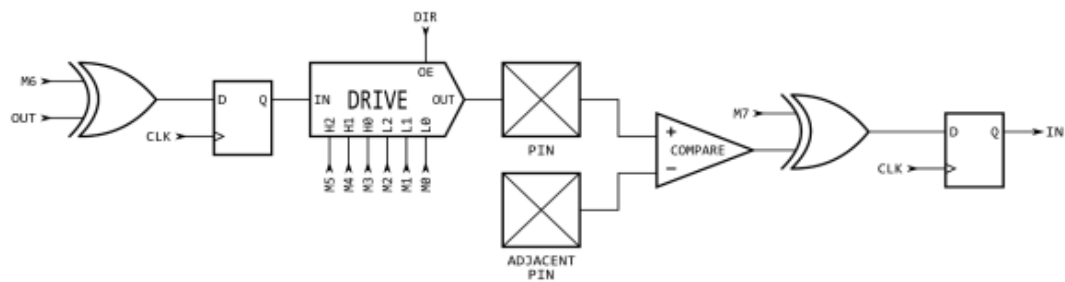
%01011MMMMMMMM - Schmitt with Adjacent-Pin Feedback, Clocked



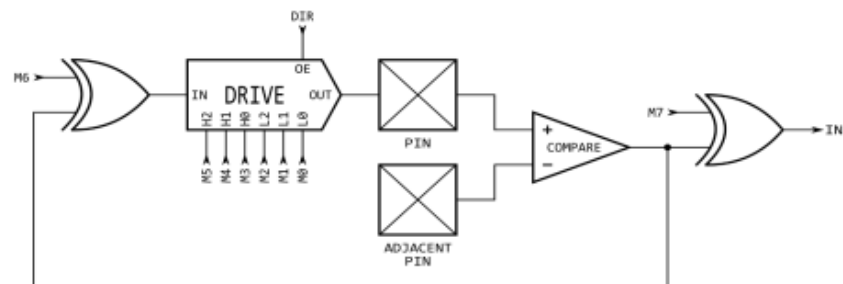
%01100MMMMMMMM - Comparator



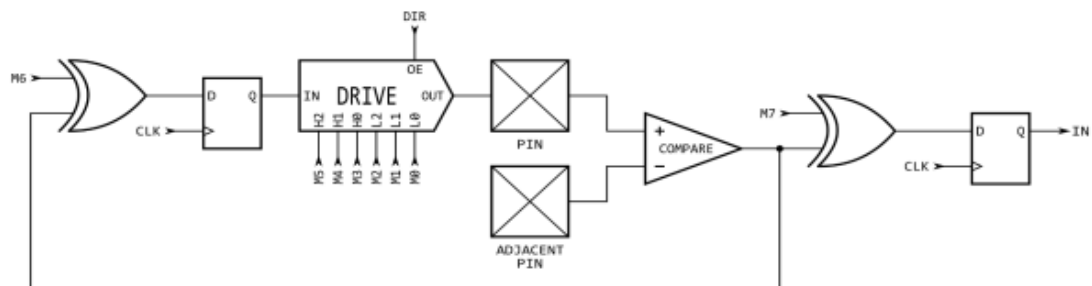
%01101MMMMMMMM - Comparator, Clocked



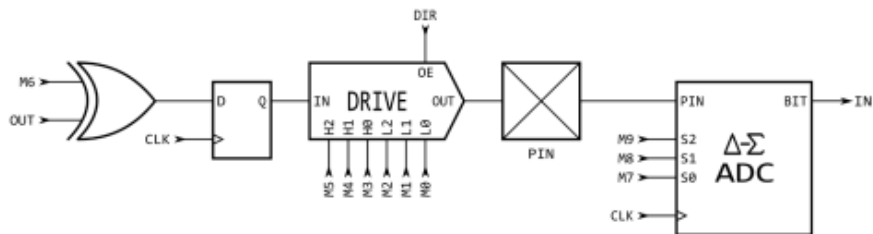
%01110MMMMMMMM - Comparator with Feedback



%01111MMMMMMMM - Comparator with Feedback, Clocked

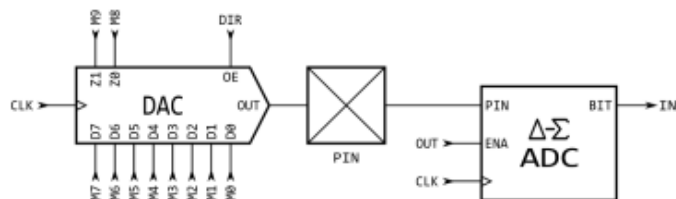


%100MMMMMMMMMM - ADC with Optional Drive



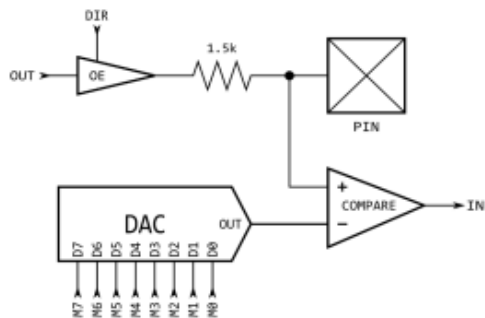
SSS	ADC
000	GND
001	VIO
010	Float
011	1x
100	3.2x
101	10x
110	32x
111	100x

%101MMMMMMMMMM - DAC with Optional ADC

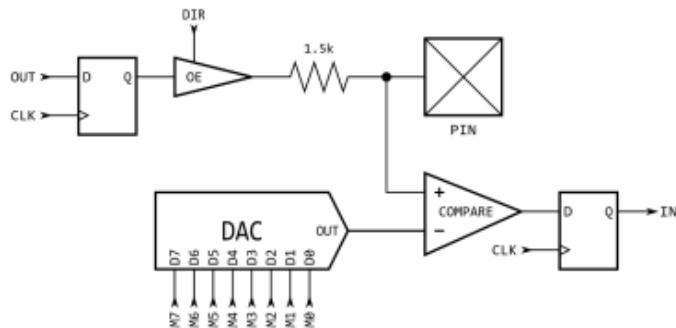


ZZ	DAC
00	990 ohm 3.3V
01	600 ohm 2.0V
10	124 ohm 3.3V
11	75 ohm 2.0V

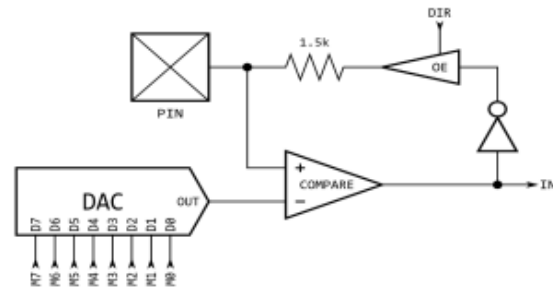
%11000MMMMMMMM - Level Comparator with 1.5k Output



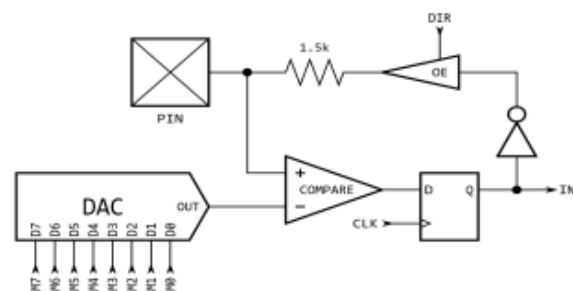
%11001MMMMMMMM - Level Comparator with 1.5k Output, Clocked



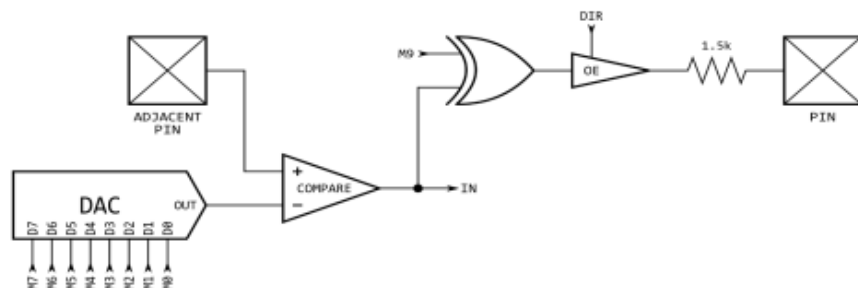
%11010MMMMMMMM - Level Comparator with Local Feedback



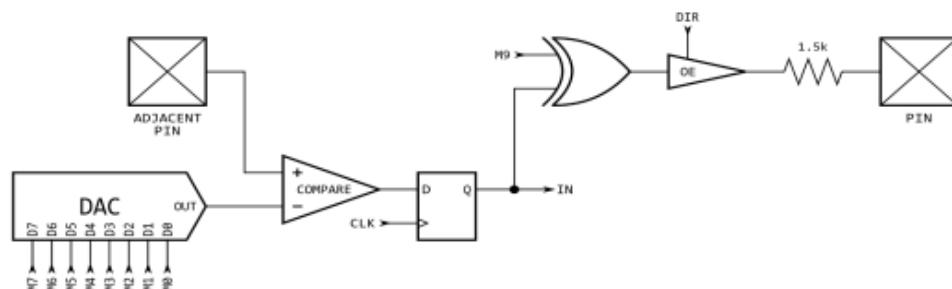
%11011MMMMMMMM - Level Comparator with Local Feedback, Clocked



%111M0MMMMMMMM - Level Comparator with Separate Feedback



%111M1MMMMMMMM - Level Comparator with Separate Feedback, Clocked



Appendix “O” PASM Instructions

Order	#S = immediate (I=1). S = register. #D = immediate (L=1). D = register. - Assembly Syntax -	* Z =(result)=0
1	NOP	No operation.
2	ROR D,{#}S {WC/WZ/WCZ}	Rotate right. D = [31:0] of ({D[31:0], D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
3	ROL D,{#}S {WC/WZ/WCZ}	Rotate left. D = [31:0] of ({D[31:0], D[31:0]} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *
4	SHR D,{#}S {WC/WZ/WCZ}	Shift right. D = [31:0] of ({32'b0, D[31:0]} >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
5	SHL D,{#}S {WC/WZ/WCZ}	Shift left. D = [31:0] of ({D[31:0], 32'b0} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *
6	RCR D,{#}S {WC/WZ/WCZ}	Rotate carry right. D = [31:0] of ({32{C}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
7	RCL D,{#}S {WC/WZ/WCZ}	Rotate carry left. D = [31:0] of ({D[31:0], {32{C}}} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *
8	SAR D,{#}S {WC/WZ/WCZ}	Shift arithmetic right. D = [31:0] of ({32{D[31]}}, D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *
9	SAL D,{#}S {WC/WZ/WCZ}	Shift arithmetic left. D = [31:0] of ({D[31:0], {32{D[0]}}} << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *
10	ADD D,{#}S {WC/WZ/WCZ}	Add S into D. D = D + S. C = carry of (D + S). *
11	ADDX D,{#}S {WC/WZ/WCZ}	Add (S + C) into D, extended. D = D + S + C. C = carry of (D + S + C). Z = Z AND (result == 0).
12	ADDS D,{#}S {WC/WZ/WCZ}	Add S into D, signed. D = D + S. C = correct sign of (D + S). *
13	ADDSD D,{#}S {WC/WZ/WCZ}	Add (S + C) into D, signed and extended. D = D + S + C. C = correct sign of (D + S + C). Z = Z AND (result == 0).

14	SUB D,{#}S {WC/WZ/WCZ}	Subtract S from D. $D = D - S$. C = borrow of $(D - S)$. *
15	SUBX D,{#}S {WC/WZ/WCZ}	Subtract $(S + C)$ from D, extended. $D = D - (S + C)$. C = borrow of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.
16	SUBS D,{#}S {WC/WZ/WCZ}	Subtract S from D, signed. $D = D - S$. C = correct sign of $(D - S)$. *
17	SUBSX D,{#}S {WC/WZ/WCZ}	Subtract $(S + C)$ from D, signed and extended. $D = D - (S + C)$. C = correct sign of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.
18	CMP D,{#}S {WC/WZ/WCZ}	Compare D to S. C = borrow of $(D - S)$. $Z = (D == S)$.
19	CMPX D,{#}S {WC/WZ/WCZ}	Compare D to $(S + C)$, extended. C = borrow of $(D - (S + C))$. $Z = Z \text{ AND } (D == S + C)$.
20	CMPS D,{#}S {WC/WZ/WCZ}	Compare D to S, signed. C = correct sign of $(D - S)$. $Z = (D == S)$.
21	CMPSX D,{#}S {WC/WZ/WCZ}	Compare D to $(S + C)$, signed and extended. C = correct sign of $(D - (S + C))$. $Z = Z \text{ AND } (D == S + C)$.
22	CMPR D,{#}S {WC/WZ/WCZ}	Compare S to D (reverse). C = borrow of $(S - D)$. $Z = (D == S)$.
23	CMPM D,{#}S {WC/WZ/WCZ}	Compare D to S, get MSB of difference into C. C = MSB of $(D - S)$. $Z = (D == S)$.
24	SUBR D,{#}S {WC/WZ/WCZ}	Subtract D from S (reverse). $D = S - D$. C = borrow of $(S - D)$. *
25	CMPSUB D,{#}S {WC/WZ/WCZ}	Compare and subtract S from D if $D \geq S$. If $D \geq S$ then $D = D - S$ and $C = 1$, else D same and $C = 0$. *
26	FGE D,{#}S {WC/WZ/WCZ}	Force $D \geq S$. If $D < S$ then $D = S$ and $C = 1$, else D same and $C = 0$. *
27	FLE D,{#}S {WC/WZ/WCZ}	Force $D \leq S$. If $D > S$ then $D = S$ and $C = 1$, else D same and $C = 0$. *
28	FGES D,{#}S {WC/WZ/WCZ}	Force $D \geq S$, signed. If $D < S$ then $D = S$ and $C = 1$, else D same and $C = 0$. *
29	FLES D,{#}S {WC/WZ/WCZ}	Force $D \leq S$, signed. If $D > S$ then $D = S$ and $C = 1$, else D same and $C = 0$. *
30	SUMC D,{#}S {WC/WZ/WCZ}	Sum $\pm S$ into D by C. If $C = 1$ then $D = D - S$, else $D = D + S$. C = correct sign of $(D \pm S)$. *
31	SUMNC D,{#}S {WC/WZ/WCZ}	Sum $\pm S$ into D by !C. If $C = 0$ then $D = D - S$, else $D = D + S$. C = correct sign of $(D \pm S)$. *
32	SUMZ D,{#}S {WC/WZ/WCZ}	Sum $\pm S$ into D by Z. If $Z = 1$ then $D = D - S$, else $D = D + S$. C = correct sign of $(D \pm S)$. *

33	SUMNZ D,{#}S {WC/WZ/WCZ}	Sum +/-S into D by !Z. If Z = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *
34	TESTB D,{#}S WC/WZ	Test bit S[4:0] of D, write to C/Z. C/Z = D[S[4:0]].
35	TESTBN D,{#}S WC/WZ	Test bit S[4:0] of !D, write to C/Z. C/Z = !D[S[4:0]].
36	TESTB D,{#}S ANDC/ANDZ	Test bit S[4:0] of D, AND into C/Z. C/Z = C/Z AND D[S[4:0]].
37	TESTBN D,{#}S ANDC/ANDZ	Test bit S[4:0] of !D, AND into C/Z. C/Z = C/Z AND !D[S[4:0]].
38	TESTB D,{#}S ORC/ORZ	Test bit S[4:0] of D, OR into C/Z. C/Z = C/Z OR D[S[4:0]].
39	TESTBN D,{#}S ORC/ORZ	Test bit S[4:0] of !D, OR into C/Z. C/Z = C/Z OR !D[S[4:0]].
40	TESTB D,{#}S XORC/XORZ	Test bit S[4:0] of D, XOR into C/Z. C/Z = C/Z XOR D[S[4:0]].
41	TESTBN D,{#}S XORC/XORZ	Test bit S[4:0] of !D, XOR into C/Z. C/Z = C/Z XOR !D[S[4:0]].
42	BITL D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = 0. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
43	BITH D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = 1. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
44	BITC D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = C. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
45	BITNC D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = !C. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
46	BITZ D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = Z. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
47	BITNZ D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = !Z. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
48	BITRND D,{#}S {WCZ}	Bits D[S[9:5]+S[4:0]:S[4:0]] = RNDs. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
49	BITNOT D,{#}S {WCZ}	Toggle bits D[S[9:5]+S[4:0]:S[4:0]]. Other bits unaffected. Prior SETQ overrides S[9:5]. C,Z = original D[S[4:0]].
50	AND D,{#}S {WC/WZ/WCZ}	AND S into D. D = D & S. C = parity of result. *
51	ANDN D,{#}S {WC/WZ/WCZ}	AND !S into D. D = D & !S. C = parity of result. *

52	OR D,{#}S {WC/WZ/WCZ}	OR S into D. $D = D \mid S$. C = parity of result. *
53	XOR D,{#}S {WC/WZ/WCZ}	XOR S into D. $D = D \wedge S$. C = parity of result. *
54	MUXC D,{#}S {WC/WZ/WCZ}	Mux C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{C\}\})$. C = parity of result. *
55	MUXNC D,{#}S {WC/WZ/WCZ}	Mux !C into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!C\}\})$. C = parity of result. *
56	MUXZ D,{#}S {WC/WZ/WCZ}	Mux Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{Z\}\})$. C = parity of result. *
57	MUXNZ D,{#}S {WC/WZ/WCZ}	Mux !Z into each D bit that is '1' in S. $D = (!S \& D) \mid (S \& \{32\{!Z\}\})$. C = parity of result. *
58	MOV D,{#}S {WC/WZ/WCZ}	Move S into D. $D = S$. C = S[31]. *
59	NOT D,{#}S {WC/WZ/WCZ}	Get !S into D. $D = !S$. C = !S[31]. *
60	NOT D {WC/WZ/WCZ}	Get !D into D. $D = !D$. C = !D[31]. *
61	ABS D,{#}S {WC/WZ/WCZ}	Get absolute value of S into D. $D = \text{ABS}(S)$. C = S[31]. *
62	ABS D {WC/WZ/WCZ}	Get absolute value of D into D. $D = \text{ABS}(D)$. C = D[31]. *
63	NEG D,{#}S {WC/WZ/WCZ}	Negate S into D. $D = -S$. C = MSB of result. *
64	NEG D {WC/WZ/WCZ}	Negate D. $D = -D$. C = MSB of result. *
65	NEGC D,{#}S {WC/WZ/WCZ}	Negate S by C into D. If C = 1 then D = -S, else D = S. C = MSB of result. *
66	NEGC D {WC/WZ/WCZ}	Negate D by C. If C = 1 then D = -D, else D = D. C = MSB of result. *
67	NEGNC D,{#}S {WC/WZ/WCZ}	Negate S by !C into D. If C = 0 then D = -S, else D = S. C = MSB of result. *
68	NEGNC D {WC/WZ/WCZ}	Negate D by !C. If C = 0 then D = -D, else D = D. C = MSB of result. *
69	NEGZ D,{#}S {WC/WZ/WCZ}	Negate S by Z into D. If Z = 1 then D = -S, else D = S. C = MSB of result. *
70	NEGZ D {WC/WZ/WCZ}	Negate D by Z. If Z = 1 then D = -D, else D = D. C = MSB of result. *
71	NEGNZ D,{#}S {WC/WZ/WCZ}	Negate S by !Z into D. If Z = 0 then D = -S, else D = S. C = MSB of result. *
72	NEGNZ D {WC/WZ/WCZ}	Negate D by !Z. If Z = 0 then D = -D, else D = D. C = MSB of result. *
73	INCMOD D,{#}S	Increment with modules, If D = S Then D = 0 and C = 1

	{WC/WZ/WCZ}	else D = D + 1 and C = 0 *
74	DECMOD D,{#}S {WC/WZ/WCZ}	Decrement with modulus. If D = 0 then D = S and C = 1, else D = D - 1 and C = 0. *
75	ZEROX D,{#}S {WC/WZ/WCZ}	Zero-extend D above bit S[4:0]. C = MSB of result. *
76	SIGNX D,{#}S {WC/WZ/WCZ}	Sign-extend D from bit S[4:0]. C = MSB of result. *
77	ENCOD D,{#}S {WC/WZ/WCZ}	Get bit position of top-most '1' in S into D. D = position of top '1' in S (0..31). C = (S != 0). *
78	ENCOD D {WC/WZ/WCZ}	Get bit position of top-most '1' in D into D. D = position of top '1' in S (0..31). C = (S != 0). *
79	ONES D,{#}S {WC/WZ/WCZ}	Get number of '1's in S into D. D = number of '1's in S (0..32). C = LSB of result. *
80	ONES D {WC/WZ/WCZ}	Get number of '1's in D into D. D = number of '1's in S (0..32). C = LSB of result. *
81	TEST D,{#}S {WC/WZ/WCZ}	Test D with S. C = parity of (D & S). Z = ((D & S) == 0).
82	TEST D {WC/WZ/WCZ}	Test D. C = parity of D. Z = (D == 0).
83	TESTN D,{#}S {WC/WZ/WCZ}	Test D with !S. C = parity of (D & !S). Z = ((D & !S) == 0).
84	SETNIB D,{#}S,#N	Set S[3:0] into nibble N in D, keeping rest of D same.
85	SETNIB {#}S	Set S[3:0] into nibble established by prior ALTSN instruction.
86	GETNIB D,{#}S,#N	Get nibble N of S into D. D = {28'b0, S.NIBBLE[N]}.
87	GETNIB D	Get nibble established by prior ALTGN instruction into D.
88	ROLNIB D,{#}S,#N	Rotate-left nibble N of S into D. D = {D[27:0], S.NIBBLE[N]}.
89	ROLNIB D	Rotate-left nibble established by prior ALTGN instruction into D.
90	SETBYTE D,{#}S,#N	Set S[7:0] into byte N in D, keeping rest of D same.
91	SETBYTE {#}S	Set S[7:0] into byte established by prior ALTSB instruction.
92	GETBYTE D,{#}S,#N	Get byte N of S into D. D = {24'b0, S.BYTE[N]}.
93	GETBYTE D	Get byte established by prior ALTGB instruction into D.
94	ROLBYTE D,{#}S,#N	Rotate-left byte N of S into D. D = {D[23:0], S.BYTE[N]}.

95	ROLBYTE D	Rotate-left byte established by prior ALTGB instruction into D.
96	SETWORD D,{#}S,#N	Set S[15:0] into word N in D, keeping rest of D same.
97	SETWORD {#}S	Set S[15:0] into word established by prior ALTSW instruction.
98	GETWORD D,{#}S,#N	Get word N of S into D. $D = \{16'b0, S.WORD[N]\}$.
99	GETWORD D	Get word established by prior ALTGW instruction into D.
100	ROLWORD D,{#}S,#N	Rotate-left word N of S into D. $D = \{D[15:0], S.WORD[N]\}$.
101	ROLWORD D	Rotate-left word established by prior ALTGW instruction into D.
102	ALTSN D,{#}S	Alter subsequent SETNIB instruction. Next D field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. $D +=$ sign-extended $S[17:9]$.
103	ALTSN D	Alter subsequent SETNIB instruction. Next D field = $D[11:3]$, N field = $D[2:0]$.
104	ALTGN D,{#}S	Alter subsequent GETNIB/ROLNIB instruction. Next S field = $(D[11:3] + S) \& \$1FF$, N field = $D[2:0]$. $D +=$ sign-extended $S[17:9]$.
105	ALTGN D	Alter subsequent GETNIB/ROLNIB instruction. Next S field = $D[11:3]$, N field = $D[2:0]$.
106	ALTSB D,{#}S	Alter subsequent SETBYTE instruction. Next D field = $(D[10:2] + S) \& \$1FF$, N field = $D[1:0]$. $D +=$ sign-extended $S[17:9]$.
107	ALTSB D	Alter subsequent SETBYTE instruction. Next D field = $D[10:2]$, N field = $D[1:0]$.
108	ALTGB D,{#}S	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = $(D[10:2] + S) \& \$1FF$, N field = $D[1:0]$. $D +=$ sign-extended $S[17:9]$.
109	ALTGB D	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = $D[10:2]$, N field = $D[1:0]$.
110	ALTSW D,{#}S	Alter subsequent SETWORD instruction. Next D field = $(D[9:1] + S) \& \$1FF$, N field = $D[0]$. $D +=$ sign-extended $S[17:9]$.
111	ALTSW D	Alter subsequent SETWORD instruction. Next D field = $D[9:1]$, N field = $D[0]$.
112	ALTGW D,{#}S	Alter subsequent GETWORD/ROLWORD instruction. Next S field = $((D[9:1] + S) \& \$1FF)$, N field = $D[0]$. $D +=$ sign-extended $S[17:9]$.
113	ALTGW D	Alter subsequent GETWORD/ROLWORD instruction. Next S field = $D[9:1]$, N field = $D[0]$.

114	ALTR D,{#}S	Alter result register address (normally D field) of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
115	ALTR D	Alter result register address (normally D field) of next instruction to D[8:0].
116	ALTD D,{#}S	Alter D field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
117	ALTD D	Alter D field of next instruction to D[8:0].
118	ALTS D,{#}S	Alter S field of next instruction to (D + S) & \$1FF. D += sign-extended S[17:9].
119	ALTS D	Alter S field of next instruction to D[8:0].
120	ALTB D,{#}S	Alter D field of next instruction to (D[13:5] + S) & \$1FF. D += sign-extended S[17:9].
121	ALTB D	Alter D field of next instruction to D[13:5].
122	ALTI D,{#}S	Substitute next instruction's I/R/D/S fields with fields from D, per S. Modify D per S.
123	ALTI D	Execute D in place of next instruction. D stays same.
124	SETR D,{#}S	Set R field of D to S[8:0]. D = {D[31:28], S[8:0], D[18:0]}.
125	SETD D,{#}S	Set D field of D to S[8:0]. D = {D[31:18], S[8:0], D[8:0]}.
126	SETS D,{#}S	Set S field of D to S[8:0]. D = {D[31:9], S[8:0]}.
127	DECOD D,{#}S	Decode S[4:0] into D. D = 1 << S[4:0].
128	DECOD D	Decode D[4:0] into D. D = 1 << D[4:0].
129	BMASK D,{#}S	Get LSB-justified bit mask of size (S[4:0] + 1) into D. D = (\$0000_0002 << S[4:0]) - 1.
130	BMASK D	Get LSB-justified bit mask of size (D[4:0] + 1) into D. D = (\$0000_0002 << D[4:0]) - 1.
131	CRCBIT D,{#}S	Iterate CRC value in D using C and polynomial in S. If (C XOR D[0]) then D = (D >> 1) XOR S, else D = (D >> 1).
132	CRCNIB D,{#}S	Iterate CRC value in D using Q[31:28] and polynomial in S. Like CRCBIT x 4. Q = Q << 4. Use 'REP #n,#1'+SETQ+CRCNIB+CRCNIB+CRCNIB...
133	MUXNITS D,{#}S	For each non-zero bit pair in S, copy that bit pair into the corresponding D bits, else leave that D bit pair the same.

134	MUXNIBS D,{#}S	For each non-zero nibble in S, copy that nibble into the corresponding D nibble, else leave that D nibble the same.
135	MUXQ D,{#}S	Used after SETQ. For each '1' bit in Q, copy the corresponding bit in S into D. $D = (D \& !Q) \mid (S \& Q)$.
136	MOVBYTES D,{#}S	Move bytes within D, per S. $D = \{D.BYTE[S[7:6]], D.BYTE[S[5:4]], D.BYTE[S[3:2]], D.BYTE[S[1:0]]\}$.
137	MUL D,{#}S {WZ}	$D = \text{unsigned}(D[15:0] * S[15:0])$. $Z = (S == 0) \mid (D == 0)$.
138	MULS D,{#}S {WZ}	$D = \text{signed}(D[15:0] * S[15:0])$. $Z = (S == 0) \mid (D == 0)$.
139	SCA D,{#}S {WZ}	Next instruction's S value = $\text{unsigned}(D[15:0] * S[15:0]) \gg 16$. *
140	SCAS D,{#}S {WZ}	Next instruction's S value = $\text{signed}(D[15:0] * S[15:0]) \gg 14$. In this scheme, \$4000 = 1.0 and \$C000 = -1.0. *
141	ADDPIX D,{#}S	Add bytes of S into bytes of D, with \$FF saturation.
142	MULPIX D,{#}S	Multiply bytes of S into bytes of D, where \$FF = 1.0 and \$00 = 0.0.
143	BLNPIX D,{#}S	Alpha-blend bytes of S into bytes of D, using SETPIV value.
144	MIXPIX D,{#}S	Mix bytes of S into bytes of D, using SETPIX and SETPIV values.
145	ADDCT1 D,{#}S	Set CT1 event to trigger on $CT = D + S$. Adds S into D.
146	ADDCT2 D,{#}S	Set CT2 event to trigger on $CT = D + S$. Adds S into D.
147	ADDCT3 D,{#}S	Set CT3 event to trigger on $CT = D + S$. Adds S into D.
148	WMLONG D,{#}S/P	Write only non-\$00 bytes in $D[31:0]$ to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.
149	RQPIN D,{#}S {WC}	Read smart pin $S[5:0]$ result "Z" into D, don't acknowledge smart pin ("Q" in RQPIN means "quiet"). C = modal result.
150	RDPIN D,{#}S {WC}	Read smart pin $S[5:0]$ result "Z" into D, acknowledge smart pin. C = modal result.
151	RDLUT D,{#}S/P {WC/WZ/WCZ}	Read data from LUT address {#}S/PTRx into D. C = MSB of data. *
152	RDBYTE D,{#}S/P {WC/WZ/WCZ}	Read zero-extended byte from hub address {#}S/PTRx into D. C = MSB of byte. *

153	RDWORD D,{#}S/P {WC/WZ/WCZ}	Read zero-extended word from hub address {#}S/PTRx into D. C = MSB of word. *
154	RDLONG D,{#}S/P {WC/WZ/WCZ}	Read long from hub address {#}S/PTRx into D. C = MSB of long. * Prior SETQ/SETQ2 invokes cog/LUT block transfer.
155	POPA D {WC/WZ/WCZ}	Read long from hub address --PTRA into D. C = MSB of long. *
156	POPB D {WC/WZ/WCZ}	Read long from hub address --PTRB into D. C = MSB of long. *
157	CALLD D,{#}S {WC/WZ/WCZ}	Call to S** by writing {C, Z, 10'b0, PC[19:0]} to D. C = S[31], Z = S[30].
158	RESI3	Resume from INT3. (CALLD \$1F0,\$1F1 WCZ)
159	RESI2	Resume from INT2. (CALLD \$1F2,\$1F3 WCZ)
160	RESI1	Resume from INT1. (CALLD \$1F4,\$1F5 WCZ)
161	RESI0	Resume from INT0. (CALLD \$1FE,\$1FF WCZ)
162	RETI3	Return from INT3. (CALLD \$1FF,\$1F1 WCZ)
163	RETI2	Return from INT2. (CALLD \$1FF,\$1F3 WCZ)
164	RETI1	Return from INT1. (CALLD \$1FF,\$1F5 WCZ)
165	RETI0	Return from INT0. (CALLD \$1FF,\$1FF WCZ)
166	CALLPA {#}D,{#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PA.
167	CALLPB {#}D,{#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PB.
168	DJZ D,{#}S	Decrement D and jump to S** if result is zero.
169	DJNZ D,{#}S	Decrement D and jump to S** if result is not zero.
170	DJF D,{#}S	Decrement D and jump to S** if result is \$FFFF_FFFF.
171	DJNF D,{#}S	Decrement D and jump to S** if result is not \$FFFF_FFFF.
172	IJZ D,{#}S	Increment D and jump to S** if result is zero.
173	IJNZ D,{#}S	Increment D and jump to S** if result is not zero.
174	TJZ D,{#}S	Test D and jump to S** if D is zero.
175	TJNZ D,{#}S	Test D and jump to S** if D is not zero.

176	TJF D,{#}S	Test D and jump to S** if D is full (D = \$FFFF_FFFF).
177	TJNF D,{#}S	Test D and jump to S** if D is not full (D != \$FFFF_FFFF).
178	TJS D,{#}S	Test D and jump to S** if D is signed (D[31] = 1).
179	TJNS D,{#}S	Test D and jump to S** if D is not signed (D[31] = 0).
180	TJV D,{#}S	Test D and jump to S** if D overflowed (D[31] != C, C = 'correct sign' from last addition/subtraction).
181	JINT {#}S	Jump to S** if INT event flag is set.
182	JCT1 {#}S	Jump to S** if CT1 event flag is set.
183	JCT2 {#}S	Jump to S** if CT2 event flag is set.
184	JCT3 {#}S	Jump to S** if CT3 event flag is set.
185	JSE1 {#}S	Jump to S** if SE1 event flag is set.
186	JSE2 {#}S	Jump to S** if SE2 event flag is set.
187	JSE3 {#}S	Jump to S** if SE3 event flag is set.
188	JSE4 {#}S	Jump to S** if SE4 event flag is set.
189	JPAT {#}S	Jump to S** if PAT event flag is set.
190	JFBW {#}S	Jump to S** if FBW event flag is set.
191	JXMT {#}S	Jump to S** if XMT event flag is set.
192	JXFI {#}S	Jump to S** if XFI event flag is set.
193	JXRO {#}S	Jump to S** if XRO event flag is set.
194	JXRL {#}S	Jump to S** if XRL event flag is set.
195	JATN {#}S	Jump to S** if ATN event flag is set.
196	JQMT {#}S	Jump to S** if QMT event flag is set.
197	JNINT {#}S	Jump to S** if INT event flag is clear.
198	JNCT1 {#}S	Jump to S** if CT1 event flag is clear.

199	JNCT2 {#}S	Jump to S** if CT2 event flag is clear.
200	JNCT3 {#}S	Jump to S** if CT3 event flag is clear.
201	JNSE1 {#}S	Jump to S** if SE1 event flag is clear.
202	JNSE2 {#}S	Jump to S** if SE2 event flag is clear.
203	JNSE3 {#}S	Jump to S** if SE3 event flag is clear.
204	JNSE4 {#}S	Jump to S** if SE4 event flag is clear.
205	JNPAT {#}S	Jump to S** if PAT event flag is clear.
206	JNFBW {#}S	Jump to S** if FBW event flag is clear.
207	JNXMT {#}S	Jump to S** if XMT event flag is clear.
208	JNXFI {#}S	Jump to S** if XFI event flag is clear.
209	JNXRO {#}S	Jump to S** if XRO event flag is clear.
210	JNXRL {#}S	Jump to S** if XRL event flag is clear.
211	JNATN {#}S	Jump to S** if ATN event flag is clear.
212	JNQMT {#}S	Jump to S** if QMT event flag is clear.
213	<empty> {#}D,{#}S	<empty>
214	<empty> {#}D,{#}S	<empty>
215	SETPAT {#}D,{#}S	Set pin pattern for PAT event. C selects INA/INB, Z selects =/!=, D provides mask value, S provides match value.
216	AKPIN {#}S	Acknowledge smart pins S[10:6]+S[5:0]..S[5:0]. Wraps within A/B pins. Prior SETQ overrides S[10:6].
217	WRPIN {#}D,{#}S	Set mode of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].
218	WXPIN {#}D,{#}S	Set "X" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].
219	WYPIN {#}D,{#}S	Set "Y" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].

220	WRLUT {#}D,{#}S/P	Write D to LUT address {#}S/PTRx.
221	WRBYTE {#}D,{#}S/P	Write byte in D[7:0] to hub address {#}S/PTRx.
222	WRWORD {#}D,{#}S/P	Write word in D[15:0] to hub address {#}S/PTRx.
223	WRLONG {#}D,{#}S/P	Write long in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.
224	PUSHA {#}D	Write long in D[31:0] to hub address PTRB++.
225	PUSHB {#}D	Write long in D[31:0] to hub address PTRB++.
226	RDFAST {#}D,{#}S	Begin new fast hub read via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.
227	WRFAST {#}D,{#}S	Begin new fast hub write via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.
228	FBLOCK {#}D,{#}S	Set next block for when block wraps. D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.
229	XINIT {#}D,{#}S	Issue streamer command immediately, zeroing phase.
230	XSTOP	Stop streamer immediately.
231	XZERO {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, zeroing phase.
232	XCONT {#}D,{#}S	Buffer new streamer command to be issued on final NCO rollover of current command, continuing phase.
233	REP {#}D,{#}S	Execute next D[8:0] instructions S times. If S = 0, repeat instructions infinitely. If D[8:0] = 0, nothing repeats.
234	COGINIT {#}D,{#}S {WC}	Start cog selected by D. S[19:0] sets hub startup address and PTRB of cog. Prior SETQ sets PTRB of cog.
235	QMUL {#}D,{#}S	Begin CORDIC unsigned multiplication of D * S. GETQX/GETQY retrieves lower/upper product.
236	QDIV {#}D,{#}S	Begin CORDIC unsigned division of {SETQ value or 32'b0, D} / S. GETQX/GETQY retrieves quotient/remainder.
237	QFRAC {#}D,{#}S	Begin CORDIC unsigned division of {D, SETQ value or 32'b0} / S. GETQX/GETQY retrieves quotient/remainder.

238	QSQRT {#}D,{#}S	Begin CORDIC square root of {S, D}. GETQX retrieves root.
239	QROTATE {#}D,{#}S	Begin CORDIC rotation of point (D, SETQ value or 32'b0) by angle S. GETQX/GETQY retrieves X/Y.
240	QVECTOR {#}D,{#}S	Begin CORDIC vectoring of point (D, S). GETQX/GETQY retrieves length/angle.
241	HUBSET {#}D	Set hub configuration to D.
242	COGID {#}D {WC}	If D is register and no WC, get cog ID (0 to 15) into D. If WC, check status of cog D[3:0], C = 1 if on.
243	COGSTOP {#}D	Stop cog D[3:0].
244	LOCKNEW D {WC}	Request a LOCK. D will be written with the LOCK number (0 to 15). C = 1 if no LOCK available.
245	LOCKRET {#}D	Return LOCK D[3:0] for reallocation.
246	LOCKTRY {#}D {WC}	Try to get LOCK D[3:0]. C = 1 if got LOCK. LOCKREL releases LOCK. LOCK is also released if owner cog stops or restarts.
247	LOCKREL {#}D {WC}	Release LOCK D[3:0]. If D is a register and WC, get current/last cog id of LOCK owner into D and LOCK status into C.
248	QLOG {#}D	Begin CORDIC number-to-logarithm conversion of D. GETQX retrieves log {5'whole_exponent, 27'fractional_exponent}.
249	QEXP {#}D	Begin CORDIC logarithm-to-number conversion of D. GETQX retrieves number.
250	RFBYTE D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended byte from FIFO into D. C = MSB of byte. *
251	RFWORD D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended word from FIFO into D. C = MSB of word. *
252	RFLONG D {WC/WZ/WCZ}	Used after RDFAST. Read long from FIFO into D. C = MSB of long. *
253	RFVAR D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended 1..4-byte value from FIFO into D. C = 0. *
254	RFVARS D {WC/WZ/WCZ}	Used after RDFAST. Read sign-extended 1..4-byte value from FIFO into D. C = MSB of value. *
255	WFBYTE {#}D	Used after WRFAST. Write byte in D[7:0] into FIFO.

256	WFWORD {#}D	Used after WRFAST. Write word in D[15:0] into FIFO.
257	WFLONG {#}D	Used after WRFAST. Write long in D[31:0] into FIFO.
258	GETQX D {WC/WZ/WCZ}	Retrieve CORDIC result X into D. Waits, in case result not ready. C = X[31]. *
259	GETQY D {WC/WZ/WCZ}	Retrieve CORDIC result Y into D. Waits, in case result not ready. C = Y[31]. *
260	GETCT D {WC}	Get CT[31:0] or CT[63:32] if WC into D. GETCT WC + GETCT gets full CT. CT=0 on reset, CT++ on every clock. C = same.
261	GETRND D {WC/WZ/WCZ}	Get RND into D/C/Z. RND is the PRNG that updates on every clock. D = RND[31:0], C = RND[31], Z = RND[30], unique per cog.
262	GETRND WC/WZ/WCZ	Get RND into C/Z. C = RND[31], Z = RND[30], unique per cog.
263	SETDACS {#}D	DAC3 = D[31:24], DAC2 = D[23:16], DAC1 = D[15:8], DAC0 = D[7:0].
264	SETXFRQ {#}D	Set streamer NCO frequency to D.
265	GETXACC D	Get the streamer's Goertzel X accumulator into D and the Y accumulator into the next instruction's S, clear accumulators.
266	WAITX {#}D {WC/WZ/WCZ}	Wait 2 + D clocks if no WC/WZ/WCZ. If WC/WZ/WCZ, wait 2 + (D & RND) clocks. C/Z = 0.
267	SETSE1 {#}D	Set SE1 event configuration to D[8:0].
268	SETSE2 {#}D	Set SE2 event configuration to D[8:0].
269	SETSE3 {#}D	Set SE3 event configuration to D[8:0].
270	SETSE4 {#}D	Set SE4 event configuration to D[8:0].
271	POLLINT {WC/WZ/WCZ}	Get INT event flag into C/Z, then clear it.
272	POLLCT1 {WC/WZ/WCZ}	Get CT1 event flag into C/Z, then clear it.
273	POLLCT2 {WC/WZ/WCZ}	Get CT2 event flag into C/Z, then clear it.
274	POLLCT3 {WC/WZ/WCZ}	Get CT3 event flag into C/Z, then clear it.
275	POLLSE1 {WC/WZ/WCZ}	Get SE1 event flag into C/Z, then clear it.
276	POLLSE2 {WC/WZ/WCZ}	Get SE2 event flag into C/Z, then clear it.
277	POLLSE3 {WC/WZ/WCZ}	Get SE3 event flag into C/Z, then clear it.

278	POLLSE4 {WC/WZ/WCZ}	Get SE4 event flag into C/Z, then clear it.
279	POLLPAT {WC/WZ/WCZ}	Get PAT event flag into C/Z, then clear it.
280	POLLFBW {WC/WZ/WCZ}	Get FBW event flag into C/Z, then clear it.
281	POLLXMT {WC/WZ/WCZ}	Get XMT event flag into C/Z, then clear it.
282	POLLXFI {WC/WZ/WCZ}	Get XFI event flag into C/Z, then clear it.
283	POLLXRO {WC/WZ/WCZ}	Get XRO event flag into C/Z, then clear it.
284	POLLXRL {WC/WZ/WCZ}	Get XRL event flag into C/Z, then clear it.
285	POLLATN {WC/WZ/WCZ}	Get ATN event flag into C/Z, then clear it.
286	POLLQMT {WC/WZ/WCZ}	Get QMT event flag into C/Z, then clear it.
287	WAITINT {WC/WZ/WCZ}	Wait for INT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
288	WAITCT1 {WC/WZ/WCZ}	Wait for CT1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
289	WAITCT2 {WC/WZ/WCZ}	Wait for CT2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
290	WAITCT3 {WC/WZ/WCZ}	Wait for CT3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
291	WAITSE1 {WC/WZ/WCZ}	Wait for SE1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
292	WAITSE2 {WC/WZ/WCZ}	Wait for SE2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
293	WAITSE3 {WC/WZ/WCZ}	Wait for SE3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
294	WAITSE4 {WC/WZ/WCZ}	Wait for SE4 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
295	WAITPAT {WC/WZ/WCZ}	Wait for PAT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.

296	WAITFBW {WC/WZ/WCZ}	Wait for FBW event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
297	WAITXMT {WC/WZ/WCZ}	Wait for XMT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
298	WAITXFI {WC/WZ/WCZ}	Wait for XFI event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
299	WAITXRO {WC/WZ/WCZ}	Wait for XRO event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
300	WAITXRL {WC/WZ/WCZ}	Wait for XRL event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
301	WAITATN {WC/WZ/WCZ}	Wait for ATN event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.
302	ALLOWI	Allow interrupts (default).
303	STALLI	Stall Interrupts.
304	TRGINT1	Trigger INT1, regardless of STALLI mode.
305	TRGINT2	Trigger INT2, regardless of STALLI mode.
306	TRGINT3	Trigger INT3, regardless of STALLI mode.
307	NIXINT1	Cancel INT1.
308	NIXINT2	Cancel INT2.
309	NIXINT3	Cancel INT3.
310	SETINT1 {#}D	Set INT1 source to D[3:0].
311	SETINT2 {#}D	Set INT2 source to D[3:0].
312	SETINT3 {#}D	Set INT3 source to D[3:0].
313	SETQ {#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set block transfer. Also used before MUXQ/COGINIT/QDIV/QFRAC/QROTATE/WAITxxx.
314	SETQ2 {#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set LUT block transfer.
315	PUSH {#}D	Push D onto stack.

316	POP D {WC/WZ/WCZ}	Pop stack (K). D = K. C = K[31]. *
317	JMP D {WC/WZ/WCZ}	Jump to D. C = D[31], Z = D[30], PC = D[19:0].
318	CALL D {WC/WZ/WCZ}	Call to D by pushing {C, Z, 10'b0, PC[19:0]} onto stack. C = D[31], Z = D[30], PC = D[19:0].
319	RET {WC/WZ/WCZ}	Return by popping stack (K). C = K[31], Z = K[30], PC = K[19:0].
320	CALLA D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR A++. C = D[31], Z = D[30], PC = D[19:0].
321	RETA {WC/WZ/WCZ}	Return by reading hub long (L) at --PTR A. C = L[31], Z = L[30], PC = L[19:0].
322	CALLB D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR B++. C = D[31], Z = D[30], PC = D[19:0].
323	RETB {WC/WZ/WCZ}	Return by reading hub long (L) at --PTR B. C = L[31], Z = L[30], PC = L[19:0].
324	JMPREL {#}D	Jump ahead/back by D instructions. For cogex, PC += D[19:0]. For hubex, PC += D[17:0] << 2.
325	SKIP {#}D	Skip instructions per D. Subsequent instructions 0..31 get cancelled for each '1' bit in D[0]..D[31].
326	SKIPF {#}D	Skip cog/LUT instructions fast per D. Like SKIP, but instead of cancelling instructions, the PC leaps over them.
327	EXECF {#}D	Jump to D[9:0] in cog/LUT and set SKIPF pattern to D[31:10]. PC = {10'b0, D[9:0]}.
328	GETPTR D	Get current FIFO hub pointer into D.
329	GETBRK D WC/WZ/WCZ	Get breakpoint/cog status into D according to WC/WZ/WCZ. See documentation for details.
330	COGBRK {#}D	If in debug ISR, trigger asynchronous breakpoint in cog D[3:0]. Cog D[3:0] must have asynchronous breakpoint enabled.
331	BRK {#}D	If in debug ISR, set next break condition to D. Else, set BRK code to D[7:0] and unconditionally trigger BRK interrupt, if enabled.
332	SETLUTS {#}D	If D[0] = 1 then enable LUT sharing, where LUT writes within the adjacent odd/even companion cog are copied to this cog's LUT.
333	SETCY {#}D	Set the colorspace converter "CY" parameter to D[31:0].

334	SETCI {#}D	Set the colorspace converter "CI" parameter to D[31:0].
335	SETCQ {#}D	Set the colorspace converter "CQ" parameter to D[31:0].
336	SETCFRQ {#}D	Set the colorspace converter "CFRQ" parameter to D[31:0].
337	SETCMOD {#}D	Set the colorspace converter "CMOD" parameter to D[8:0].
338	SETPIV {#}D	Set BLNPIX/MIXPIX blend factor to D[7:0].
339	SETPIX {#}D	Set MIXPIX mode to D[5:0].
340	COGATN {#}D	Strobe "attention" of all cogs whose corresponding bits are high in D[15:0].
341	TESTP {#}D WC/WZ	Test IN bit of pin D[5:0], write to C/Z. $C/Z = IN[D[5:0]]$.
342	TESTPN {#}D WC/WZ	Test !IN bit of pin D[5:0], write to C/Z. $C/Z = !IN[D[5:0]]$.
343	TESTP {#}D ANDC/ANDZ	Test IN bit of pin D[5:0], AND into C/Z. $C/Z = C/Z \text{ AND } IN[D[5:0]]$.
344	TESTPN {#}D ANDC/ANDZ	Test !IN bit of pin D[5:0], AND into C/Z. $C/Z = C/Z \text{ AND } !IN[D[5:0]]$.
345	TESTP {#}D ORC/ORZ	Test IN bit of pin D[5:0], OR into C/Z. $C/Z = C/Z \text{ OR } IN[D[5:0]]$.
346	TESTPN {#}D ORC/ORZ	Test !IN bit of pin D[5:0], OR into C/Z. $C/Z = C/Z \text{ OR } !IN[D[5:0]]$.
347	TESTP {#}D XORC/XORZ	Test IN bit of pin D[5:0], XOR into C/Z. $C/Z = C/Z \text{ XOR } IN[D[5:0]]$.
348	TESTPN {#}D XORC/XORZ	Test !IN bit of pin D[5:0], XOR into C/Z. $C/Z = C/Z \text{ XOR } !IN[D[5:0]]$.
349	DIRL {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = 0. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
350	DIRH {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = 1. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
351	DIRC {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
352	DIRNC {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = !C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
353	DIRZ {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
354	DIRNZ {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.

355	DIRRND {#}D {WCZ}	DIR bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
356	DIRNOT {#}D {WCZ}	Toggle DIR bits of pins D[10:6]+D[5:0]..D[5:0]. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.
357	OUTL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
358	OUTH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
359	OUTC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
360	OUTNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
361	OUTZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
362	OUTNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
363	OUTRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
364	OUTNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
365	FLTL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
366	FLTH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
367	FLTC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
368	FLTNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
369	FLTZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.

370	FLTNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
371	FLTRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
372	FLTNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
373	DRVL {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 0. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
374	DRVH {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = 1. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
375	DRVC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
376	DRVNC {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
377	DRVZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
378	DRVNZ {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = !Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
379	DRVRND {#}D {WCZ}	OUT bits of pins D[10:6]+D[5:0]..D[5:0] = RNDs. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
380	DRVNOT {#}D {WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0]..D[5:0]. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.
381	SPLITB D	Split every 4th bit of D into bytes. D = {D[31], D[27], D[23], D[19], ...D[12], D[8], D[4], D[0]}.
382	MERGE8 D	Merge bits of bytes in D. D = {D[31], D[23], D[15], D[7], ...D[24], D[16], D[8], D[0]}.
383	SPLITW D	Split odd/even bits of D into words. D = {D[31], D[29], D[27], D[25], ...D[6], D[4], D[2], D[0]}.
384	MERGEW D	Merge bits of words in D. D = {D[31], D[15], D[30], D[14], ...D[17], D[1], D[16], D[0]}.

385	SEUSSF D	Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Forward pattern.
386	SEUSSR D	Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Reverse pattern.
387	RGBSQZ D	Squeeze 8:8:8 RGB value in D[31:8] into 5:6:5 value in D[15:0]. D = {15'b0, D[31:27], D[23:18], D[15:11]}.
388	RGBEXP D	Expand 5:6:5 RGB value in D[15:0] into 8:8:8 value in D[31:8]. D = {D[15:11,15:13], D[10:5,10:9], D[4:0,4:2], 8'b0}.
389	SPLITB D	Split every 4th bit of D into bytes. D = {D[31], D[27], D[23], D[19], ...D[12], D[8], D[4], D[0]}.
390	REV D	Reverse D bits. D = D[0:31].
391	RCZR D {WC/WZ/WCZ}	Rotate C,Z right through D. D = {C, Z, D[31:2]}. C = D[1], Z = D[0].
392	RCZL D {WC/WZ/WCZ}	Rotate C,Z left through D. D = {D[29:0], C, Z}. C = D[31], Z = D[30].
393	WRC D	Write 0 or 1 to D, according to C. D = {31'b0, C}.
394	WRNC D	Write 0 or 1 to D, according to !C. D = {31'b0, !C}.
395	WRZ D	Write 0 or 1 to D, according to Z. D = {31'b0, Z}.
396	WRNZ D	Write 0 or 1 to D, according to !Z. D = {31'b0, !Z}.
397	MODCZ c,z {WC/WZ/WCZ}	Modify C and Z according to cccc and zzzz. C = cccc[{C,Z}], Z = zzzz[{C,Z}].
398	MODC c {WC}	Modify C according to cccc. C = cccc[{C,Z}].
399	MODZ z {WZ}	Modify Z according to zzzz. Z = zzzz[{C,Z}].
400	SETSCP {#}D	Set four-channel oscilloscope enable to D[6] and set input pin base to D[5:2].
401	GETSCP D	Get four-channel oscilloscope samples into D. D = {ch3[7:0],ch2[7:0],ch1[7:0],ch0[7:0]}.
402	JMP #{\}A	Jump to A. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
403	CALL #{\}A	Call to A by pushing {C, Z, 10'b0, PC[19:0]} onto stack. If R = 1 then PC += A, else PC = A. "\" forces R = 0.
404	CALLA #{\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR+++. If R = 1 then PC += A, else PC = A. "\" forces R = 0.

405	CALLB #{\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTRB++. If R = 1 then PC += A, else PC = A. "\ " forces R = 0.
406	CALLD PA/PB/PTRA/PTRB,#{\}A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to PA/PB/PTRA/PTRB (per W). If R = 1 then PC += A, else PC = A. "\ " forces R = 0.
407	LOC PA/PB/PTRA/PTRB,#{\}A	Get {12'b0, address[19:0]} into PA/PB/PTRA/PTRB (per W). If R = 1, address = PC + A, else address = A. "\ " forces R = 0.
408	AUGS #n	Queue #n to be used as upper 23 bits for next #S occurrence, so that the next 9-bit #S will be augmented to 32 bits.
409	AUGD #n	Queue #n to be used as upper 23 bits for next #D occurrence, so that the next 9-bit #D will be augmented to 32 bits.
410	_RET_ <inst> <ops>	Execute <inst> always and return if no branch. If <inst> is not branching then return by popping stack[19:0] into PC.
411	IF_NC_AND_NZ <inst> <ops>	Execute <inst> if C = 0 and Z = 0.
412	IF_NZ_AND_NC <inst> <ops>	Execute <inst> if C = 0 and Z = 0.
413	IF_GT <inst> <ops>	Execute <inst> if C = 0 and Z = 0, or if 'greater than' after a comparison/subtraction.
414	IF_A <inst> <ops>	Execute <inst> if C = 0 and Z = 0, or if 'above' after a comparison/subtraction.
415	IF_00 <inst> <ops>	Execute <inst> if C = 0 and Z = 0.
416	IF_NC_AND_Z <inst> <ops>	Execute <inst> if C = 0 and Z = 1.
417	IF_Z_AND_NC <inst> <ops>	Execute <inst> if C = 0 and Z = 1.
418	IF_01 <inst> <ops>	Execute <inst> if C = 0 and Z = 1.
419	IF_NC <inst> <ops>	Execute <inst> if C = 0.
420	IF_GE <inst> <ops>	Execute <inst> if C = 0, or if 'greater than or equal' after a comparison/subtraction.
421	IF_AE <inst> <ops>	Execute <inst> if C = 0, or if 'above or equal' after a comparison/subtraction.
422	IF_0X <inst> <ops>	Execute <inst> if C = 0.
423	IF_C_AND_NZ <inst> <ops>	Execute <inst> if C = 1 and Z = 0.
424	IF_NZ_AND_C <inst> <ops>	Execute <inst> if C = 1 and Z = 0.

425	IF_10	<inst> <ops>	Execute <inst> if C = 1 and Z = 0.
426	IF_NZ	<inst> <ops>	Execute <inst> if Z = 0.
427	IF_NE	<inst> <ops>	Execute <inst> if Z = 0, or if 'not equal' after a comparison/subtraction.
428	IF_X0	<inst> <ops>	Execute <inst> if Z = 0.
429	IF_C_NE_Z	<inst> <ops>	Execute <inst> if C != Z.
430	IF_Z_NE_C	<inst> <ops>	Execute <inst> if C != Z.
431	IF_DIFF	<inst> <ops>	Execute <inst> if C != Z.
432	IF_NC_OR_NZ	<inst> <ops>	Execute <inst> if C = 0 or Z = 0.
433	IF_NZ_OR_NC	<inst> <ops>	Execute <inst> if C = 0 or Z = 0.
434	IF_NOT_11	<inst> <ops>	Execute <inst> if C = 0 or Z = 0.
435	IF_C_AND_Z	<inst> <ops>	Execute <inst> if C = 1 and Z = 1.
436	IF_Z_AND_C	<inst> <ops>	Execute <inst> if C = 1 and Z = 1.
437	IF_11	<inst> <ops>	Execute <inst> if C = 1 and Z = 1.
438	IF_C_EQ_Z	<inst> <ops>	Execute <inst> if C = Z.
439	IF_Z_EQ_C	<inst> <ops>	Execute <inst> if C = Z.
440	IF_SAME	<inst> <ops>	Execute <inst> if C = Z.
441	IF_Z	<inst> <ops>	Execute <inst> if Z = 1.
442	IF_E	<inst> <ops>	Execute <inst> if Z = 1, or if 'equal' after a comparison/subtraction.
443	IF_X1	<inst> <ops>	Execute <inst> if Z = 1.
444	IF_NC_OR_Z	<inst> <ops>	Execute <inst> if C = 0 or Z = 1.
445	IF_Z_OR_NC	<inst> <ops>	Execute <inst> if C = 0 or Z = 1.
446	IF_NOT_10	<inst> <ops>	Execute <inst> if C = 0 or Z = 1.
447	IF_C	<inst> <ops>	Execute <inst> if C = 1.
448	IF_LT	<inst> <ops>	Execute <inst> if C = 1, or if 'less than' after a comparison/subtraction.

449	IF_B <inst> <ops>	Execute <inst> if C = 1, or if 'below' after a comparison/subtraction.
450	IF_1X <inst> <ops>	Execute <inst> if C = 1.
451	IF_C_OR_NZ <inst> <ops>	Execute <inst> if C = 1 or Z = 0.
452	IF_NZ_OR_C <inst> <ops>	Execute <inst> if C = 1 or Z = 0.
453	IF_NOT_01 <inst> <ops>	Execute <inst> if C = 1 or Z = 0.
454	IF_C_OR_Z <inst> <ops>	Execute <inst> if C = 1 or Z = 1.
455	IF_Z_OR_C <inst> <ops>	Execute <inst> if C = 1 or Z = 1.
456	IF_LE <inst> <ops>	Execute <inst> if C = 1 or Z = 1, or if 'less than or equal' after a comparison/subtraction.
457	IF_BE <inst> <ops>	Execute <inst> if C = 1 or Z = 1, or if 'below or equal' after a comparison/subtraction.
458	IF_NOT_00 <inst> <ops>	Execute <inst> if C = 1 or Z = 1.
459	<inst> <ops>	Execute <inst> always. This is the default when no instruction prefix is expressed.
460	_CLR	C/Z = 0
461	_NC_AND_NZ	C/Z = !C AND !Z
462	_NZ_AND_NC	C/Z = !C AND !Z
463	_GT	C/Z = !C AND !Z, or 'greater than' after a comparison/subtraction.
464	_NC_AND_Z	C/Z = !C AND Z
465	_Z_AND_NC	C/Z = !C AND Z
466	_NC	C/Z = !C
467	_GE	C/Z = !C, or 'greater than or equal' after a comparison/subtraction.
468	_C_AND_NZ	C/Z = C AND !Z
469	_NZ_AND_C	C/Z = C AND !Z
470	_NZ	C/Z = !Z

471	<code>_NE</code>	$C/Z = !Z$, or 'not equal' after a comparison/subtraction.
472	<code>_C_NE_Z</code>	$C/Z = C \text{ NOT_EQUAL_TO } Z$
473	<code>_Z_NE_C</code>	$C/Z = C \text{ NOT_EQUAL_TO } Z$
474	<code>_NC_OR_NZ</code>	$C/Z = !C \text{ OR } !Z$
475	<code>_NZ_OR_NC</code>	$C/Z = !C \text{ OR } !Z$
476	<code>_C_AND_Z</code>	$C/Z = C \text{ AND } Z$
477	<code>_Z_AND_C</code>	$C/Z = C \text{ AND } Z$
478	<code>_C_EQ_Z</code>	$C/Z = C \text{ EQUAL_TO } Z$
479	<code>_Z_EQ_C</code>	$C/Z = C \text{ EQUAL_TO } Z$
480	<code>_Z</code>	$C/Z = Z$
481	<code>_E</code>	$C/Z = Z$, or 'equal' after a comparison/subtraction.
482	<code>_NC_OR_Z</code>	$C/Z = !C \text{ OR } Z$
483	<code>_Z_OR_NC</code>	$C/Z = !C \text{ OR } Z$
484	<code>_C</code>	$C/Z = C$
485	<code>_LT</code>	$C/Z = C$, or 'less than' after a comparison/subtraction.
486	<code>_C_OR_NZ</code>	$C/Z = C \text{ OR } !Z$
487	<code>_NZ_OR_C</code>	$C/Z = C \text{ OR } !Z$
488	<code>_C_OR_Z</code>	$C/Z = C \text{ OR } Z$
489	<code>_Z_OR_C</code>	$C/Z = C \text{ OR } Z$
490	<code>_LE</code>	$C/Z = C \text{ OR } Z$, or 'less than or equal' after a comparison/subtraction.
491	<code>_SET</code>	$C/Z = 1$
		$C/Z = 1$

Math and Logic Instructions				
Instruction			Description	Clocks Reg, LUT, & Hub
ABS	D	{WC/WZ/WCZ}	Get absolute value of D into D. $D = \text{ABS}(D)$. $C = D[31]$. *	2
ABS	D, {#}S	{WC/WZ/WCZ}	Get absolute value of S into D. $D = \text{ABS}(S)$. $C = S[31]$. *	2
ADD	D, {#}S	{WC/WZ/WCZ}	Add S into D. $D = D + S$. $C = \text{carry of } (D + S)$. *	2
ADD S	D, {#}S	{WC/WZ/WCZ}	Add S into D, signed. $D = D + S$. $C = \text{correct sign of } (D + S)$. *	2
ADD SX	D, {#}S	{WC/WZ/WCZ}	Add (S + C) into D, signed and extended. $D = D + S + C$. $C = \text{correct sign of } (D + S + C)$. $Z = Z \text{ AND } (\text{result} == 0)$.	2
ADD X	D, {#}S	{WC/WZ/WCZ}	Add (S + C) into D, extended. $D = D + S + C$. $C = \text{carry of } (D + S + C)$. $Z = Z \text{ AND } (\text{result} == 0)$.	2
AND	D, {#}S	{WC/WZ/WCZ}	AND S into D. $D = D \& S$. $C = \text{parity of result}$. *	2
AND N	D, {#}S	{WC/WZ/WCZ}	AND !S into D. $D = D \& !S$. $C = \text{parity of result}$. *	2
BIT C	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = C$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BITH	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = 1$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT L	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = 0$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT NC	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = !C$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT NOT	D, {#}S	{WCZ}	Toggle bits $D[S[9:5]+S[4:0]-S[4:0]]$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT NZ	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = !Z$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT RND	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = \text{RNDs}$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BIT Z	D, {#}S	{WCZ}	Bits $D[S[9:5]+S[4:0]-S[4:0]] = Z$. Other bits unaffected. Prior SETQ overrides $S[9:5]$. $C, Z = \text{original } D[S[4:0]]$.	2
BMASK	D		Get LSB-justified bit mask of size $(D[4:0] + 1)$ into D. $D = (S0000_0002 \ll D[4:0]) - 1$.	2
BMASK	D, {#}S		Get LSB-justified bit mask of size $(S[4:0] + 1)$ into D. $D = (S0000_0002 \ll S[4:0]) - 1$.	2
CMP	D, {#}S	{WC/WZ/WCZ}	Compare D to S. $C = \text{borrow of } (D - S)$. $Z = (D == S)$.	2

CMPM	D, {#}S	{WC/WZ/WCZ}	Compare D to S, get MSB of difference into C. C = MSB of (D - S). Z = (D == S).	2
CMPR	D, {#}S	{WC/WZ/WCZ}	Compare S to D (reverse). C = borrow of (S - D). Z = (D == S).	2
CMPS	D, {#}S	{WC/WZ/WCZ}	Compare D to S, signed. C = correct sign of (D - S). Z = (D == S).	2
CMPSUB	D, {#}S	{WC/WZ/WCZ}	Compare and subtract S from D if D >= S. If D < S then D = D - S and C = 1, else D same and C = 0. *	2
CMPSX	D, {#}S	{WC/WZ/WCZ}	Compare D to (S + C), signed and extended. C = correct sign of (D - (S + C)). Z = Z AND (D == S + C).	2
CMPX	D, {#}S	{WC/WZ/WCZ}	Compare D to (S + C), extended. C = borrow of (D - (S + C)). Z = Z AND (D == S + C).	2
CRCBIT	D, {#}S		Iterate CRC value in D using C and polynomial in S. If (C XOR D[0]) then D = (D >> 1) XOR S, else D = (D >> 1).	2
CRCNIB	D, {#}S		Iterate CRC value in D using Q[31:28] and polynomial in S. Like CRCBIT x 4. Q = Q << 4. Use 'REP #n, #1' SETQ+CRCNIB+CRCNIB+CRCNIB...	2
DECMOD	D, {#}S	{WC/WZ/WCZ}	Decrement with modulus. If D = 0 then D = S and C = 1, else D = D - 1 and C = 0. *	2
DECOD	D		Decode D[4:0] into D. D = 1 << D[4:0].	2
DECOD	D, {#}S		Decode S[4:0] into D. D = 1 << S[4:0].	2
ENCOD	D	{WC/WZ/WCZ}	Get bit position of top-most '1' in D into D. D = position of top '1' in S (0..31). C = (S != 0). *	2
ENCOD	D, {#}S	{WC/WZ/WCZ}	Get bit position of top-most '1' in S into D. D = position of top '1' in S (0..31). C = (S != 0). *	2
FGE	D, {#}S	{WC/WZ/WCZ}	Force D >= S. If D < S then D = S and C = 1, else D same and C = 0. *	2
FGES	D, {#}S	{WC/WZ/WCZ}	Force D >= S, signed. If D < S then D = S and C = 1, else D same and C = 0. *	2
FLE	D, {#}S	{WC/WZ/WCZ}	Force D <= S. If D > S then D = S and C = 1, else D same and C = 0. *	2
FLES	D, {#}S	{WC/WZ/WCZ}	Force D <= S, signed. If D > S then D = S and C = 1, else D same and C = 0. *	2
GETBYTE	D		Get byte established by prior ALTGB instruction into D.	2
GETBYTE	D, {#}S, #N		Get byte N of S into D. D = {24'b0, S.BYTE[N]}.	2
GETNIB	D		Get nibble established by prior ALTGN instruction into D.	2
GETNIB	D, {#}S, #N		Get nibble N of S into D. D = {28'b0, S.NIBBLE[N]}.	2
GETWORD	D		Get word established by prior ALTGW instruction into D.	2
GETWORD	D, {#}S, #N		Get word N of S into D. D = {16'b0, S.WORD[N]}.	2
INCMOD	D, {#}S	{WC/WZ/WCZ}	Increment with modulus. If D = S then D = 0 and C = 1, else D = D + 1 and C = 0. *	2

LOC	$\Gamma A/PB/PTRA/PTRB, \# \{ \backslash \} A$	Get {12b0, address[19:0]} into PA/PB/PTRA/PTRB (per W). If R = 1, address = PC + A, else address = A. "\n" forces R = 0.	2
MERGE B	D	Merge bits of bytes in D. $D = \{D[31], D[23], D[15], D[7], \dots D[24], D[16], D[8], D[0]\}$.	2
MERGE W	D	Merge bits of words in D. $D = \{D[31], D[15], D[30], D[14], \dots D[17], D[1], D[16], D[0]\}$.	2
MODC	c {WC}	Modify C according to cccc. $C = cccc\{C, Z\}$.	2
MODCZ	c, z {WC/WZ/WCZ}	Modify C and Z according to cccc and zzzz. $C = cccc\{C, Z\}$, $Z = zzzz\{C, Z\}$.	2
MODZ	z {WZ}	Modify Z according to zzzz. $Z = zzzz\{C, Z\}$.	2
MOV	D, {#}S {WC/WZ/WCZ}	Move S into D. $D = S$. $C = S[31]$. *	2
MOVBYTS	D, {#}S	Move bytes within D, per S. $D = \{D.BYTE[S[7:6]], D.BYTE[S[5:4]], D.BYTE[S[3:2]], D.BYTE[S[1:0]]\}$.	2
MUL	D, {#}S {WZ}	$D = \text{unsigned}(D[15:0] * S[15:0])$. $Z = (S == 0) \vee (D == 0)$.	2
MULS	D, {#}S {WZ}	$D = \text{signed}(D[15:0] * S[15:0])$. $Z = (S == 0) \vee (D == 0)$.	2
MUXC	D, {#}S {WC/WZ/WCZ}	Mux C into each D bit that is '1' in S. $D = (S \& D) \vee (S \& \{32\{C\}\})$. C = parity of result. *	2
MUXNC	D, {#}S {WC/WZ/WCZ}	Mux !C into each D bit that is '1' in S. $D = (S \& D) \vee (S \& \{32\{!C\}\})$. C = parity of result. *	2
MUXNIBS	D, {#}S	For each non-zero nibble in S, copy that nibble into the corresponding D nibble, else leave that D nibble the same.	2
MUXNITS	D, {#}S	For each non-zero bit pair in S, copy that bit pair into the corresponding D bits, else leave that D bit pair the same.	2
MUXNZ	D, {#}S {WC/WZ/WCZ}	Mux !Z into each D bit that is '1' in S. $D = (S \& D) \vee (S \& \{32\{!Z\}\})$. C = parity of result. *	2
MUXQ	D, {#}S	Used after SETQ. For each '1' bit in Q, copy the corresponding bit in S into D. $D = (D \& !Q) \vee (S \& Q)$.	2
MUXZ	D, {#}S {WC/WZ/WCZ}	Mux Z into each D bit that is '1' in S. $D = (S \& D) \vee (S \& \{32\{Z\}\})$. C = parity of result. *	2
NEG	D {WC/WZ/WCZ}	Negate D. $D = -D$. C = MSB of result. *	2
NEG	D, {#}S {WC/WZ/WCZ}	Negate S into D. $D = -S$. C = MSB of result. *	2
NEGC	D {WC/WZ/WCZ}	Negate D by C. If C = 1 then $D = -D$, else $D = D$. C = MSB of result. *	2
NEGC	D, {#}S {WC/WZ/WCZ}	Negate S by C into D. If C = 1 then $D = -S$, else $D = S$. C = MSB of result. *	2
NEGNC	D {WC/WZ/WCZ}	Negate D by !C. If C = 0 then $D = -D$, else $D = D$. C = MSB of result. *	2
NEGNC	D, {#}S {WC/WZ/WCZ}	Negate S by !C into D. If C = 0 then $D = -S$, else $D = S$. C = MSB of result. *	2
NEGNZ	D {WC/WZ/WCZ}	Negate D by !Z. If Z = 0 then $D = -D$, else $D = D$. C = MSB of result. *	2
NEGNZ	D, {#}S {WC/WZ/WCZ}	Negate S by !Z into D. If Z = 0 then $D = -S$, else $D = S$. C = MSB of result. *	2

NEGZ	D	{WC/WZ/WCZ}	Negate D by Z. If Z = 1 then D = -D, else D = D. C = MSB of result. *	2
NEGZ	D, {#}S	{WC/WZ/WCZ}	Negate S by Z into D. If Z = 1 then D = -S, else D = S. C = MSB of result. *	2
NOT	D	{WC/WZ/WCZ}	Get !D into D. D = !D. C = !D[31]. *	2
NOT	D, {#}S	{WC/WZ/WCZ}	Get !S into D. D = !S. C = !S[31]. *	2
ONES	D	{WC/WZ/WCZ}	Get number of 1's in D into D. D = number of 1's in S (0..32). C = LSB of result. *	2
ONES	D, {#}S	{WC/WZ/WCZ}	Get number of 1's in S into D. D = number of 1's in S (0..32). C = LSB of result. *	2
OR	D, {#}S	{WC/WZ/WCZ}	OR S into D. D = D S. C = parity of result. *	2
RCL	D, {#}S	{WC/WZ/WCZ}	Rotate carry left. D = [63:32] of (([D[31:0], {32{C}}] << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *	2
RCR	D, {#}S	{WC/WZ/WCZ}	Rotate carry right. D = [31:0] of ((([32{C}], D[31:0]) >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *	2
RCZL	D	{WC/WZ/WCZ}	Rotate C,Z left through D. D = {D[29:0], C, Z}. C = D[31], Z = D[30].	2
RCZR	D	{WC/WZ/WCZ}	Rotate C,Z right through D. D = {C, Z, D[31:2]}. C = D[1], Z = D[0].	2
REV	D		Reverse D bits. D = D[0:31].	2
RGBEXP	D		Expand 5:6:5 RGB value in D[15:0] into 8:8:8 value in D[31:8]. D = {D[15:11,15:13], D[10:5,10:9], D[4:0,4:2], 8'b0}.	2
RGBSQZ	D		Squeeze 8:8:8 RGB value in D[31:8] into 5:6:5 value in D[15:0]. D = {15'b0, D[31:27], D[23:18], D[15:11]}.	2
ROL	D, {#}S	{WC/WZ/WCZ}	Rotate left. D = [63:32] of ([D[31:0], D[31:0]] << S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[31]. *	2
ROLBYTE	D		Rotate-left byte established by prior ALTGB instruction into D.	2
ROLBYTE	D, {#}S, #N		Rotate-left byte N of S into D. D = {D[23:0], S.BYTE[N]}.	2
ROLNIB	D		Rotate-left nibble established by prior ALTGN instruction into D.	2
ROLNIB	D, {#}S, #N		Rotate-left nibble N of S into D. D = {D[27:0], S.NIBBLE[N]}.	2
ROLWORD	D		Rotate-left word established by prior ALTGW instruction into D.	2
ROLWORD	D, {#}S, #N		Rotate-left word N of S into D. D = {D[15:0], S.WORD[N]}.	2
ROR	D, {#}S	{WC/WZ/WCZ}	Rotate right. D = [31:0] of ([D[31:0], D[31:0]] >> S[4:0]). C = last bit shifted out if S[4:0] > 0, else D[0]. *	2
SAL	D, {#}S	{WC/WZ/WCZ}	Shift arithmetic left. D = [63:32] of ([D[31:0], {32{D[0]}}] << S[4:0]). C = last bit shifted out if S[4:0] > 0, else	2

			D[31]. *	
SAR	D, {#}S	{WC/WZ/WCZ}	Shift arithmetic right. $D = [31:0]$ of $(([32:D[31]]), D[31:0]) \gg S[4:0]$. $C =$ last bit shifted out if $S[4:0] > 0$, else $D[0]$. *	2
SCA	D, {#}S	{WZ}	Next instruction's S value = unsigned $(D[15:0] * S[15:0]) \gg 16$. *	2
SCAS	D, {#}S	{WZ}	Next instruction's S value = signed $(D[15:0] * S[15:0]) \gg 14$. In this scheme, $S4000 = 1.0$ and $SC000 = -1.0$. *	2
SETBYTE	{#}S		Set $S[7:0]$ into byte established by prior ALTSB instruction.	2
SETBYTE	D, {#}S, #N		Set $S[7:0]$ into byte N in D, keeping rest of D same.	2
SETD	D, {#}S		Set D field of D to $S[8:0]$. $D = \{D[31:18], S[8:0], D[8:0]\}$.	2
SETNIB	{#}S		Set $S[3:0]$ into nibble established by prior ALTSTN instruction.	2
SETNIB	D, {#}S, #N		Set $S[3:0]$ into nibble N in D, keeping rest of D same.	2
SETR	D, {#}S		Set R field of D to $S[8:0]$. $D = \{D[31:28], S[8:0], D[18:0]\}$.	2
SETS	D, {#}S		Set S field of D to $S[8:0]$. $D = \{D[31:9], S[8:0]\}$.	2
SETWORD	{#}S		Set $S[15:0]$ into word established by prior ALTSTW instruction.	2
SETWORD	D, {#}S, #N		Set $S[15:0]$ into word N in D, keeping rest of D same.	2
SEUSSF	D		Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Forward pattern.	2
SEUSSR	D		Relocate and periodically invert bits within D. Returns to original value on 32nd iteration. Reverse pattern.	2
SHL	D, {#}S	{WC/WZ/WCZ}	Shift left. $D = [63:32]$ of $((D[31:0], 32'b0) \ll S[4:0])$. $C =$ last bit shifted out if $S[4:0] > 0$, else $D[31]$. *	2
SHR	D, {#}S	{WC/WZ/WCZ}	Shift right. $D = [31:0]$ of $(([32'b0, D[31:0]]) \gg S[4:0])$. $C =$ last bit shifted out if $S[4:0] > 0$, else $D[0]$. *	?
SIGNX	D, {#}S	{WC/WZ/WCZ}	Sign-extend D from bit $S[4:0]$. $C =$ MSB of result. *	2
SPLITB	D		Split every 4th bit of D into bytes. $D = \{D[31], D[27], D[23], D[19], \dots, D[12], D[8], D[4], D[0]\}$.	2
SPLITW	D		Split odd/even bits of D into words. $D = \{D[31], D[29], D[27], D[25], \dots, D[6], D[4], D[2], D[0]\}$.	2
SUB	D, {#}S	{WC/WZ/WCZ}	Subtract S from D. $D = D - S$. $C =$ borrow of $(D - S)$. *	2
SUBR	D, {#}S	{WC/WZ/WCZ}	Subtract D from S (reverse). $D = S - D$. $C =$ borrow of $(S - D)$. *	2
SUBS	D, {#}S	{WC/WZ/WCZ}	Subtract S from D, signed. $D = D - S$. $C =$ correct sign of $(D - S)$. *	2
SUBSX	D, {#}S	{WC/WZ/WCZ}	Subtract $(S + C)$ from D, signed and extended. $D = D - (S + C)$. $C =$ correct sign of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.	2
SUBX	D, {#}S	{WC/WZ/WCZ}	Subtract $(S + C)$ from D, extended. $D = D - (S + C)$. $C =$ borrow of $(D - (S + C))$. $Z = Z \text{ AND } (\text{result} == 0)$.	2
SUMC	D, {#}S	{WC/WZ/WCZ}	Sum $\div/-S$ into D by C. If $C = 1$ then $D = D - S$, else $D = D + S$. $C =$ correct sign of $(D \div/- S)$. *	2

SUMNC	D, {#}S	{WC/WZ/WCZ}	Sum +/-S into D by !C. If C = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *	2
SUMNZ	D, {#}S	{WC/WZ/WCZ}	Sum +/-S into D by !Z. If Z = 0 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *	2
SUMZ	D, {#}S	{WC/WZ/WCZ}	Sum +/-S into D by Z. If Z = 1 then D = D - S, else D = D + S. C = correct sign of (D +/- S). *	2
TEST	D	{WC/WZ/WCZ}	Test D. C = parity of D. Z = (D == 0).	2
TEST	D, {#}S	{WC/WZ/WCZ}	Test D with S. C = parity of (D & S). Z = ((D & S) == 0).	2
TESTB	D, {#}S	WC/WZ	Test bit S[4:0] of D, write to C/Z. C/Z = D[S[4:0]].	2
TESTB	D, {#}S	ORC/ORZ	Test bit S[4:0] of D, OR into C/Z. C/Z = C/Z OR D[S[4:0]].	2
TESTB	D, {#}S	ANDC/ANDZ	Test bit S[4:0] of D, AND into C/Z. C/Z = C/Z AND D[S[4:0]].	2
TESTB	D, {#}S	XORC/XORZ	Test bit S[4:0] of D, XOR into C/Z. C/Z = C/Z XOR D[S[4:0]].	2
TESTBN	D, {#}S	WC/WZ	Test bit S[4:0] of !D, write to C/Z. C/Z = !D[S[4:0]].	2
TESTBN	D, {#}S	ORC/ORZ	Test bit S[4:0] of !D, OR into C/Z. C/Z = C/Z OR !D[S[4:0]].	2
TESTBN	D, {#}S	ANDC/ANDZ	Test bit S[4:0] of !D, AND into C/Z. C/Z = C/Z AND !D[S[4:0]].	2
TESTBN	D, {#}S	XORC/XORZ	Test bit S[4:0] of !D, XOR into C/Z. C/Z = C/Z XOR !D[S[4:0]].	2
TESTN	D, {#}S	{WC/WZ/WCZ}	Test D with !S. C = parity of (D & !S). Z = ((D & !S) == 0).	2
WRC	D		Write 0 or 1 to D, according to C. D = {31'b0, C}.	2
WRNC	D		Write 0 or 1 to D, according to !C. D = {31'b0, !C}.	2
WRNZ	D		Write 0 or 1 to D, according to !Z. D = {31'b0, !Z}.	2
WRZ	D		Write 0 or 1 to D, according to Z. D = {31'b0, Z}.	2
XOR	D, {#}S	{WC/WZ/WCZ}	XOR S into D. D = D ^ S. C = parity of result. *	2
XOR032	D		Iterate D with xoroshiro32+ PRNG algorithm and put PRNG result into next instruction's S.	2
ZEROX	D, {#}S	{WC/WZ/WCZ}	Zero-extend D above bit S[4:0]. C = MSB of result. *	2

Pin & Smart Pin Instructions				
Instruction			Description	Clocks Cog, LUT & Hub
Pin				
DIRC	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRH	{#}D	{WCZ}	DIR hits of pins D[10:6]+D[5:0].D[5:0] = 1. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRL	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = 0. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRNC	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = !C. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRNOT	{#}D	{WCZ}	Toggle DIR bits of pins D[10:6]+D[5:0].D[5:0]. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRNZ	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = !Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRRND	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = RNDs. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DIRZ	{#}D	{WCZ}	DIR bits of pins D[10:6]+D[5:0].D[5:0] = Z. Wraps within DIRA/DIRB. Prior SETQ overrides D[10:6]. C,Z = DIR bit.	2
DRVCL	{#}D	{WCZ}	OUT bits of pins D[10:6]+D[5:0].D[5:0] = C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2
DRVH	{#}D	{WCZ}	OUT bits of pins D[10:6]+D[5:0].D[5:0] = 1. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2
DRVL	{#}D	{WCZ}	OUT bits of pins D[10:6]+D[5:0].D[5:0] = 0. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2
DRVNC	{#}D	{WCZ}	OUT bits of pins D[10:6]+D[5:0].D[5:0] = !C. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2
DRVNOT	{#}D	{WCZ}	Toggle OUT bits of pins D[10:6]+D[5:0].D[5:0]. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2
DRVNZ	{#}D	{WCZ}	OUT bits of pins D[10:6]+D[5:0].D[5:0] = !Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10:6]. C,Z = OUT bit.	2

DRVNRD	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = RNDs. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
DRVZ	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = Z. DIR bits = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTC	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTH	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = 1. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FTLTL	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = 0. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTNC	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = !C. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTNOT	{#}D	{WCZ}	Toggle OUT bits of pins D[10-6]+D[5-0].D[5-0]. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTNZ	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = !Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTRND	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = RNDs. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
FLTZ	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = Z. DIR bits = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTC	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTH	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = 1. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTL	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = 0. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTNC	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = !C. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTNOT	{#}D	{WCZ}	Toggle OUT bits of pins D[10-6]+D[5-0].D[5-0]. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z =	2

			OUT bit.	
OUTNZ	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = !Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTRND	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = RNDs. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
OUTZ	{#}D	{WCZ}	OUT bits of pins D[10-6]+D[5-0].D[5-0] = Z. Wraps within OUTA/OUTB. Prior SETQ overrides D[10-6]. C,Z = OUT bit.	2
TESTP	{#}D	WC/WZ	Test IN bit of pin D[5-0], write to C/Z. C/Z = IN[D[5-0]].	2
TESTP	{#}D	ORC/ORZ	Test IN bit of pin D[5-0], OR into C/Z. C/Z = C/Z OR IN[D[5-0]].	2
TESTP	{#}D	ANDC/ANDZ	Test IN bit of pin D[5-0], AND into C/Z. C/Z = C/Z AND IN[D[5-0]].	2
TESTP	{#}D	XORC/XORZ	Test IN bit of pin D[5-0], XOR into C/Z. C/Z = C/Z XOR IN[D[5-0]].	2
TESTPN	{#}D	WC/WZ	Test !IN bit of pin D[5-0], write to C/Z. C/Z = !IN[D[5-0]].	2
TESTPN	{#}D	ORC/ORZ	Test !IN bit of pin D[5-0], OR into C/Z. C/Z = C/Z OR !IN[D[5-0]].	2
TESTPN	{#}D	ANDC/ANDZ	Test !IN bit of pin D[5-0], AND into C/Z. C/Z = C/Z AND !IN[D[5-0]].	2
TESTPN	{#}D	XORC/XORZ	Test !IN bit of pin D[5-0], XOR into C/Z. C/Z = C/Z XOR !IN[D[5-0]].	2

Smart Pin			
AKPIN	{#}S	Acknowledge smart pins S[10:6]+S[5:0]..S[5:0]. Wraps within A/B pins. Prior SETQ overrides S[10:6].	2
GETSCP	D	Get four-channel oscilloscope samples into D. D = {ch3[7:0],ch2[7:0],ch1[7:0],ch0[7:0]}.	2
RDPIN	D, {#}S	{WC} Read smart pin S[5:0] result "Z" into D, acknowledge smart pin. C = modal result.	2
RQPIN	D, {#}S	{WC} Read smart pin S[5:0] result "Z" into D, don't acknowledge smart pin ("Q" in RQPIN means "quiet"). C = modal result.	2
SETDACS	{#}D	DAC3 = D[31:24], DAC2 = D[23:16], DAC1 = D[15:8], DAC0 = D[7:0].	2
SETSCP	{#}D	Set four-channel oscilloscope enable to D[6] and set input pin base to D[5:2].	2
WRPIN	{#}D, {#}S	Set mode of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].	2
WXPIN	{#}D, {#}S	Set "X" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].	2
WYPIN	{#}D, {#}S	Set "Y" of smart pins S[10:6]+S[5:0]..S[5:0] to D, acknowledge smart pins. Wraps within A/B pins. Prior SETQ overrides S[10:6].	2

Branch Instructions		
Instruction	Description	Clocks Cog & LUT / Hub
CALL #(\)A	Call to A by pushing {C, Z, 10'b0, PC[19:0]} onto stack. If R = 1 then PC += A, else PC = A. "\ " forces R = 0.	4 / 13...20
CALL D {WC/WZ/WCZ}	Call to D by pushing {C, Z, 10'b0, PC[19:0]} onto stack. C = D[31], Z = D[30], PC = D[19:0].	4 / 13...20
CALLA #(\)A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR A++. If R = 1 then PC += A, else PC = A. "\ " forces R = 0.	5...12 ¹ / 14...32 ¹
CALLA D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR A++. C = D[31], Z = D[30], PC = D[19:0].	5...12 ¹ / 14...32 ¹
CALLB #(\)A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR B++. If R = 1 then PC += A, else PC = A. "\ " forces R = 0.	5...12 ¹ / 14...32 ¹
CALLB D {WC/WZ/WCZ}	Call to D by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTR B++. C = D[31], Z = D[30], PC = D[19:0].	5...12 ¹ / 14...32 ¹
CALLD D, {#}S {WC/WZ/WCZ}	Call to S** by writing {C, Z, 10'b0, PC[19:0]} to D. C = S[31], Z = S[30].	4 / 13...20
CALLD PA/PB/PTRA/PTRB, #(\)A	Call to A by writing {C, Z, 10'b0, PC[19:0]} to PA/PB/PTRA/PTRB (per W). If R = 1 then PC += A, else PC = A. "\ " forces R = 0.	4 / 13...20
CALLPA {#}D, {#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PA.	4 / 13...20
CALLPB {#}D, {#}S	Call to S** by pushing {C, Z, 10'b0, PC[19:0]} onto stack, copy D to PB.	4 / 13...20
DJF D, {#}S	Decrement D and jump to S** if result is SFFFF_FFFF.	2 or 4 / 2 or 13...20
DJNF D, {#}S	Decrement D and jump to S** if result is not SFFFF_FFFF.	2 or 4 / 2 or 13...20
DJNZ D, {#}S	Decrement D and jump to S** if result is not zero.	2 or 4 / 2 or 13...20
DJZ D, {#}S	Decrement D and jump to S** if result is zero.	2 or 4 / 2 or 13...20
EXECF {#}D	Jump to D[9:0] in cog/LUT and set SKIPF pattern to D[31:10]. PC = {10'b0, D[9:0]}.	4 / 4

IJNZ	D, {#}S	Increment D and jump to S** if result is not zero.	2 or 4 / 2 or 13...20
IJZ	D, {#}S	Increment D and jump to S** if result is zero.	2 or 4 / 2 or 13...20
JMP	#(\)A	Jump to A. If R = 1 then PC += A, else PC = A. "v" forces R = 0.	4 / 13...20
JMP	D {WC/WZ/WCZ}	Jump to D. C = D[31], Z = D[30], PC = D[19:0].	4 / 13...20
JMPREL	{#}D	Jump ahead/back by D instructions. For cogex, PC += D[19:0]. For hubex, PC += D[17:0] << 2.	4 / 13...20
REP	{#}D, {#}S	Execute next D[8:0] instructions S times. If S = 0, repeat instructions infinitely. If D[8:0] = 0, nothing repeats.	2 / 2
RESI0		Resume from INTO. (CALLD \$1FF,\$1FF WCZ)	4 / 13...20
RESI1		Resume from INT1. (CALLD \$1F4,\$1F5 WCZ)	4 / 13...20
RESI2		Resume from INT2. (CALLD \$1F2,\$1F3 WCZ)	4 / 13...20
RESI3		Resume from INT3. (CALLD \$1F0,\$1F1 WCZ)	4 / 13...20
RET	{WC/WZ/WCZ}	Return by popping stack (K). C = K[31], Z = K[30], PC = K[19:0].	4 / 13...20
RETA	{WC/WZ/WCZ}	Return by reading hub long (L) at --PTRA. C = L[31], Z = L[30], PC = L[19:0].	11...18 ¹ / 20...40 ¹
RETB	{WC/WZ/WCZ}	Return by reading hub long (L) at --PTRB. C = L[31], Z = L[30], PC = L[19:0].	11...18 ¹ / 20...40 ¹
RETI0		Return from INTO. (CALLD \$1FF,\$1FF WCZ)	4 / 13...20
RETI1		Return from INT1. (CALLD \$1FF,\$1F5 WCZ)	4 / 13...20
RETI2		Return from INT2. (CALLD \$1FF,\$1F3 WCZ)	4 / 13...20
RETI3		Return from INT3. (CALLD \$1FF,\$1F1 WCZ)	4 / 13...20
SKIP	{#}D	Skip instructions per D. Subsequent instructions 0..31 get cancelled for each '1' bit in D[0]..D[31].	2 / 2
SKIPF	{#}D	Skip cog/LUT instructions fast per D. Like SKIP, but instead of cancelling instructions, the PC leaps over them.	2 / ILLEGAL
TJF	D, {#}S	Test D and jump to S** if D is full (D = \$FFFF_FFFF).	2 or 4 / 2 or 13...20
TJNF	D, {#}S	Test D and jump to S** if D is not full (D != \$FFFF_FFFF).	2 or 4 / 2 or 13...20
TJNS	D, {#}S	Test D and jump to S** if D is not signed (D[31] = 0).	2 or 4 / 2 or 13...20
TJNZ	D, {#}S	Test D and jump to S** if D is not zero.	2 or 4 / 2 or 13...20
TJS	D, {#}S	Test D and jump to S** if D is signed (D[31] = 1).	2 or 4 / 2 or 13...20
TJV	D, {#}S	Test D and jump to S** if D overflowed (D[31] != C, C = 'correct sign' from last addition/subtraction).	2 or 4 / 2 or 13...20
TJZ	D, {#}S	Test D and jump to S** if D is zero.	2 or 4 / 2 or 13...20

Hub Control, FIFO, & RAM Instructions			
Instruction		Description	Clocks Cog & LUT / Hub
Hub Control			
COGID	{#}D {WC}	If D is register and no WC, get cog ID (0 to 15) into D. If WC, check status of cog D[3:0]. C = 1 if on.	2...9, +2 if result / same
COGINIT	{#}D, {#}S {WC}	Start cog selected by D. S[19:0] sets hub startup address and PTRB of cog. Prior SETQ sets PTRB of cog.	2...9, +2 if result / same
COGSTOP	{#}D	Stop cog D[3:0].	2...9 / same
LOCKNEW	D {WC}	Request a LOCK. D will be written with the LOCK number (0 to 15). C = 1 if no LOCK available.	4...11 / same
LOCKREL	{#}D {WC}	Release LOCK D[3:0]. If D is a register and WC, get current/last cog id of LOCK owner into D and LOCK status into C.	2...9, +2 if result / same
LOCKRET	{#}D	Return LOCK D[3:0] for reallocation.	2...9 / same
LOCKTRY	{#}D {WC}	Try to get LOCK D[3:0]. C = 1 if got LOCK. LOCKREL releases LOCK. LOCK is also released if owner cog stops or restarts.	2...9, +2 if result / same
HUBSET	{#}D	Set hub configuration to D.	2...9 / same

Hub FIFO			
GETPTR	D	Get current FIFO hub pointer into D.	2 / FIFO IN USE
FBLOCK	{#}D, {#}S	Set next block for when block wraps. D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.	2 / FIFO IN USE
RDFAST	{#}D, {#}S	Begin new fast hub read via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.	2 or WRFAST finish + 10...17 / FIFO IN USE
WRFAST	{#}D, {#}S	Begin new fast hub write via FIFO. D[31] = no wait, D[13:0] = block size in 64-byte units (0 = max), S[19:0] = block start address.	2 or WRFAST finish + 3 / FIFO IN USE
RFBYTE	D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended byte from FIFO into D. C = MSB of byte. *	2 / FIFO IN USE
RFLONG	D {WC/WZ/WCZ}	Used after RDFAST. Read long from FIFO into D. C = MSB of long. *	2 / FIFO IN USE
RFVAR	D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended 1.4-byte value from FIFO into D. C = 0. *	2 / FIFO IN USE
RFVARS	D {WC/WZ/WCZ}	Used after RDFAST. Read sign-extended 1.4-byte value from FIFO into D. C = MSB of value. *	2 / FIFO IN USE
RWORD	D {WC/WZ/WCZ}	Used after RDFAST. Read zero-extended word from FIFO into D. C = MSB of word. *	2 / FIFO IN USE
WFBYTE	{#}D	Used after WRFAST. Write byte in D[7:0] into FIFO.	2 / FIFO IN USE
WFLONG	{#}D	Used after WRFAST. Write long in D[31:0] into FIFO.	2 / FIFO IN USE
WWORD	{#}D	Used after WRFAST. Write word in D[15:0] into FIFO.	2 / FIFO IN USE
Hub RAM			
POPA	D {WC/WZ/WCZ}	Read long from hub address --PTRA into D. C = MSB of long. *	9...16 ¹ / 9...26 ¹
POPB	D {WC/WZ/WCZ}	Read long from hub address --PTRB into D. C = MSB of long. *	9...16 ¹ / 9...26 ¹
RDBYTE	D, {#}S/P {WC/WZ/WCZ}	Read zero-extended byte from hub address {#}S/PTRx into D. C = MSB of byte. *	9...16 / 9...26
RDLONG	D, {#}S/P {WC/WZ/WCZ}	Read long from hub address {#}S/PTRx into D. C = MSB of long. * Prior SETQ/SETQ2 invokes cog/LUT block transfer.	9...16 ¹ / 9...26 ¹
RDWORD	D, {#}S/P {WC/WZ/WCZ}	Read zero-extended word from hub address {#}S/PTRx into D. C = MSB of word. *	9...16 ¹ / 9...26 ¹
PUSHA	{#}D	Write long in D[31:0] to hub address PTRA++.	3...10 ¹ / 3...20 ¹
PUSHB	{#}D	Write long in D[31:0] to hub address PTRB++.	3...10 ¹ / 3...20 ¹
WMLONG	D, {#}S/P	Write only non-S00 bytes in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.	3...10 ¹ / 3...20 ¹
WRBYTE	{#}D, {#}S/P	Write byte in D[7:0] to hub address {#}S/PTRx.	3...10 / 3...20
WRLONG	{#}D, {#}S/P	Write long in D[31:0] to hub address {#}S/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.	3...10 ¹ / 3...20 ¹
WRWORD	{#}D, {#}S/P	Write word in D[15:0] to hub address {#}S/PTRx.	3...10 ¹ / 3...20 ¹

¹ +1 if crosses hub long

Event Instructions		
Instruction	Description	Clocks Cog & LUT / Hub
ADDCT1 D, {#}S	Set CT1 event to trigger on $CT = D + S$. Adds S into D.	2
ADDCT2 D, {#}S	Set CT2 event to trigger on $CT = D + S$. Adds S into D.	2
ADDCT3 D, {#}S	Set CT3 event to trigger on $CT = D + S$. Adds S into D.	2
COGATN {#}D	Strobe "attention" of all cogs whose corresponding bits are high in D[15:0].	2
JATN {#}S	Jump to S** if ATN event flag is set.	2 or 4 / 2 or 13...20
JCT1 {#}S	Jump to S** if CT1 event flag is set.	2 or 4 / 2 or 13...20
JCT2 {#}S	Jump to S** if CT2 event flag is set.	2 or 4 / 2 or 13...20
JCT3 {#}S	Jump to S** if CT3 event flag is set.	2 or 4 / 2 or 13...20
JFBW {#}S	Jump to S** if FBW event flag is set.	2 or 4 / 2 or 13...20
JINT {#}S	Jump to S** if INT event flag is set.	2 or 4 / 2 or 13...20
JNATN {#}S	Jump to S** if ATN event flag is clear.	2 or 4 / 2 or 13...20
JNCT1 {#}S	Jump to S** if CT1 event flag is clear.	2 or 4 / 2 or 13...20
JNCT2 {#}S	Jump to S** if CT2 event flag is clear.	2 or 4 / 2 or 13...20
JNCT3 {#}S	Jump to S** if CT3 event flag is clear.	2 or 4 / 2 or 13...20
JNFBW {#}S	Jump to S** if FBW event flag is clear.	2 or 4 / 2 or 13...20
JNINT {#}S	Jump to S** if INT event flag is clear.	2 or 4 / 2 or 13...20
JNPAT {#}S	Jump to S** if PAT event flag is clear.	2 or 4 / 2 or 13...20
JNQMT {#}S	Jump to S** if QMT event flag is clear.	2 or 4 / 2 or 13...20
JNSE1 {#}S	Jump to S** if SE1 event flag is clear.	2 or 4 / 2 or 13...20
JNSE2 {#}S	Jump to S** if SE2 event flag is clear.	2 or 4 / 2 or 13...20
JNSE3 {#}S	Jump to S** if SE3 event flag is clear.	2 or 4 / 2 or 13...20
JNSE4 {#}S	Jump to S** if SE4 event flag is clear.	2 or 4 / 2 or 13...20
JNXFI {#}S	Jump to S** if XF1 event flag is clear.	2 or 4 / 2 or 13...20
JNXMT {#}S	Jump to S** if XMT event flag is clear.	2 or 4 / 2 or 13...20

JNXRL	{#}S	Jump to S** if XRL event flag is clear.	2 or 4 / 2 or 13...20
JNXRO	{#}S	Jump to S** if XRO event flag is clear.	2 or 4 / 2 or 13...20
JPAT	{#}S	Jump to S** if PAT event flag is set.	2 or 4 / 2 or 13...20
JQMT	{#}S	Jump to S** if QMT event flag is set.	2 or 4 / 2 or 13...20
JSE1	{#}S	Jump to S** if SE1 event flag is set.	2 or 4 / 2 or 13...20
JSE2	{#}S	Jump to S** if SE2 event flag is set.	2 or 4 / 2 or 13...20
JSE3	{#}S	Jump to S** if SE3 event flag is set.	2 or 4 / 2 or 13...20
JSE4	{#}S	Jump to S** if SE4 event flag is set.	2 or 4 / 2 or 13...20
JXFI	{#}S	Jump to S** if XFI event flag is set.	2 or 4 / 2 or 13...20
JXMT	{#}S	Jump to S** if XMT event flag is set.	2 or 4 / 2 or 13...20
JXRL	{#}S	Jump to S** if XRL event flag is set.	2 or 4 / 2 or 13...20
JXRO	{#}S	Jump to S** if XRO event flag is set.	2 or 4 / 2 or 13...20
POLLATN	{WC/WZ/WCZ}	Get ATN event flag into C/Z, then clear it.	2
POLLCT1	{WC/WZ/WCZ}	Get CT1 event flag into C/Z, then clear it.	2
POLLCT2	{WC/WZ/WCZ}	Get CT2 event flag into C/Z, then clear it.	2
POLLCT3	{WC/WZ/WCZ}	Get CT3 event flag into C/Z, then clear it.	2
POLLFBW	{WC/WZ/WCZ}	Get FBW event flag into C/Z, then clear it.	2
POLLINT	{WC/WZ/WCZ}	Get INT event flag into C/Z, then clear it.	2
POLLPAT	{WC/WZ/WCZ}	Get PAT event flag into C/Z, then clear it.	2
POLLQMT	{WC/WZ/WCZ}	Get QMT event flag into C/Z, then clear it.	2
POLLSE1	{WC/WZ/WCZ}	Get SE1 event flag into C/Z, then clear it.	2
POLLSE2	{WC/WZ/WCZ}	Get SE2 event flag into C/Z, then clear it.	2
POLLSE3	{WC/WZ/WCZ}	Get SE3 event flag into C/Z, then clear it.	2
POLLSE4	{WC/WZ/WCZ}	Get SE4 event flag into C/Z, then clear it.	2
POLLXFI	{WC/WZ/WCZ}	Get XFI event flag into C/Z, then clear it.	2
POLLXMT	{WC/WZ/WCZ}	Get XMT event flag into C/Z, then clear it.	2
POLLXRL	{WC/WZ/WCZ}	Get XRL event flag into C/Z, then clear it.	2
POLLXRO	{WC/WZ/WCZ}	Get XRO event flag into C/Z, then clear it.	2
SETPAT	{#}D, {#}S	Set pin pattern for PAT event. C selects INA/INB, Z selects =/!=, D provides mask value, S provides match value.	2

SETPAT	{#}D, {#}S	Set pin pattern for PAT event. C selects INA/INB, Z selects =/!=, D provides mask value, S provides match value.	2
SETSE1	{#}D	Set SE1 event configuration to D[8:0].	2
SETSE2	{#}D	Set SE2 event configuration to D[8:0].	2
SETSE3	{#}D	Set SE3 event configuration to D[8:0].	2
SETSE4	{#}D	Set SE4 event configuration to D[8:0].	2
WAITATN	{WC/WZ/WCZ}	Wait for ATN event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITCT1	{WC/WZ/WCZ}	Wait for CT1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITCT2	{WC/WZ/WCZ}	Wait for CT2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITCT3	{WC/WZ/WCZ}	Wait for CT3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITFBW	{WC/WZ/WCZ}	Wait for FBW event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITINT	{WC/WZ/WCZ}	Wait for INT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITPAT	{WC/WZ/WCZ}	Wait for PAT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITSE1	{WC/WZ/WCZ}	Wait for SE1 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITSE2	{WC/WZ/WCZ}	Wait for SE2 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITSE3	{WC/WZ/WCZ}	Wait for SE3 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITSE4	{WC/WZ/WCZ}	Wait for SE4 event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITXFI	{WC/WZ/WCZ}	Wait for XFI event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITXMT	{WC/WZ/WCZ}	Wait for XMT event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITXRL	{WC/WZ/WCZ}	Wait for XRL event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+
WAITXRO	{WC/WZ/WCZ}	Wait for XRO event flag, then clear it. Prior SETQ sets optional CT timeout value. C/Z = timeout.	2+

Interrupt Instructions		
Instruction	Description	Clocks Cog, LUT & Hub
ALLOWI	Allow interrupts (default).	2
BRK {#}D	If in debug ISR, set next break condition to D. Else, set BRK code to D[7:0] and unconditionally trigger BRK interrupt, if enabled.	2
COGBRK {#}D	If in debug ISR, trigger asynchronous breakpoint in cog D[3:0]. Cog D[3:0] must have asynchronous breakpoint enabled.	2
GETBRK D WC/WZ/WCZ	Get breakpoint/cog status into D according to WC/WZ/WCZ. See documentation for details.	2
NIXINT1	Cancel INT1.	2
NIXINT2	Cancel INT2.	2
NIXINT3	Cancel INT3.	2
SETINT1 {#}D	Set INT1 source to D[3:0].	2
SETINT2 {#}D	Set INT2 source to D[3:0].	2
SETINT3 {#}D	Set INT3 source to D[3:0].	2
STALLI	Stall Interrupts.	2
TRGINT1	Trigger INT1, regardless of STALLI mode.	2
TRGINT2	Trigger INT2, regardless of STALLI mode.	2
TRGINT3	Trigger INT3, regardless of STALLI mode.	2

Register Indirection Instructions		
Instruction	Description	Clocks Cog & LUT / Hub
ALTB D, {#}S	Alter D field of next instruction to D[13:5].	2
ALTB D, {#}S	Alter D field of next instruction to (D[13:5] + S) & S1FF. D += sign-extended S[17:9].	2
ALTD D	Alter D field of next instruction to D[8:0].	2
ALTD D, {#}S	Alter D field of next instruction to (D + S) & S1FF. D += sign-extended S[17:9].	2
ALTGB D	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = D[10:2], N field = D[1:0].	2
ALTGB D, {#}S	Alter subsequent GETBYTE/ROLBYTE instruction. Next S field = (D[10:2] + S) & S1FF, N field = D[1:0]. D += sign-extended S[17:9].	2
ALTGN D	Alter subsequent GETNIB/ROLNIB instruction. Next S field = D[11:3], N field = D[2:0].	2
ALTGN D, {#}S	Alter subsequent GETNIB/ROLNIB instruction. Next S field = (D[11:3] + S) & S1FF, N field = D[2:0]. D += sign-extended S[17:9].	2
ALTGW D	Alter subsequent GETWORD/ROLWORD instruction. Next S field = D[9:1], N field = D[0].	2
ALTGW D, {#}S	Alter subsequent GETWORD/ROLWORD instruction. Next S field = ((D[9:1] + S) & S1FF), N field = D[0]. D += sign-extended S[17:9].	2
ALTI D	Execute D in place of next instruction. D stays same.	2
ALTI D, {#}S	Substitute next instruction's I/R/D/S fields with fields from D, per S. Modify D per S.	2
ALTR D	Alter result register address (normally D field) of next instruction to D[8:0].	2
ALTR D, {#}S	Alter result register address (normally D field) of next instruction to (D + S) & S1FF. D += sign-extended S[17:9].	2
ALTS D	Alter S field of next instruction to D[8:0].	2
ALTS D, {#}S	Alter S field of next instruction to (D + S) & S1FF. D += sign-extended S[17:9].	2
ALTSB D	Alter subsequent SETBYTE instruction. Next D field = D[10:2], N field = D[1:0].	2
ALTSB D, {#}S	Alter subsequent SETBYTE instruction. Next D field = (D[10:2] + S) & S1FF, N field = D[1:0]. D += sign-extended S[17:9].	2
ALTSN D	Alter subsequent SETNIB instruction. Next D field = D[11:3], N field = D[2:0].	2
ALTSN D, {#}S	Alter subsequent SETNIB instruction. Next D field = (D[11:3] + S) & S1FF, N field = D[2:0]. D += sign-extended S[17:9].	2
ALTSW D	Alter subsequent SETWORD instruction. Next D field = D[9:1], N field = D[0].	2
ALTSW D, {#}S	Alter subsequent SETWORD instruction. Next D field = (D[9:1] + S) & S1FF, N field = D[0]. D += sign-extended S[17:9].	2

CORDIC Solver Instructions		
Instruction	Description	Clocks Cog, LUT & Hub
GETQX D {WC/WZ/WCZ}	Retrieve CORDIC result X into D. Waits, in case result not ready. C = X[31]. ¹	2...58
GETQY D {WC/WZ/WCZ}	Retrieve CORDIC result Y into D. Waits, in case result not ready. C = Y[31]. ¹	2...58
QDIV {#}D, {#}S	Begin CORDIC unsigned division of (SETQ value or 32'b0, D) / S. GETQX/GETQY retrieves quotient/remainder.	2...9
QEXP {#}D	Begin CORDIC logarithm-to-number conversion of D. GETQX retrieves number.	2...9
QFRAC {#}D, {#}S	Begin CORDIC unsigned division of (D, SETQ value or 32'b0) / S. GETQX/GETQY retrieves quotient/remainder.	2...9
QLOG {#}D	Begin CORDIC number-to-logarithm conversion of D. GETQX retrieves log (5 ^{whole_exponent} , 2 ^{fractional_exponent}).	2...9
QMUL {#}D, {#}S	Begin CORDIC unsigned multiplication of D * S. GETQX/GETQY retrieves lower/upper product.	2...9
QROTATE {#}D, {#}S	Begin CORDIC rotation of point (D, SETQ value or 32'b0) by angle S. GETQX/GETQY retrieves X/Y.	2...9
QSQRT {#}D, {#}S	Begin CORDIC square root of (S, D). GETQX retrieves root.	2...9
QVECTOR {#}D, {#}S	Begin CORDIC vectoring of point (D, S). GETQX/GETQY retrieves length/angle.	2...9

¹ Z = (result == 0)

Color Space Converter and Pixel Mixer Instructions		
Instruction	Description	Clocks Cog, LUT & Hub
Color Space Converter		
SETCFRQ {#}D	Set the colorspace converter "CFRQ" parameter to D[31:0].	2
SETCI {#}D	Set the colorspace converter "CI" parameter to D[31:0].	2
SETCMOD {#}D	Set the colorspace converter "CMOD" parameter to D[8:0].	2
SETCQ {#}D	Set the colorspace converter "CQ" parameter to D[31:0].	2
SETCY {#}D	Set the colorspace converter "CY" parameter to D[31:0].	2
Pixel Mixer		
ADDPPIX D, {#}S	Add bytes of S into bytes of D, with SFF saturation.	7
BLNPIX D, {#}S	Alpha-blend bytes of S into bytes of D, using SETPIV value.	7
MIXPIX D, {#}S	Mix bytes of S into bytes of D, using SETPIX and SETPIV values.	7
MULPIX D, {#}S	Multiply bytes of S into bytes of D, where SFF = 1.0 and S00 = 0.0.	7
SETPIV {#}D	Set BLNPIX/MIXPIX blend factor to D[7:0].	2
SETPIX {#}D	Set MIXPIX mode to D[5:0].	2

Lookup Table, Streamer, and Misc Instructions		
Instruction	Description	Clocks Cog & LUT / Hub
Lookup Table		
RDLUT D, {#}S/P {WC/WZ/WCZ}	Read data from LUT address {#}S/PTRx into D. C = MSB of data. *	3
SETLUTS {#}D	If D[0] = 1 then enable LUT sharing, where LUT writes within the adjacent odd/even companion cog are copied to this cog's LUT.	2
WRLUT {#}D, {#}S/P	Write D to LUT address {#}S/PTRx.	2
Streamer		
GETXACC D	Get the streamer's Goertzel X accumulator into D and the Y accumulator into the next instruction's S, clear accumulators.	2
SETXFRQ {#}D	Set streamer NCO frequency to D.	2
XCONT {#}D, {#}S	Buffer new streamer command to be issued on final NCO rollover of current command, continuing phase.	2+
XINIT {#}D, {#}S	Issue streamer command immediately, zeroing phase.	2
XSTOP	Stop streamer immediately.	2
XZERO {#}D, {#}S	Buffer new streamer command to be issued on final NCO rollover of current command, zeroing phase.	2+

Miscellaneous			
AUGD	#n	Queue #n to be used as upper 23 bits for next #D occurrence, so that the next 9-bit #D will be augmented to 32 bits.	2
AUGS	#n	Queue #n to be used as upper 23 bits for next #S occurrence, so that the next 9-bit #S will be augmented to 32 bits.	2
GETCT	D {WC}	Get CT[31:0] or CT[63:32] if WC into D. GETCT WC + GETCT gets full CT. CT=0 on reset, CT++ on every clock. C = same.	2
GETRND	WC/WZ/WCZ	Get RND into C/Z. C = RND[31], Z = RND[30], unique per cog.	2
GETRND	D {WC/WZ/WCZ}	Get RND into D/C/Z. RND is the PRNG that updates on every clock. D = RND[31:0], C = RND[31], Z = RND[30], unique per cog.	2
NOP		No operation.	2
POP	D {WC/WZ/WCZ}	Pop stack (K). D = K. C = K[31]. *	2
PUSH	{#}D	Push D onto stack.	2
SETQ	{#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set block transfer. Also used before MUXQ/COGINIT/QDIV/QFRAC/QROTATE/WAITxxx.	2
SETQ2	{#}D	Set Q to D. Use before RDLONG/WRLONG/WMLONG to set LUT block transfer.	2
WAITX	{#}D {WC/WZ/WCZ}	Wait 2 + D clocks if no WC/WZ/WCZ. If WC/WZ/WCZ, wait 2 + (D & RND) clocks. C/Z = 0.	2 + D

Reference Material

Parallax Documentation

Propeller 2 Silicon Documentation	125 pages
Propeller 2 Hardware Manual	74 pages
Propeller 2 PASM Instructions	spread sheet
Spin 2 Language Documentation(Draft)	47 pages
Propeller 2 Smart Pin Supplementary Documentation	54 pages
Spin 2 Send Command Short Tutorial(JonTitus)	2 pages
Parallax Propeller 2 (P2X*C4M64P) Data Sheet (Updated 2021\07\09)	51 pages
Propeller 2 Questions and Answers	spread sheet
Propeller 2 Rev A Documentation (V32-09/2018)	81 pages
Rev A PASM Instructions	spread sheet

Miscellaneous Info (Doesn't Fit anywhere)

- 1) When using Forum chat line plae ~~~ then code followed by~~~ this keeps code indentation

Index

A

ABSD, 397
 ABSDS, 395
 ADC Analog Digital, 54
 ADD, 270
 ADDPIX D,{#}S, 561
 Address Convention, 689
 ADDS, 273
 ADDSX, 275
 ADDX, 272
 AKPIN {#}S, 585
 ALLOWI, 610
 ALTB D, 529
 ALTB D,{#}S, 526
 ALTD D, 518
 ALTD D,{#}S, 516
 ALTGB D, 505
 ALTGB D,{#}S, 505
 ALTGN D, 503
 ALTGN D,{#}S, 503
 ALTGW D, 509
 ALTGW D,{#}S, 509
 ALTI D, 533
 ALTI D,{#}S, 533
 ALTR D, 514
 ALTR D,{#}S, 511
 ALTS D, 523
 ALTS D,{#}S, 521
 ALTSB D, 504
 ALTSB D,{#}S, 504
 ALTSN D, 502
 ALTSN D,{#}S, 502
 ALTSW D, 508
 ALTSW D,{#}S, 508
 Analog Input Smart Pin, 54
 Analog Out Smart Pin, 48
 AnalogIn and AnalogOut, 60
 AND, 371
 ANDN, 373

B

BITC, 357
 BITH, 354
 BITL, 349
 BITNC, 360
 BITNOT, 369
 BITNZ, 365
 BITRND, 367

BITZ, 363
 BMASK D, 542
 BMASK D,{#}S, 540
 Byte Declaration, 684

C

CALLD D,{#}S, 574
 CALLPA, 577
 CALLPB, 577
 CMP, 287
 CMPM, 301
 CMPR, 299
 CMPS, 293
 CMPSUB, 303
 CMPSX, 296
 CMPX, 290
 COG HUB Access, 732
 COG Overview, 76
 COGID {#}D {WC}, 598
 COGINIT {#}D,{#}S {WC}, 595
 COGSTOP {#}D, 598
 CON Block, 158
 Cordic Solver, 745
 CRC8 Cycle Redundancy Check, 699
 CRCBIT D,{#}S, 544
 CRCNIB D,{#}S, 546

D

Dat Block, 212
 Debug for Testing, 112
 DECMOD, 422
 DECOD D, 539
 DECOD D,{#}S, 537
 Digital Pin Operation, 26
 DIR Direction, 628
 DJF D,{#}S, 578
 DJNF D,{#}S, 578
 DJNZ D,{#}S, 578
 DIJZ D, 578

E

ENCODD, 430
 ENCODDS, 428

F

FBLOCK {#}D,{#}S, 590

FGE, 305
 FGES, 311
 FLE, 309
 FLES, 313

G

GETBYTE D, 473
 GETBYTE D,{#}S,#N, 471, 475
 GETCT D {WC}, 601
 GETNIB D, 452, 454
 GETNIB D,{#}S,#N, 451
 GETQX D {WC/WZ/WCZ}, 601
 GETQY D {WC/WZ/WCZ}, 601
 GETRND WC/WZ/WCZ, 601
 GETRND D {WC/WZ/WCZ}, 601
 GETWORD D, 491
 GETWORD D,{#}S,#N, 489, 493
 GETXACC D, 603

H

Hub RAM, 731
 HUB RAM Slice, 733
 HUBSET {#}D, 597

I

IJNZ D,{#}S, 579
 IJZ D,{#}S, 579
 INCMOD, 420
 Interrupt Jump Instructions, 662

J

JATN {#}S, 581
 JCT1 {#}S, 581
 JCT2 {#}S, 581
 JCT3 {#}S, 581
 JFBW {#}S, 581
 JINT {#}S, 581
 JNATN {#}S, 582
 JNCT1 {#}S, 581
 JNCT2 {#}S, 581
 JNCT3 {#}S, 582
 JNFBW {#}S, 582
 JNINT {#}S, 581
 JNPAT {#}S, 582
 JNQMT {#}S, 582
 JNSE1 {#}S, 582
 JNSE2 {#}S, 582
 JNSE3 {#}S, 582
 JNSE4 {#}S, 582
 JNXFI {#}S, 582

JNXMT {#}S, 582
 JNXRL {#}S, 582
 JNXRO {#}S, 582
 JPAT {#}S, 581
 JQMT {#}S, 581
 JSE1 {#}S, 581
 JSE2 {#}S, 581
 JSE3 {#}S, 581
 JSE4 {#}S, 581
 JXFI {#}S, 581
 JXMT {#}S, 581
 JXRL {#}S, 581
 JXRO {#}S, 581

L

LOCKNEW D {WC}, 598
 LOCKREL {#}D {WC}, 598
 LOCKRET {#}D, 598
 LOCKTRY {#}D {WC}, 598
 Long Declaration, 688

M

Method Pointer, 227
 MOVBYTES D,{#}S, 553
 MUL D,{#}S {WZ}, 555
 MULPIX D,{#}S, 562
 MULS D,{#}S {WZ}, 557
 MUXC, 379
 MUXNC, 383
 MUXNIBS D,{#}S, 550
 MUXNITS D,{#}S, 548
 MUXNZ, 387
 MUXQ D,{#}S, 551
 MUXZ, 385

N

NEGCD, 406
 NEGCDs, 404
 NEGd, 402
 NEGDS, 399
 NEGNCd, 410
 NEGNCDS, 408
 NEGNZ, 418
 NEGNZDS, 416
 NEGZD, 414
 NEGZDS, 412
 NIXINT1, 610
 NIXINT2, 610
 NIXINT3, 610
 NOP No operation, 248
 NOTD, 393

NOTDS, 391

O

OBJ Block, 162

Operators, 219

OR, 375

P

PASM Propeller Assembly, 237

POLLCT1/WAITCT1 event CT1, 609

POLLCT1/WAITCT1 Get CT1, 607

POLLCT2/WAITCT2 event CT2, 609

POLLCT2/WAITCT2 Get CT2, 607

POLLCT3/WAITCT3 event CT3, 609

POLLCT3/WAITCT3 Get CT3, 607

POLLINT {WC/WZ/WCZ}, 607

POPA D {WC/WZ/WCZ}, 573

POPB D {WC/WZ/WCZ}, 573

PRI Block, 211

Program Structure, 148

Propeller Tool, 148

PUB Block, 195

PWM Pulse Width Modulation, 45

Q

QDIV {#}D,{#}S, 596

QEXP {#}D, 599

QFRAC {#}D,{#}S, 596

QLOG {#}D, 599

QMUL {#}D,{#}S, 596

QROTATE {#}D,{#}S, 596

QSQRT {#}D,{#}S, 596

QVECTOR {#}D,{#}S, 596

R

RCL Rotate Carry Left, 260

RCR Rotate Carry Right, 260

RDBYTE D,{#}S/P {WC/WZ/WCZ}, 571

RDFAST {#}D,{#}S, 589

RDLONG D,{#}S/P {WC/WZ/WCZ}, 571

RDLUT D,{#}S/P {WC/WZ/WCZ}, 571

RDpin D,{#}S {WC}, 571

RDWORD D,{#}S/P {WC/WZ/WCZ}, 571

RECV, 235

REP {#}D,{#}S, 592

RESIO, 575

RESI1, 575

RESI2, 575

RESI3, 575

RETI0, 576

RETI1, 576

RETI2, 576

RETI3, 576

RFBYTE D {WC/WZ/WCZ}, 600

RFLONG D {WC/WZ/WCZ}, 600

RFVAR D {WC/WZ/WCZ}, 600

RFVARS D {WC/WZ/WCZ}, 600

RWORD D {WC/WZ/WCZ}, 600

ROL Rotate Left, 249

ROLBYTE D, 479

ROLBYTE D,{#}S,#N, 477, 479

ROLNIBD, 459

ROLNIBDSN, 457, 461

ROLWORD D, 498

ROLWORD D,{#}S,#N, 496, 500

ROR Rotate Right, 249

RQPIN D,{#}S {WC}, 571

S

SAR Shift Arithmetic Right, 264

SCA D,{#}S {WZ}, 558

SCAS D,{#}S {WZ}, 560

SEND, 231

SETBYTE {#}S, 467

SETBYTE D,{#}S,#N, 465, 469

SETD D, S/#, 534

SETDACS {#}, 602

SETINT1, 610

SETINT2, 610

SETINT3, 610

SETNIB, 446

SETNIB D,{#}S,#N, 448

SETNIBDS, 444

SETPAT {#}D,{#}S, 584

SETQ {#}D, 611

SETQ2 {#}D, 611

SETR D, S/#, 534

SETS D, S/#, 534

SETSE1 {#}D, 605

SETSE2 {#}D, 605

SETSE3 {#}D, 605

SETSE4 {#}D, 605

SETWORD {#}S, 484

SETWORD D,{#}S,#N, 482

SETWORD D,{S},#N, 486

SETXFRQ {#}D, 602

Shift Arithmetic Left, 264

SHL Shift Left, 255

SHR Shift Right, 255

SIGNX, 426

Smart Pin Block Diagram, 3

SPIN I/O Digital Methods, 41

STALLI, 610

SUB, 277
 SUBR, 285
 SUBS, 281
 SUBSX, 283
 SUBX, 279
 SUMC, 315
 SUMNC, 318
 SUMNZ, 322
 SUMZ, 320

T

TESTB, 324
 TESTB_ANDC/ANDZ, 331
 TESTB_ORC_ORZ, 337
 TESTB_XORC_XORZ, 343
 TESTBN, 328
 TESTBN_ANDC/ANDZ, 334
 TESTBN_ORC_ORZ, 340
 TESTBN_XORC_XORZ, 346
 TESTD, 438
 TESTDS, 436
 TESTND, 440
 TJF D,{#}S, 580
 TJNF D,{#}S, 580
 TJNS D,{#}S, 580
 TJNZ D,{#}S, 580
 TJS D,{#}S, 580
 TJV D,{#}S, 580
 TJZ D,{#}S, 580
 TRGINT1, 610
 TRGINT2, 610
 TRGINT3, 610

V

VAR Block, 190

W

WAITINT {WC/WZ/WCZ}, 609
 WAITX {#}D {WC/WZ/WCZ}, 604
 WFBYTE {#}D, 600
 WFLONG {#}D, 600
 WFWORD {#}D, 600
 Word Declaration, 687
 WRBYTE {#}D,{#}S, 587
 WRFast {#}D,{#}S, 589
 WRLONG {#}D,{#}S, 587
 WRLUT {#}D,{#}S, 587
 WRPIN {#}D,{#}S, 585
 WRWORD {#}D,{#}S/, 587
 WXPIN {#}D,{#}S, 585
 WYPIN {#}D,{#}S, 585

X

XCONT {#}D,{#}S, 588, 591
 XINIT {#}D,{#}S, 591
 XOR, 377
 XSTOP, 591
 XZERO {#}D,{#}S, 588, 591

Z

ZEROX, 424