Early Propeller Chip Documentation
Rev 0.1

Documentation

Hardware Description
- Main Memory
- HUB
  - Bus Access
  - COG Control
  - LOCK Control
  - CLK Control
    - Internal RC
    - External XTAL
    - PLL
- COG
  - Register Map
  - Peripheral Detail
  - Instruction Set

Language Description

RESET Detail

Upon reset, all COGs are shut down, forcing all pins to float. Then, a single COG is booted with a startup program from the main memory's ROM. This program checks for a serial host connection on P31 (RX). If a host is present, a conversation ensues via P30 (TX), and the host may load the main memory's RAM with high-level code. The host may also direct the code to be programmed into an EEPROM before executing it. If no host is present, the startup program attempts to read and execute high-level code from a 24LC256 EEPROM on P28 (SCL) and P29 (SDA). If neither a serial host nor an EEPROM is present, the program will place the chip in shutdown mode. Until a new reset occurs, power is minimized and all pins will be floating.

High-level code is what initially executes from the programmer's perspective. It is capable of booting extra cogs with either other high-level tasks or assembly-language programs. In the case of high-level code execution, a COG is loaded with an interpreter program from the main memory's ROM which executes the high-level code residing in the main memory's RAM.

COG Detail

Each COG has its own 512 x 32-bit register space. These 512 registers are all RAM, except for the last 16 which are special-purpose registers. The RAM space is used for executable code, data, and variables. The last 16 locations serve as interfaces to the HUB, I/O pins, and local COG peripherals.

When a COG is booted, its RAM locations are loaded sequentially from main memory and its special-purpose registers are cleared to zero. After loading, the COG begins executing instructions, starting at location $000. It will continue to execute code until it is stopped or rebooted by either itself or another COG, or a reset occurs.

When a COG is stopped or a reset occurs, all COG I/O registers are instantaneously cleared to 0, canceling all influence the COG had on I/O pins and power consumption.

Here is a summary of the special registers within a COG:

COG Registers

Address	Name	Access	Description
$000-$1EF	-	Read/Write	General Purpose RAM
$1F0	PAR	Read-Only *	Long-address Parameter
$1F1	CNT	Read-Only *	System Counter
$1F2	INA	Read-Only *	Input States for P31..P0
$1F3	INB	Read-Only *	Input States for P63..P32 **
$1F4	OUTA	Read/Write	Output States for P31..P0
$1F5	OUTB	Read/Write	Output States for P63..P32 **

$1F6	DIRA	Read/Write	Direction States for P31..P0
$1F7	DIRB	Read/Write	Direction States for P63..P32 **
$1F8	CTRA	Read/Write	Counter A Control
$1F9	CTRB	Read/Write	Counter B Control
$1FA	FRQA	Read/Write	Counter A Frequency
$1FB	FRQB	Read/Write	Counter B Frequency
$1FC	PHSA	Read/Write	Counter A Phase
$1FD	PHSB	Read/Write	Counter B Phase
$1FE	VCFG	Read/Write	Video Configuration
$1FF	VSCL	Read/Write	Video Scale

* Only accessible as a Source Register (i.e. MOV DEST,SOURCE)
** Allocated for future use, but not currently implemented

The COG executes 32-bit instructions from its 512-register space. The instruction codes contain several bit fields. The general format for an assembly instruction is as follows:

{address label} {execution condition} instruction destination , {#}source {execution effects – comma separated}

Examples:

```
entry           rdlong  temp,PAR      wz      'wait for command (non-0)
      if_z      jmp     #entry
```

32-Bit Instruction Detail

Operation Code	Update Z Flag	Update C Flag	Update Result	Source #	Execution Condition	Destination Register	Source Register / #
31..26	25	24	23	22	21..18	17..9	8..0
iiiiii	z	c	r	i	cccc	ddddddddd	sssssssss

Field	Range	Description
Operation Code	31..26	Perform ADD, SUB, AND, OR, JMP, etc.
Execution Effects	25..23	In conjunction with Controls writes to C Flag, Z Flag, and Destination Register: If bit 25 is set, the Z Flag will be written If bit 24 is set, the C Flag will be written If bit 23 is set, the Destination Register will be written
Source Literal	22	If set, bits 8..0 will be zero-extended and used as a 32-bit constant
Execution Condition	21..18	C and Z flag condition upon which instruction will execute
Destination Register	17..9	Register which supplies the first 32-bit input to the instruction and/or receives the effect of the operation
Source Register or Literal	8..0	Register or constant which supplies the second 32-bit input to the instruction

Source Register or Literal Value

This 9-bit field either selects a register (any of the 512) or provides a zero-extended literal value to be used as the source by the instruction. Bit 22 controls how this field is used: 0 for register, 1 for literal. When entering assembly code, the # symbol sets literal mode.

Destination Register

This 9-bit field selects which of the 512 registers will be used as the destination by the instruction.

Execution Condition

This 4-bit field selects the condition upon which the instruction will execute. The condition is a logical combination of the C and Z flags. The condition codes may precede any instruction and are as follows:

Execution Condition	Instruction Prefix	Instruction Prefix	Instruction Prefix
1111	ALWAYS *		
1100	IF_C		IF_B
0011	IF_NC		IF_AE
1010	IF_Z		IF_E
0101	IF_NZ		IF_NE
1000	IF_C_AND_Z	IF_Z_AND_C	
0100	IF_C_AND_NZ	IF_NZ_AND_C	
0010	IF_NC_AND_Z	IF_Z_AND_NC	
0001	IF_NC_AND_NZ	IF_NZ_AND_NC	IF_A
1110	IF_C_OR_Z	IF_Z_OR_C	IF_BE
1101	IF_C_OR_NZ	IF_NZ_OR_C	
1011	IF_NC_OR_Z	IF_Z_OR_NC	
0111	IF_NC_OR_NZ	IF_NZ_OR_NC	
1001	IF_C_EQ_Z	IF_Z_EQ_C	
0110	IF_C_NE_Z	IF_Z_NE_C	
0000	NEVER		

* ALWAYS is the default, in case no condition is specified.

Execution Effects

After the instruction and destination/source are specified, you may enter any desired execution effect(s). These are two-character symbols that gate the updating of C, Z and D. The C and/or Z flags will only be affected if 'WC' and/or 'WZ' follow an instruction. While no instruction will automatically write C or Z, each instruction has some default write-D behavior which can be overridden by 'WR' or 'NR'. If multiple execution effects are entered after an instruction, they must be separated by commas. Only whitespace is required to separate instruction arguments from the postfixes.

Instruction Postfix	Effect
WC	Write C
WZ	Write Z
WR	Write D
NR	Don't Write D

Propeller Assembly Language Instruction Set

iiiiii zcri cccc ddddddddd sssssssss	Instruction	Description	Z out	C out	r	Clocks
000000 000i 1111 ddddddddd sssssssss	**WRBYTE D,S**	Write D[7..0] to main memory byte S[15..0]	-	-	0	7..22 *
000000 001i 1111 ddddddddd sssssssss	**RDBYTE D,S**	Read main memory byte S[15..0] into D (0-ext'd)	Result = 0	-	1	7..22 *
000001 000i 1111 ddddddddd sssssssss	**WRWORD D,S**	Write D[15..0] to main memory word S[15..1]	-	-	0	7..22 *
000001 001i 1111 ddddddddd sssssssss	**RDWORD D,S**	Read main memory word S[15..1] into D (0-ext'd)	Result = 0	-	1	7..22 *
000010 000i 1111 ddddddddd sssssssss	**WRLONG D,S**	Write D to main memory long S[15..2]	-	-	0	7..22 *
000010 001i 1111 ddddddddd sssssssss	**RDLONG D,S**	Read main memory long S[15..2] into D	Result = 0	-	1	7..22 *
000011 000i 1111 ddddddddd sssssssss	**HUBOP D,S**	Perform hub operation according to S	Result = 0	-	0	7..22 *
000011 0001 1111 ddddddddd ------000	**CLKSET D**	Set the global CLK register to D[7..0]	-	-	0	7..22 *
000011 0011 1111 ddddddddd ------001	**COGID D**	Get *this* COG number (0..7) into D	Result = 0	-	1	7..22 *
000011 0001 1111 ddddddddd ------010	**COGINIT D**	Initialize a COG according to D	Result = 0	No COG free	0	7..22 *
000011 0001 1111 ddddddddd ------011	**COGSTOP D**	Stop COG number D[2..0]	-	-	0	7..22 *
000011 0011 1111 ddddddddd ------100	**LOCKNEW D**	Checkout a new LOCK number (0..7) into D	Result = 0	No LOCK free	1	7..22 *
000011 0001 1111 ddddddddd ------101	**LOCKRET D**	Return LOCK number D[2..0]	-	-	0	7..22 *
000011 0001 1111 ddddddddd ------110	**LOCKSET D**	Set LOCK number D[2..0]	-	Prior LOCK state	0	7..22 *
000011 0001 1111 ddddddddd ------111	**LOCKCLR D**	Clear LOCK number D[2..0]	Result = 0	Prior LOCK state	0	7..22 *
000100 001i 1111 ddddddddd sssssssss	**MUL D,S**	Multiply unsigned D[15..0] by S[15..0]	Result = 0	-	1	future
000101 001i 1111 ddddddddd sssssssss	**MULS D,S**	Multiply signed D[15..0] by S[15..0]	Result = 0	-	1	future
000110 001i 1111 ddddddddd sssssssss	**ENC D,S**	Encode magnitude of S into D, result = 0..31	Result = 0	-	1	future
000111 001i 1111 ddddddddd sssssssss	**ONES D,S**	Get number of 1's in S into D, result = 0..31	Result = 0	-	1	future
001000 001i 1111 ddddddddd sssssssss	**ROR D,S**	Rotate D right by S[4..0] bits	Result = 0	D[0]	1	4
001001 001i 1111 ddddddddd sssssssss	**ROL D,S**	Rotate D left by S[4..0] bits	Result = 0	D[31]	1	4
001010 001i 1111 ddddddddd sssssssss	**SHR D,S**	Shift D right by S[4..0] bits	Result = 0	D[0]	1	4
001011 001i 1111 ddddddddd sssssssss	**SHL D,S**	Shift D left by S[4..0] bits	Result = 0	D[31]	1	4
001100 001i 1111 ddddddddd sssssssss	**RCR D,S**	Rotate carry right into D by S[4..0] bits	Result = 0	D[0]	1	4
001101 001i 1111 ddddddddd sssssssss	**RCL D,S**	Rotate carry left into D by S[4..0] bits	Result = 0	D[31]	1	4
001110 001i 1111 ddddddddd sssssssss	**SAR D,S**	Shift D arithmetically right by S[4..0] bits	Result = 0	D[0]	1	4
001111 001i 1111 ddddddddd sssssssss	**REV D,S**	Reverse 32–S[4..0] bottom bits in D and 0-extend	Result = 0	D[0]	1	4
010000 001i 1111 ddddddddd sssssssss	**MINS D,S**	Set D to S if signed (D < S)	D = S	Signed (D < S)	1	4
010001 001i 1111 ddddddddd sssssssss	**MAXS D,S**	Set D to S if signed (D => S)	D = S	Signed (D < S)	1	4
010010 001i 1111 ddddddddd sssssssss	**MIN D,S**	Set D to S if unsigned (D < S)	D = S	Unsigned (D < S)	1	4
010011 001i 1111 ddddddddd sssssssss	**MAX D,S**	Set D to S if unsigned (D => S)	D = S	Unsigned (D < S)	1	4
010100 001i 1111 ddddddddd sssssssss	**MOVS D,S**	Insert S[8..0] into D[8..0]	Result = 0	-	1	4
010101 001i 1111 ddddddddd sssssssss	**MOVD D,S**	Insert S[8..0] into D[17..9]	Result = 0	-	1	4
010110 001i 1111 ddddddddd sssssssss	**MOVI D,S**	Insert S[8..0] into D[31..23]	Result = 0	-	1	4
010111 001i 1111 ddddddddd sssssssss	**JMPRET D,S**	Insert PC+1 into D[8..0] and set PC to S[8..0]	Result = 0	-	1	4
010111 000i 1111 --------- sssssssss	**JMP S**	Set PC to S[8..0]	Result = 0	-	0	4
010111 0011 1111 ????????? sssssssss	**CALL #S**	Like JMPRET, but assembler handles details	Result = 0	-	1	4
010111 0001 1111 --------- ---------	**RET**	Like JMP, but assembler handles details	Result = 0	-	0	4
011000 000i 1111 ddddddddd sssssssss	**TEST D,S**	AND S with D to affect flags only	Result = 0	Parity of Result	0	4
011000 001i 1111 ddddddddd sssssssss	**AND D,S**	AND S into D	Result = 0	Parity of Result	1	4
011001 001i 1111 ddddddddd sssssssss	**ANDN D,S**	AND !S into D	Result = 0	Parity of Result	1	4
011010 001i 1111 ddddddddd sssssssss	**OR D,S**	OR S into D	Result = 0	Parity of Result	1	4
011011 001i 1111 ddddddddd sssssssss	**XOR D,S**	XOR S into D	Result = 0	Parity of Result	1	4
011100 001i 1111 ddddddddd sssssssss	**MUXC D,S**	Copy C to bits in D using S as mask	Result = 0	Parity of Result	1	4
011101 001i 1111 ddddddddd sssssssss	**MUXNC D,S**	Copy !C to bits in D using S as mask	Result = 0	Parity of Result	1	4
011110 001i 1111 ddddddddd sssssssss	**MUXZ D,S**	Copy Z to bits in D using S as mask	Result = 0	Parity of Result	1	4
011111 001i 1111 ddddddddd sssssssss	**MUXNZ D,S**	Copy !Z to bits in D using S as mask	Result = 0	Parity of Result	1	4
100000 001i 1111 ddddddddd sssssssss	**ADD D,S**	Add S into D	Result = 0	Unsigned Carry	1	4
100001 001i 1111 ddddddddd sssssssss	**SUB D,S**	Subtract S from D	Result = 0	Unsigned Borrow	1	4
100001 000i 1111 ddddddddd sssssssss	**CMP D,S**	Compare D to S	Result = 0	Unsigned Borrow	0	4
100010 001i 1111 ddddddddd sssssssss	**ADDABS D,S**	Add absolute S into D	Result = 0	Unsigned Carry	1	4
100011 001i 1111 ddddddddd sssssssss	**SUBABS D,S**	Subtract absolute S from D	Result = 0	Unsigned Borrow	1	4
100100 001i 1111 ddddddddd sssssssss	**SUMC D,S**	Sum either –S if C or S if !C into D	Result = 0	Signed Overflow	1	4
100101 001i 1111 ddddddddd sssssssss	**SUMNC D,S**	Sum either S if C or –S if !C into D	Result = 0	Signed Overflow	1	4
100110 001i 1111 ddddddddd sssssssss	**SUMZ D,S**	Sum either –S if Z or S if !Z into D	Result = 0	Signed Overflow	1	4
100111 001i 1111 ddddddddd sssssssss	**SUMNZ D,S**	Sum either S if Z or –S if !Z into D	Result = 0	Signed Overflow	1	4
101000 001i 1111 ddddddddd sssssssss	**MOV D,S**	Set D to S	Result = 0	S[31]	1	4
101001 001i 1111 ddddddddd sssssssss	**NEG D,S**	Set D to –S	Result = 0	S[31]	1	4
101010 001i 1111 ddddddddd sssssssss	**ABS D,S**	Set D to absolute S	Result = 0	S[31]	1	4
101011 001i 1111 ddddddddd sssssssss	**ABSNEG D,S**	Set D to –absolute S	Result = 0	S[31]	1	4
101100 001i 1111 ddddddddd sssssssss	**NEGC D,S**	Set D to either –S if C or S if !C	Result = 0	S[31]	1	4
101101 001i 1111 ddddddddd sssssssss	**NEGNC D,S**	Set D to either S if C or –S if !C	Result = 0	S[31]	1	4
101110 001i 1111 ddddddddd sssssssss	**NEGZ D,S**	Set D to either –S if Z or S if !Z	Result = 0	S[31]	1	4
101111 001i 1111 ddddddddd sssssssss	**NEGNZ D,S**	Set D to either S if Z or –S if !Z	Result = 0	S[31]	1	4
110000 000i 1111 ddddddddd sssssssss	**CMPS D,S**	Compare-signed D to S	Result = 0	Signed Borrow	0	4
110001 000i 1111 ddddddddd sssssssss	**CMPSX D,S**	Compare-signed-extended D to S+C	Z & (Result = 0)	Signed Borrow	0	4
110010 001i 1111 ddddddddd sssssssss	**ADDX D,S**	Add-extended S+C into D	Z & (Result = 0)	Unsigned Carry	1	4
110011 001i 1111 ddddddddd sssssssss	**SUBX D,S**	Subtract-extended S+C from D	Z & (Result = 0)	Unsigned Borrow	1	4
110011 000i 1111 ddddddddd sssssssss	**CMPX D,S**	Compare-extended D to S+C	Z & (Result = 0)	Unsigned Borrow	0	4
110100 001i 1111 ddddddddd sssssssss	**ADDS D,S**	Add-signed S into D	Result = 0	Signed Overflow	1	4
110101 001i 1111 ddddddddd sssssssss	**SUBS D,S**	Subtract-signed S from D	Result = 0	Signed Overflow	1	4
110110 001i 1111 ddddddddd sssssssss	**ADDSX D,S**	Add-signed-extended S+C into D	Z & (Result = 0)	Signed Overflow	1	4
110111 001i 1111 ddddddddd sssssssss	**SUBSX D,S**	Subtract-signed-extended S+C from D	Z & (Result = 0)	Signed Overflow	1	4
111000 000i 1111 ddddddddd sssssssss	**CMPSUB D,S**	Subtract S from D if D => S	D = S	Unsigned (D => S)	0	4
111001 001i 1111 ddddddddd sssssssss	**DJNZ D,S**	Dec D, jump if not zero to S (no jump = 8 clocks)	Result = 0	Unsigned Borrow	1	4 or 8
111010 000i 1111 ddddddddd sssssssss	**TJNZ D,S**	Test D, jump if not zero to S (no jump = 8 clocks)	Result = 0	0	0	4 or 8
111011 000i 1111 ddddddddd sssssssss	**TJZ D,S**	Test D, jump if zero to S (no jump = 8 clocks)	Result = 0	0	0	4 or 8
111100 000i 1111 ddddddddd sssssssss	**WAITPEQ D,S**	Wait for pins equal - (INA & S) = D	Result = 0	-	0	5+
111101 000i 1111 ddddddddd sssssssss	**WAITPNE D,S**	Wait for pins not equal - (INA & S) != D	Result = 0	-	0	5+
111110 001i 1111 ddddddddd sssssssss	**WAITCNT D,S**	Wait for CNT = D, then add S into D	Result = 0	Unsigned Carry	1	5+
111111 000i 1111 ddddddddd sssssssss	**WAITVID D,S**	Wait for video peripheral to grab D and S	Result = 0	-	0	5+
------ ---- 0000 --------- ---------	**NOP**	No operation, just elapses 4 clocks	-	-	-	4

* The HUB allows each COG an opportunity to execute a HUB instruction every 16 clocks. Because each COG runs independently of the HUB, each COG must sync to the HUB when executing a HUB instruction. This will cause a HUB instruction to take between 7 and 22 clocks. Afterwards, there will be 9 free clocks before a subsequent HUB instruction can execute and take the minimal 7 clocks. This is enough time to execute two 4-clock instructions without missing the next sync. So, to minimize clock waste, you can insert two normal instructions between any two otherwise-contiguous HUB instructions, without any increase in execution time. Beware that HUB instructions can cause execution timing to appear indeterminate - particularly, the first HUB instruction in a sequence.

HUB Instructions

The instructions which access main memory, the global CLK register, the COGs, and the LOCKs are all 'HUB' instructions. What critically transpires during their execution is a function of the central HUB, which coordinates chip-wide operations.

The HUB runs at half the COGs' clock frequency, serving each of the eight COGs, in turn, with each subsequent clock. From the COGs' perspectives, this is once every 16 clocks. Because the HUB runs steadily, dedicating a clock cycle to each COG, and because the COGs run independently, taking various numbers of clocks for different instructions, each COG must re-sync to the HUB whenever it executes a HUB instruction. This results in execution times ranging from 7 to 22 clocks for each hub instruction. Once a HUB instruction has executed, there will be 9 free clocks before another HUB instruction could execute and take the minimal 7 clocks. Nine clocks is enough time for two 4-clock instructions to execute before another HUB instruction would take 8 clocks. So, to minimize clock waste, you can insert two 4-clock instructions between any two otherwise-contiguous HUB instructions without any increase in execution time. Beware that HUB instructions can inject jitter into your execution schedule – particularly the first one in a sequence. For deterministic timing, you might want to place them outside of time-critical code sequences in which the HUB-sync is unknown.

The following sub-sections elaborate on different HUB instruction groups.

Main memory access

The HUB instructions which access main memory are:

```
RDBYTE  D,S     'Read byte at S into D and zero-extend
RDWORD  D,S     'Read word at S into D and zero-extend
RDLONG  D,S     'Read long at S into D

WRBYTE  D,S     'Write D into byte at S
WRWORD  D,S     'Write D into word at S
WRLONG  D,S     'Write D into long at S
```

The main memory is 64K bytes in size and can be accessed from any COG. It is accessible as 64K bytes, 32K words, or 16K longs.

All accesses are aligned according to data size. Words can only begin at even addresses and longs can only begin at even-even addresses. Of course, the programmer can arrange his data any way he wants, but to take advantage of efficient word and long accesses, he must pay attention to alignment.

CLK register write

Inside the HUB is a global CLK register which selects the master clock. A single HUB instruction used to set this register:

```
CLKSET  D       'Write D[7..0] into the global CLK register
```

See the documentation on the Global CLK Register for a detailed description of its function.

COG control

There are three HUB instructions which govern the starting and stopping of COGs:

```
COGID   D       'Get this COG number into D
COGINIT D       'Init a cog according to D
COGSTOP D       'Stop cog number D
```

COGID writes the executing COG number into D. This instruction is used when a COG needs to know 'who' it is, among the eight, for the purpose of stopping or restarting itself.

COGINIT is the instruction used to start or restart a COG. The D register has three fields within it that determine which COG gets started, where its program begins in main memory, and what its PAR register will contain.

D[31..18] will be written to bits [15..2] of the started COG's PAR register. By this mechanism, a long address parameter can be conveyed to a COG at start-time. This parameter is intended to be used as a pointer to some agreed-upon structure in main memory via which communication with another COG (or COGs) may take place.

D[17..4] specifies the long address in main memory of the COG program to be loaded. COG registers $000..$1EF will be loaded sequentially, starting at this address. The writable COG registers $1F4..$1FF will be initialized to 0. When the loading and initializing are complete, the started COG will begin executing from register $000.

D[3] determines whether a new COG or a specified COG will be started.

If D[3] is '1', the HUB will start the lowest-numbered inactive COG and return that COG's number (0..7) into D. The C flag will also be used to convey whether or not there was an inactive COG available. A '0' in C indicates that the operation was successful. A '1' indicates that no COG was available and none was started. Be sure to put both the 'WR' and 'WC' modifiers after the COGINIT instruction so that these results will be written into D and C. Starting COGs in this fashion means that you will never have to plan which COG will run what program. The HUB, which knows the instantaneous state of all COGs, will do the picking for you.

If D[3] is '0', the HUB will start the COG numbered in D[2..0]. This is useful for restarting active COGs. To start a new COG, it is recommended that you use the D[3] = '1' approach, detailed above.

COGSTOP simply stops the COG numbered in D[2..0]. When a COG is stopped, it receives no clock and consumes no power. It will be held in an inactive reset state until started by a COGINIT instruction.

LOCK usage

LOCKs are special global bits which can be utilized to solve the problem of multiple-access contention, or *cross-COG clobbering*.

Imagine you have an area in main memory which holds some data that is to be accessed by more than one COG. If the elemental data exceeds a long, in size, it will take multiple reads or writes for any COG to access it. It would be necessary to avoid the possibility of one or more COGs reading data while one or more other COGs are writing (or *think* they are writing) the same data. Access needs to be limited to one COG at a time, in order to avoid misreads and miswrites. Such exclusive access control can be achieved by using a LOCK.

A LOCK is a global bit that can be set or cleared by any COG through certain HUB instructions. At the time a COG sets or clears a LOCK, it can also learn the prior state of that LOCK. The fact that the HUB allows only one COG at a time to set or clear a LOCK, makes this an effective access control mechanism.

To use a LOCK, all COGs which intend to share some resource must agree on a LOCK number and its initial state. For our discussion, let's make the LOCK's initial state '0'. Now, for any COG to gain exclusive access to the shared resource, it must keep setting the LOCK until it sees that its prior state was '0'. After this, all other COGs will get only '1's back. The winning COG now has exclusive access to the resource. When it is done accessing the resource, it must clear the LOCK so that another COG can gain access. LOCKs may be used creatively for many such purposes.

The HUB maintains an inventory of eight LOCKs. It keeps track of which LOCKs are available for check-out and notes when a LOCK is returned. On reset, all LOCKs are 'returned' and ready for check-out. Though it's possible to set or clear any arbitrary LOCK, it is recommended that you check-out a LOCK, so that the HUB will know that it is in use and will not grant it to some other COG. It's also critical to return the LOCK when you are through with it, so that it may be checked out again.

There are four HUB instructions used to manipulate LOCKs:

```
LOCKNEW D       wc      'Check-out a new LOCK number into D, C=1 if none
LOCKRET D               'Return LOCK number D
LOCKSET D       wc      'Set LOCK number D, C=prior LOCK state
LOCKCLR D       wc      'Clear LOCK number D, C=prior LOCK state
```

LOCKNEW checks out a new LOCK from the HUB and puts its number (0..7) into D. The C flag is used to convey whether or not the operation was successful. A '0' indicates success, while a '1' indicates that no free LOCK was available. Be sure to put a 'WC' modifier after the instruction, so that C will be written.

LOCKRET returns the LOCK numbered in D[2..0] to the HUB's inventory.

LOCKSET sets the LOCK numbered in D[2..0] and, if 'WC' is after the instruction, writes the LOCK's prior state into the C flag.

LOCKCLR clears the LOCK numbered in D[2..0] and, if 'WC' is after the instruction, writes the LOCK's prior state into the C flag.

Branching Instructions

Branching instructions are used to alter the course of the COG's program counter, or PC. The PC is a 9-bit counter that tracks the program's execution address. It normally increments with each instruction, but it can be loaded with an arbitrary value via branching instructions.

Important note: All the branching instructions (except 'RET') use S to specify the branch address. Always remember to put '#' in front of the S expression when you intend to jump to a constant address. This will almost always be the case in your code. If you don't use the '#', you will be specifying a register's contents as your branch address. Sometimes, you may want to do this, but most often your intent will be to branch to a constant address and you must remember to use '#'. For the applicable instructions cited in this section, both forms will be shown to keep you mindful of this critical issue.

Important note #2: Remember, like all other instructions, these branches can be preceded by conditionals, making them into conditional branches.

The simplest branch instruction is 'JMP S'. This simply sets the PC to S[8..0]:

```
            JMP     #S      'Jump to the constant address S
            JMP     S       'Jump to where register S points
```

The most complex branch instruction is 'JMPRET D,S'. This is like 'JMP S' but has the additional effect of writing PC+1 (what would have been the next execution address) into bits 8..0 of D. The purpose of that extra action is to insert a 'return address' into a 'JMP #S' instruction's sssssssss bit field. This way, if the code branched to by 'JMPRET D,S' ends with a 'JMP #S' instruction at the D address, that segment of code effectively becomes a subroutine that many identical 'JMPRET D,S' instructions could branch to, and then return from.

Here is 'JMPRET':

```
            JMPRET  D,#S    'Jump to the constant address S and set D[8..0] to PC+1
            JMPRET  D,S     'Jump to where register S points and set D[8..0] to PC+1
```

To keep you sane, the assembler provides special 'CALL' and 'RET' instructions which form 'JMPRET D,#S' and 'JMP #S' instructions, respectively. This makes it easy to realize the subroutine scheme described above. These special instructions must be used with coordinated symbol names that you make up. To use the 'CALL' instruction, you must specify a '#' followed by the symbolic name of the subroutine you are calling. There must be, somewhere in your program, an identical symbol name which ends with '_ret' and precedes a 'RET' instruction. The assembler uses the address of the +'_ret' symbol as the D in the 'JMPRET D,#S', or 'CALL' instruction; the S comes from the subroutine's start symbol. The 'RET' instruction is assembled as 'JMP #0', but gets modified at run-time. Here is a contextual example of these instructions:

```
            CALL    #sub1   'Call to the constant address "sub1"
                            '…and set the sssssssss bit field of "sub1_ret" to PC+1

'-------------------------------------------------------------------------------

sub1        <code>          'Start of "sub1" subroutine
sub1_ret    RET             'Return to caller (sssssssss modified, jump to #S)
```

Note that the COG has no call stack. It uses end-of-subroutine jumps to return to callers. For this reason, subroutines cannot be called recursively. However, there is no 'depth' limit.

Finally, there are three conditional branching instructions which branch according to D:

```
            DJNZ    D,#S    'Decrement D, jump to constant address S if D is not 0
            DJNZ    D,S     'Decrement D, jump to where register S points if D is not 0

            TJNZ    D,#S    'Test D, jump to constant address S if D is not 0
            TJNZ    D,S     'Test D, jump to where register S points if D is not 0

            TJZ     D,#S    'Test D, jump to constant address S if D is 0
            TJZ     D,S     'Test D, jump to where register S points if D is 0
```

These D-dependent branches are very useful for fast looping. They take 4 clocks when they branch, and 8 clocks when they don't.

Add, Subtract, and Compare Instructions

There is a variety of add, subtract, and compare instructions which warrants elaboration. Some instructions perform unsigned math, while others perform signed math. Some perform base-long operations, while others perform extended-long operations. The differences lie in how the Z and C flags are treated, particularly the C flag. By having several adder-based instructions that treat the C flag differently, the need for separate overflow and sign flags is gotten around. This two-flag system allows for a compact, but complete 4-bit condition code to gate the operation of each instruction.

Below is a table of the basic math instructions:

ADD/SUB/CMP	Base	Extended

Unsigned	ADD D,S
	SUB D,S
	CMP D,S
		ADDX D,S
		SUBX D,S
		CMPX D,S

Signed	ADDS D,S
	SUBS D,S
	CMPS D,S
		ADDSX D,S
		SUBSX D,S
		CMPSX D,S

Here are the unsigned base instructions, shown with both flags written:

```
ADD     D,S     wz,wc   'Add S into D, Z=1 if 0, C=1 if carry
SUB     D,S     wz,wc   'Subtract S from D, Z=1 if 0, C=1 if borrow
CMP     D,S     wz,wc   'Compare D to S, Z=1 if 0, C=1 if borrow
```

The 'ADD' instruction produces a carry, or C=1, if the addition of S into D causes a roll-over in D. In other words, the result is greater than $FFFFFFFF. The 'SUB' instruction produces a borrow, or C=1, if the subtraction of S from D causes a roll-under in D. Or, in other words, the result is less than 0 (S is greater than D). The 'SUB' and 'CMP' instructions are the same, except that 'CMP' does not store the result – it only indicates through the Z and C flags if D is above, below, or equal to S.

Next are the signed base instructions, shown with both flags written:

```
ADDS    D,S     wz,wc   'Add-signed S into D, Z=1 if 0, C=1 if overflow
SUBS    D,S     wz,wc   'Subtract-signed S from D, Z=1 if 0, C=1 if overflow
CMPS    D,S     wz,wc   'Compare-signed D to S, Z=1 if 0, C=1 if borrow
```

In the signed 'ADDS' and 'SUBS' instructions, an overflow, or C=1, occurs when the result is either above $7FFFFFFF or below $80000000 – the range of signed long values. The signed 'CMPS' instruction, however, produces a signed borrow, or C=1, if S is greater than D, signs considered.

The unsigned and extended instructions are shown below with both flags written:

```
ADDX    D,S     wz,wc   'Add-extended S+C into D, AND Z, C=1 if carry
SUBX    D,S     wz,wc   'Subtract-extended S+C from D, AND Z, C=1 if borrow
CMPX    D,S     wz,wc   'Compare-extended D to S+C, AND Z, C=1 if borrow
```

These instructions differ from 'ADD', 'SUB', and 'CMP' by incorporating C into the computation, and by AND'ing the old Z with what would have been the new Z. These extra qualities make these instructions chainable. For example, the following double-long operations wind up with Z and C valid for the entire 64-bit operation:

```
ADD     D0,S0   wz,wc   'Add lower longs, affect flags
ADDX    D1,S1   wz,wc   'Add upper longs, Z=1 if 0, C=1 if carry

SUB     D0,S0   wz,wc   'Subtract lower longs, affect flags
SUBX    D1,S1   wz,wc   'Subtract upper longs, Z=1 if 0, C=1 if borrow

CMP     D0,S0   wz,wc   'Compare lower longs, affect flags
CMPX    D1,S1   wz,wc   'Compare upper longs, Z=1 if 0, C=1 if borrow
```

These operations could be further extended by adding an extra 'ADDX', 'SUBX', or 'CMPX' for every additional long pair to be operated on.

The signed and extended instructions are as follows, shown with both flags written:

```
ADDSX    D,S     wz,wc   'Add-signed-extended S+C into D, AND Z, C=overflow
```

```
SUBSX    D,S     wz,wc   'Sub-signed-extended S+C from D, AND Z, C=overflow
CMPSX    D,S     wz,wc   'Compare-signed-extended D to S+C, AND Z, C=borrow
```

Like 'ADDX', 'SUBX', and 'CMPX', these instructions incorporate C into the operation, and AND the old Z with what would have been the new Z. The difference is in their C flag output, which is signed. Here they are, used in double-long operations, with both Z and C valid afterwards for the entire 64-bit operation:

```
ADD     D0,S0   wz,wc   'Add lower longs, then signed-extended uppers
ADDSX   D1,S1   wz,wc   'Z=1 if 0, C=1 if overflow

SUB     D0,S0   wz,wc   'Subtract lower longs, then signed-extended uppers
SUBSX   D1,S1   wz,wc   'Z=1 if 0, C=1 if overflow

CMP     D0,S0   wz,wc   'Compare lower longs, then signed-extended uppers
CMPSX   D1,S1   wz,wc   'Z=1 if 0, C=1 if signed borrow
```

To extend these further, 'ADDX', 'SUBX', and 'CMPX' instructions may be inserted between the shown pairs to handle any middle-longs. In a multiple-long signed operation, the beginning instruction is unsigned, any middle instructions are unsigned-extended, and the last instruction is signed-extended.

All these add, subtract, and compare instructions are quite similar. The differences lie in whether they are signed (C indicates overflow or signed borrow) and whether they are extended (C flag incorporated into operation, and Z flag AND'd).

Multiplication, Division, and Square Root

Multiplication, division, and square root can be computed by using add, subtract, and shift instructions.

Here is an unsigned multiplier routine that multiplies two 16-bit values to yield a 32-bit product:

```
'
' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
multiply        shl     x,#16           'get multiplicand into x[31..16]
                mov     t,#16           'ready for 16 multiplier bits
                shr     y,#1    wc      'get initial multiplier bit into c

:loop   if_c    add     y,x     wc      'if c set, add multiplicand into product
                rcr     y,#1    wc      'get next multiplier bit into c, shift product
                djnz    t,#:loop        'loop until done

multiply_ret    ret                     'return with product in y[31..0]
```

The above routine's execution time could be cut by ~1/3 if the loop was unrolled, repeating the 'add'/'rcr' sequence and getting rid of the 'DJNZ' instruction.

Division is like multiplication, but backwards. It is potentially more complex, though, because a comparison test must be performed before a subtraction can take place. To remedy this, there is a special 'CMPSUB D,S' instructions which tests to see if a subtraction can be performed without causing an underflow. If no underflow would occur, the subtraction takes place and the C output is 1. If an underflow would occur, D is left alone and the C output is 0.

Here is an unsigned divider routine that divides a 32-bit value by a 16-bit value to yield a 16-bit quotient and a 16-bit remainder:

```
'
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide          shl     y,#15           'get divisor into y[30..15]
                mov     t,#16           'ready for 16 quotient bits

:loop           cmpsub  x,y     wc      'if y =< x then subtract it, quotient bit into c
                rcl     x,#1            'rotate c into quotient, shift dividend
                djnz    t,#:loop        'loop until done
```

```
divide_ret      ret                     'quotient in x[15..0], remainder in x[31..16]
```

Like the multiplier routine, this divider routine could be recoded with a sequence of 16 'CMPSUB'+'RCL' instruction pairs to get rid of the 'DJNZ' and cut execution time by ~1/3. By making such changes, speed can often be gained at the expense of code size.

Here is a square-root routine that uses the 'CMPSUB' instruction:

```
'
' Compute square-root of y[31..0] into x[15..0]
'
root            mov     a,#0            'reset accumulator
                mov     x,#0            'reset root
                mov     t,#16           'ready for 16 root bits

:loop           shl     y,#1    wc      'rotate top two bits of y into accumulator
                rcl     a,#1
                shl     y,#1    wc
                rcl     a,#1
                shl     x,#2            'determine next bit of root
                or      x,#1
                cmpsub  a,x     wc
                shr     x,#2
                rcl     x,#1
                djnz    t,#:loop        'loop until done

root_ret        ret                     'square root in x[15..0]
```

Many complex math functions can be realized by additions, subtractions, and shifts. Though specific examples were given here, these types of algorithms may be coded in many different ways to best suit the application.

WAIT Instructions

The 'wait' instructions are used to stall the COG until some target condition is met. Once a wait instruction is engaged, the target condition is re-checked every clock cycle. Once the condition is met, the instruction finishes and the next instruction executes. During the wait, COG power consumption is reduced. These instructions are mainly used to align a COG's execution timing with some event.

The first two wait instructions deal with the I/O pins' input states. They wait for some set of pins to either equal, or not equal, some set of states. The S operand is used as a mask to isolate the pins of importance, while the D operand is the value to compare the isolated pins to. These instructions are as follows:

```
            WAITPEQ D,S             'Wait for (INA & S) to equal D
            WAITPNE D,S             'Wait for (INA & S) to not equal D
```

The 'WAITCNT D,S' instruction waits for the global free-running 32-bit counter to equal some value. The global counter increments at the COG clock rate and can be read via the CNT register in each COG. Keep in mind that 32 bits can take a long time to wrap around if you miss the mark. This instruction is:

```
            WAITCNT D,S             'Wait for CNT to equal D, then add S into D
```

By adding S into D, this instruction can singly serve as a synchronization mechanism to align execution to a fixed period. Here is an example of this:

```
            mov     dira,#1         'Make P0 an output
            add     time,cnt        'Add cnt into time to accommodate the initial sync

:loop       waitcnt time,period     '(re)sync to cnt, adding in the period afterwards
            xor     outa,#1         'toggle P0 – always on the same relative clock
            <code>                  'put non-deterministic code after time-critical code
            jmp     #:loop          'loop to re-sync to the next period
```

```
time        long    3               'cnt-tracking variable, initial 3 accommodates sync
period      long    100             '100 clocks per period
```

Lastly, there is the 'WAITVID D,S' instruction which passes color and pixel data to the video peripheral. The video peripheral must be running for this instruction not to hang:

```
            WAITVID D,S             'Hand off colors in D, pixels in S
```

When the video peripheral is ready for its next set of color and pixel data, it grabs D and S, regardless of what instruction the COG is currently executing. It is critical that the COG be waiting in a 'WAITVID D,S' instruction when this happens – otherwise, the video peripheral picks up indeterminate data and displays it. You have to make sure your program beats the video peripheral to the pickup.

Main Memory

Address Range	Memory Type	Contents
$0000-$7FFF	RAM	User Memory – 8,192 longs / 16,384 words / 32,768 bytes
$8000-$BFFF	ROM	Character Definitions – 4,096 longs define 256 different 16 x 32-pixel characters
$C000-$CFFF	ROM	Log Table – 2,048 words of fractional exponents
$D000-$DFFF	ROM	Anti-Log Table – 2,048 words of top-bit-set 17-bit mantissas
$E000-$F001	ROM	Sine Table – 2,049 unsigned words cover 0° to 90° of sine, inclusively
$F002-$FFFF	ROM	Booter and Interpreter programs

The tables in ROM are intended to facilitate real-time signal processing by high-speed assembly language programs. High-level programs may not find much direct use for this data, though they may leverage objects which take advantage of it.

User Memory ($0000-$7FFF)

The first half of main memory is all RAM. This space is used for your program, variables, and stack(s).

When a program is loaded into the chip, either serially or from an external EEPROM, this entire memory space is written. Your program will begin at $0010 and extend for some number of longs. The area after your program, extending to $7FFF, is used as stack space. Data spanning from $0000 to $000F is used mainly to hold the initial interpreter pointers. However, there are two values stored here that might be of interest to your program: a long at $0000 contains the initial master clock frequency, in Hertz, and a byte at $000F contains the initial value written into the global CLK register. If you change the global CLK register, you will need to update these two variables so that objects which reference them will have current information. The master clock frequency can be read as 'FREQ' in the high-level language. It can also be read *and written* as 'LONG[0]'. The global CLK register copy can be read and written as 'BYTE[15]'.

Character Definitions ($8000-$BFFF)

The first half of ROM is dedicated to a set of 256 character definitions. Each character definition is 16 pixels wide by 32 pixels tall. These character definitions can be used for video generation, graphical LCD's, printing, etc.

The character set is based on a North American / Western European layout, with many specialized characters added and inserted. There are connecting waveform and schematic building-block characters, Greek characters commonly used in electronics, and several arrows and bullets.

The character definitions are numbered 0 to 255 from left-to-right, then top-to-bottom, per the diagram below. They are arranged as follows: Each pair of adjacent even-odd characters is merged together to form 32 longs. The first character pair is located in $8000-$807F. The second pair occupies $8080-$80FF, and so on, until the last pair fills $BF80-$BFFF. The character pairs are merged such that each character's 16 horizontal pixels (per row) are spaced apart and interleaved with their neighbors' such that the even character takes bits 0, 2, 4, ...30, and the odd character takes bits 1, 3, 5, ...31. The leftmost pixels are in the lowest bits, while the rightmost pixels are in the highest bits. This forms a long for each row of pixels in the character pair. 32 such longs, building from top row down to bottom, make up the complete merged-pair definition. The definitions are encoded in this manner so that a COG video peripheral can handle the merged longs directly, using color selection to display either the even or the odd character.

Some character codes have inescapable meanings, such as 9 for Tab, 10 for Line Feed, and 13 for Carriage Return (0 can also be touchy). These character codes invoke *actions* and do not equate to static character definitions. For this reason, their character definitions have been used for special four-color characters. These four-color characters are used for drawing 3-D box edges at run-time and are implemented as 16 x 16 pixel cells, as opposed to the normal 16 x 32 pixel cells. They occupy even-odd character pairs 0-1, 8-9, 10-11, and 12-13.

Log and Anti-Log Tables ($C000-DFFF)

The log and anti-log tables are useful for converting values between their number form and exponent form.

When numbers are encoded into exponent form, simple math operations take on more complex effects. For example 'add' and 'subtract' become 'multiply' and 'divide'. 'Shift-left' becomes 'square' and 'shift-right' becomes 'square-root'. 'Divide by 3' will produce 'cube root'. Once the exponent is converted back to a number, the result will be apparent. This process is imperfect, but quite fast.

For applications where many multiplies and divides must be performed in the absence of many additions and subtractions, exponential encoding can greatly speed things up. Exponential encoding is also useful for compressing numbers into fewer bits – sacrificing resolution at higher magnitude. In many applications, such as audio synthesis, the nature of signals is logarithmic in both frequency and magnitude. Processing such data in exponent form is quite natural and efficient, as it lends a 'linear' simplicity to what is actually logarithmic.

The code examples given below use each table's samples verbatim. Higher resolution could be achieved by linearly interpolating between table samples, since the slope change is very slightly from sample to sample. The cost, though, would be larger code and lower execution speed.

Log Table ($C000-$CFFF)

The log table contains data used to convert unsigned numbers into base-2 exponents.

The log table is comprised of 2,048 unsigned words which make up the base-2 *fractional* exponents of numbers. To use this table, you must first determine the *integer* portion of the exponent of the number you are converting. This is simply the leading bit position. For $60000000 this would be 30 ($1E). This integer portion will always fit within 5 bits. Isolate these 5 bits into the result so that they occupy bit positions 20..16. In our case of $60000000, we would now have a partial result of $001E0000. Next, top-justify and isolate the first 11 bits below the leading bit into positions 11..1. This would be $0800 for our example. Add $C000 for the log table base and you now have the actual word address of the fractional exponent. By reading the word at $C800, we get the value $95C0. Adding this into the partial result yields $001E95C0 – that's $60000000 in exponent form. Note that bits 20..16 make up the integer portion of the exponent, while bits 15..0 make up the fractional portion, with bit 15 being the ½, bit 14 being the ¼, and so on, down to bit 0. The exponent can now be manipulated by adding, subtracting, and shifting. Always insure that your math operations will never drive the exponent below 0 or cause it to overflow bit 20. Otherwise, it may not convert back to a number correctly.

Here is a routine that will convert an unsigned number into its base-2 exponent using the log table:

```
'
' Convert number to exponent
'
' on entry: num holds 32-bit unsigned value
' on exit:  exp holds 21-bit exponent with 5 integer bits and 16 fractional bits
'
numexp          mov     exp,#0                  'clear exponent

                test    num,num4        wz      'get integer portion of exponent
                muxnz   exp,exp4                'while top-justifying number
    if_z        shl     num,#16
                test    num,num3        wz
                muxnz   exp,exp3
```

```
    if_z        shl     num,#8
                test    num,num2        wz
                muxnz   exp,exp2
    if_z        shl     num,#4
                test    num,num1        wz
                muxnz   exp,exp1
    if_z        shl     num,#2
                test    num,num0        wz
                muxnz   exp,exp0
    if_z        shl     num,#1

                shr     num,#30-11              'justify sub-leading bits as word offset
                and     num,table_mask          'isolate table offset bits
                add     num,table_log           'add log table address
                rdword  num,num                 'read fractional portion of exponent
                or      exp,num                 'combine fractional and integer portions

numexp_ret      ret                             '91..106 clocks
                                                '(variance is due to HUB sync on RDWORD)

num4            long    $FFFF0000
num3            long    $FF000000
num2            long    $F0000000
num1            long    $C0000000
num0            long    $80000000
exp4            long    $00100000
exp3            long    $00080000
exp2            long    $00040000
exp1            long    $00020000
exp0            long    $00010000
table_mask      long    $0FFE                   'table offset mask
table_log       long    $C000                   'log table base

num             long    0                       'input
exp             long    0                       'output
```

Anti-Log Table ($D000-$DFFF)

The anti-log table contains data used to convert base 2 exponents into unsigned numbers.

The anti-log table is comprised of 2,048 unsigned words which are each the lower 16-bits of a 17-bit mantissa (the 17th bit is implied and must be set separately). To use this table, shift the top 11 bits of the exponent fraction (bits 15..5) into bits 11..1 and isolate. Add $D000 for the anti-log table base. Read the word at that location into the result – this is the mantissa. Next, shift the mantissa left to bits 30..15 and set bit 31 – the missing 17th bit of the mantissa. The last step is to shift the result right by 31 minus the exponent integer in bits 20..16. The exponent is now converted to an unsigned number.

Here is a routine that will convert a base-2 exponent into an unsigned number using the anti-log table:

```
'
' Convert exponent to number
'
' on entry: exp holds 21-bit exponent with 5 integer bits and 16 fraction bits
' on exit:  num holds 32-bit unsigned value
'
expnum          mov     num,exp                 'get exponent into number
                shr     num,#15-11              'justify exponent fraction as word offset
                and     num,table_mask          'isolate table offset bits
                or      num,table_antilog       'add anti-log table address
                rdword  num,num                 'read mantissa word into number
                shl     num,#15                 'shift mantissa into bits 30..15
                or      num,num0                'set top bit (17th bit of mantissa)
                shr     exp,#20-4               'shift exponent integer into bits 4..0
                xor     exp,#$1F                'inverse bits to get shift count
                shr     num,exp                 'shift number into final position
```

```
expnum_ret      ret                             '47..62 clocks
                                                '(variance is due to HUB sync on RDWORD)

num0            long    $80000000               '17th bit of the mantissa
table_mask      long    $0FFE                   'table offset mask
table_antilog   long    $C000                   'anti-log table base

exp             long    0                       'input
num             long    0                       'output
```

Sine Table ($E000-$F001)

The sine table provides 2,049 unsigned 16-bit sine samples spanning from 0° to 90°, inclusively (0.0439° resolution).

A small amount of assembly code can mirror and flip the sine table samples to create a full-cycle sine/cosine lookup routine which has 13-bit angle resolution, 17-bit sample resolution, and perfectly-centered minimum, maximum, and zero points, so that full-cycle integration always yields zero (critical for correlators):

```
'
' Get sine/cosine
'
'      quadrant:  1             2             3             4
'         angle:  $0000..$07FF  $0800..$0FFF  $1000..$17FF  $1800..$1FFF
'   table index:  $0000..$07FF  $0800..$0001  $0000..$07FF  $0800..$0001
'        mirror:  +offset       -offset       +offset       -offset
'          flip:  +sample       +sample       -sample       -sample
'
' on entry: sin[12..0] holds angle (0° to just under 360°)
' on exit:  sin holds signed value ranging from $0000FFFF ('1') to $FFFF0001 ('-1')
'
getcos          add     sin,sin_90              'for cosine, add 90°
getsin          test    sin,sin_90      wc      'get quadrant 2|4 into c
                test    sin_sin_180     wz      'get quadrant 3|4 into nz
                negc    sin,sin                 'if quadrant 2|4, negate offset
                or      sin,sin_table           'or in sin table address >> 1
                shl     sin,#1                  'shift left to get final word address
                rdword  sin,sin                 'read word sample from $E000 to $F000
                negnz   sin,sin                 'if quadrant 3|4, negate sample
getsin_ret
getcos_ret      ret                             '39..54 clocks
                                                '(variance is due to HUB sync on RDWORD)


sin_90          long    $0800
sin_180         long    $1000
sin_table       long    $E000 >> 1              'sine table base shifted right

sin             long    0
```

As with the log and anti-log tables, linear interpolation could be applied to the sine table to achieve higher resolution.

Booter and Interpreter ($F002-$FFFF)

The last part of ROM contains the booter and the interpreter. These are assembly language programs which are critical to the operation of the chip.

The booter program runs automatically on reset and is responsible for booting the chip. The booter begins its job by waiting briefly to see if a host is attempting to connect serially on P31 and P30. If a host is present, it can receive a high-level program into RAM. It may then program that data into an external EEPROM and/or launch the interpreter to execute it. If no serial host is present, the booter will attempt to load a high-level program into RAM from an external EEPROM on P29 and P28. If successful, the interpreter will be

launched to execute it. If no EEPROM is present, or the high-level program was corrupted, the booter will shut down the chip to conserve power.

The interpreter program's job is to execute high-level programs. It is initially launched by the booter, in the same COG that the booter was running in. As the interpreter executes a high-level program, it may be directed to re-launch itself into other COGs, so that other high-level programs can run concurrently.

The Global CLK Register

The global CLK register controls the chip's master clock source. It is writable from the high-level 'CLKSET(value)' instruction or the assembly language 'CLKSET D' instruction. Whenever the CLK register is written, a global delay of ~100us will occur as the clock-switchover circuit transitions from potentially one clock source to another.

Whenever this register is changed, a copy of the value written should be placed in BYTE[4]. Also, LONG[0] should be updated with the resulting master clock frequency. This is done so that objects which reference these data will have current information upon which to base their timing calculations.

The CLK register is composed as follows:

Bit	7	6	5	4	3	2	1	0
Name	**RESET**	**PLLENA**	**OSCENA**	**OSCM1**	**OSCM0**	**CLKSEL2**	**CLKSEL1**	**CLKSEL0**

RESET	Effect
0	Always write '0' here unless you intend to reset the chip.
1	Same as a hardware reset – reboots the chip.

PLLENA	Effect
0	Disables the PLL circuit.
1	Enables the PLL circuit. The PLL internally multiplies the XIN pin frequency by 16. OSCENA must be '1' to propagate the XIN signal to the PLL. The PLL's internal frequency must be kept within 64MHz to 128MHz – this translates to an XIN frequency range of 4.00MHz to 8.00MHz. Allow 100µs for the PLL to stabilize before switching to one of its outputs via the CLKSEL bits. Once the OSC and PLL circuits are enabled and stabilized, you can switch freely among all clock sources by changing the CLKSEL bits.

OSCENA	Effect
0	Disables the OSC circuit.
1	Enables the OSC circuit so that a clock signal can be input to XIN, or so that XIN and XOUT can function together as a feedback oscillator. The OSCM bits select the operating mode of the OSC circuit. Note that no external resistors or capacitors are required for crystals and resonators. Allow a crystal or resonator 10ms to stabilize before switching to an OSC or PLL output via the CLKSEL bits. When enabling the OSC circuit, the PLL may be enabled at the same time so that they can share the stabilization period.

OSCM1	**OSCM0**	XOUT Resistance	XIN/XOUT Capacitance	Frequency Range
0	0	Infinite	6pF (pad only)	DC to 80MHz Input
0	1	2000 Ohms	36pF	4MHz to 16MHz Crystal/Resonator
1	0	1000 Ohms	26pF	8MHz to 32MHz Crystal/Resonator
1	1	500 Ohms	16pF	20MHz to 60MHz Crystal/Resonator

CLKSEL2	**CLKSEL1**	**CLKSEL0**	Master Clock	Source	Notes
0	0	0	~12MHz	Internal	No external parts. May range from 8MHz to 20MHz.
0	0	1	~20KHz	Internal	Very low power. May range from 13KHz to 33KHz.
0	1	0	XIN	OSC	OSCENA must be '1'.
0	1	1	XIN x 1	OSC+PLL	OSCENA and PLLENA must be '1'.
1	0	0	XIN x 2	OSC+PLL	OSCENA and PLLENA must be '1'.
1	0	1	XIN x 4	OSC+PLL	OSCENA and PLLENA must be '1'.
1	1	0	XIN x 8	OSC+PLL	OSCENA and PLLENA must be '1'.
1	1	1	XIN x 16	OSC+PLL	OSCENA and PLLENA must be '1'.