IV.2.1 IMAGE GENERATION CONTROLLER FUNCTIONAL DESCRIPTION

1. OVERVIEW

The IGC is implemented as a fully synchronous monolithic device using the Hewlett-Packard CMOS14B 0.7 micron CMOS process. The 11.025 x 11.025 mm die is packaged in a 352-pin TypeII ball-grid-array (BGA) package. Nominal clock rate is 100 Mhz (input port at 125 MHz).

The heart of the IGC is a simple microcode sequencer. This sequencer generates cycle-by-cycle micro-instructions; it also controls a separate address generator and a coefficient serializer. In the EIGC, the micro-instruction outputs drive the ALU micro-instruction inputs to the EMCs, the address outputs drive the pixel-memory address inputs to the EMCs, and the coefficient serializer outputs drive the inputs to the linear expression evaluators on the EMCs. In the TIGC, the micro-instruction outputs drive the micro-instruction inputs to the TASICs, the address outputs drive the module select inputs to the TASICs, and the coefficient serializer outputs are unused.

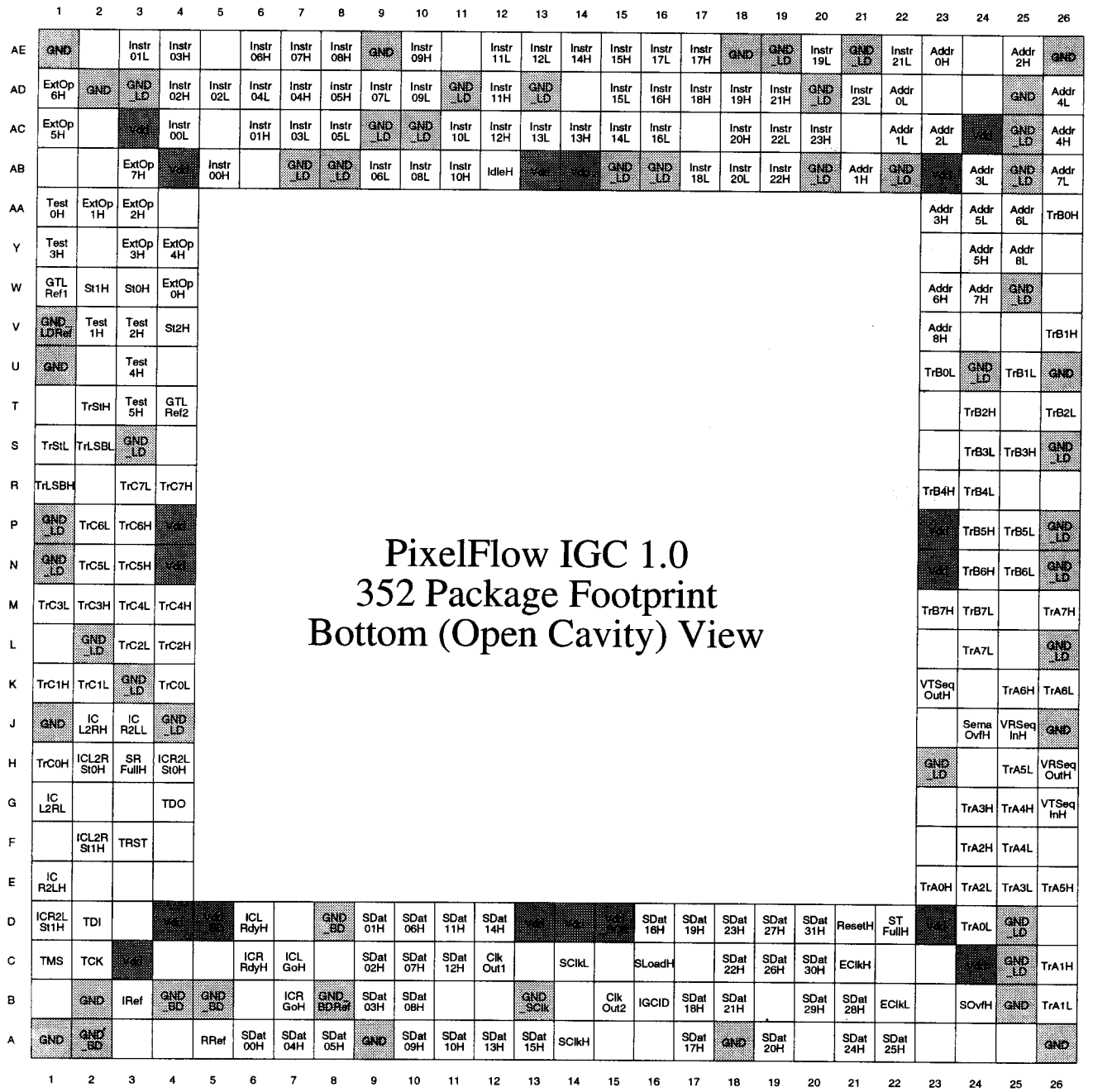
The 32-bit input ports of the two IGCs are connected together, and they accept a single stream of commands; an opcode bit specifies the intended IGC for each command, and a personality pin distinguishes the two IGCs. Commands in the input stream are variable-length in format; they are parsed into a common format by a Stream Parser. Next, they are deposited into one of a parallel pair of FIFOs, one for normal rendering commands and the other for commands to initiate image composition operations (which are executed asynchronously from normal commands). A set of semaphores allows interlocking of command sequences between the two IGCs and between the two FIFOs on each IGC.

Finally, instructions pass from the FIFOs into the sequencer, where they are executed. The sequencer contains 1024 words of 64-bit wide microcode memory. Several loop counters, initialized from fields in the instruction opcode, allow variable length loops. Address outputs are generated by a set of 3 address counters (plus a refresh counter), which also are initialized from fields in the instruction opcode and controlled by the microcode sequencer. The coefficient serializer consists of a set of shifters which convert the various coefficient formats supplied with a command into a byte-serial 2's-complement fixed-point representation; these coefficient shifters also are controlled by the microcode sequencer. The Image Generation Controller (IGC) is a custom VLSI controller chip for the PixelFlow Graphics Engine; two IGCs, loaded with different firmware, are used to control the rasterizer core on each PixelFlow board. The EMC Control IGC (EIGC) supervises the byte-serial, SIMD execution of instructions by the array of processing-elements contained on the enhanced memory chips (EMC's) of the rasterizer core; it directly generates all EMC inputs, except for the power, clock, and reference signals, the control signals and data for the local port, and the data for the image composition port. The TASIC Control IGC (TIGC) controls the texture/frame buffer memory and the texture-ASICs (TASICs), which are used to connect the EMCs to this external memory. Both IGCs also control other

miscellaneous circuitry in the rasterizer core

II. FUNCTIONAL SPECIFICATION

The IGC pinout is shown in Figure 1:



The following table summarizes the IGC pins and their functions.

SIGNAL NAME	#	LEVELS	TIMING	DESCRIPTION
GND	**29**	—	—	Ground pins for core circuitry, input pads, LVTTL output pads, EClk buffer, and ESD protection.
Vdd	**29**	—	—	Power supply for core circuitry, input pads, LVTTL output pads, EClk buffer, and ESD protection. Nominally 3.3 volts.
GND_SClk **Vdd_SClk**	**2**	—	—	Isolated power and ground for the SClk clock-aligner. Nominally same as Vdd level (3.3 volts).
GND_BD	**4**	—	—	Signal returns for bi-directional signalling pads (**IC[LR][Rdy,Go]H**).
Vdd_BD	**1**	—	—	Isolated power for the bi-directional signalling pads. Nominally same as Vdd level (3.3 volts).
IRef	**1**	—	—	Current reference signal for bi-directional signalling pads. Nominally 5 milli-amps.
Rref **GND_BDRRef**	**2**	—	—	Reference resistor for the bi-directional signalling pads. Nominally 50 ohms.
GND_LDRef	**1**	—	—	Signal return for GTL reference signal.
GTLRef{1:2}	**2**	—	—	GTL reference signals.
GND_LD	**31**	—	—	Signal returns for GTL signalling pads.
TCK,TDI,TMS **TRST,TDO**	**5**	LVTTL	TCK	JEDEC standard JTAG boundary-scan port.
EClkH **EClkL**	**2**	differential PECL	defines EClk domain	Input port clock, differential pair. **ResetH**, **SLoadH**, and **SDat[0:31]H** inputs must be synchronous with this clock. Nominally 125 MHz.
SClkH **SClkL**	**2**	differential custom (see specs)	defines SClk domain	Salphasic clock, differential pair. Other inputs and all chip outputs are synchronous with this clock. Nominal frequency is 100 MHz.
IGCID	**1**	LVTTL	static	Personality pin, to distinguish between the two IGCs. Normally set to logic-zero on the EIGC, set to logic-one on the TIGC.
ResetH	**1**	LVTTL	EClk input	Reset signal, asserted for two or more **EClk** cycles to initialize device.
SLoadH	**1**	LVTTL	EClk input	Data strobe for input data on **SDat{0:31}H** inputs.
SDat{00:31}H	**32**	LVTTL	EClk input	Input data pins, for command input. Ignored if **SLoadH** = 0. Bit 00 is LSB.
St{0:2}H	**3**	LVTTL	SClk input	External condition codes, testable as branch conditions in microcode sequencer.

Test{0:5}H	**6**	LVTTL	SClk input	Test-mode select. Multiplexeschipinternal state onto **ExtOp{0:7}H** outputs. Set low for normal operation.
VRSeqInH **VTSeqInH**	**2**	LVTTL	SClk input	Semaphore 'V' signals for RFIFO and T FIFO, from other IGC.
ICLRdyH **ICRRdyH**	**2**	current mode	SClk bi-dir'al	Bi-directional *READY* chain for image-compostion controller.
ICLGoH **ICRGoH**	**2**	current mode	SClk bi-dir'al	Bi-directional *GO* chain for image-compostion controller.
ClkOut1 **ClkOut2**	**2**	LVTTL	on-chip SClk	**SClk** outputs, for testing, and driving other parts.
SRFullH **STFullH**	**2**	LVTTL	SClk output	Almost-full signals for input FIFOs.
IdleH	**1**	LVTTL	SClk output	"Idle" signal; asserted when chip quiescent.
ICL2R{HL} **ICR2L{HL}**	**4**	GTL-pair	SClk output	GTL-pair "run" outputs: strobe image-composition "left-to-right" and "right-to-left" paths.
ICL2RSt{0:1}H **ICR2LSt{0:1}H**	**4**	LVTTL	SClk output	Two-bit status code for image-composition L2R and R2L paths.
VRSeqOutH **VTSeqOutH**	**2**	LVTTL	SClk output	Semaphore 'V' signals for RFIFO and T FIFO on other IGC.
SemaOvfH	**1**	LVTTL	SClk output	Semaphore overflow signal; logical-OR of overflows from 4 semaphore counters.
SOvfH	**1**	LVTTL	SClk output	Overflow signal for FIFOs; logical-OR of overflows from RFIFO and TFIFO.
Instr{00:23}H **Instr{00:23}L**	**48**	GTL-pair	SClk output	GTL-pair sequencer micro-instruction outputs.
ExtOp{0:7}H	**8**	LVTTL	SClk output	External-operation outputs; also used with **Test{0:5}H** to view chip internal state.
Addr{0:8}{HL}	**18**	GTL-pair	SClk output	GTL-pair address outputs.
TrSt{HL}	**2**	GTL-pair	SClk output	GTL-pair LEE data strobe.
TrLSB{HL}	**2**	GTL-pair	SClk output	GTL-pair LEE LSByte indicator.
TrA{0:7}{HL}	**16**	GTL-pair	SClk output	GTL-pair A-coefficient byte-stream (bit 0 is LSB).
TrB{0:7}{HL}	**16**	GTL-pair	SClk output	GTL-pair B-coefficient byte-stream.
TrC{0:7}{HL}	**16**	GTL-pair	SClk output	GTL-pair C-coefficient byte-stream.

IV.3.2—2 External Interface Specifications

Operating Conditions					
Parameter		**Min**	**Nom**	**Max**	**Unit**
Vdd	Power supply voltage	3.0	3.3	3.6	Volts
VIH	Logic-HI input voltage, LVTTL inputs		0.4		Volts
VIL	Logic-LO Input voltage, LVTTL inputs		2.8		Volts
	Common-mode voltage, EClk{HL} inputs	1.2		Vdd - 0.9	Volts
	Differential signal peak-peak, EClk{HL} inputs	1.0		5.0	Volts
	Common-mode voltage, SClk{HL} inputs	2.2			Volts[1]
	Differential signal peak-peak, SClk{HL} inputs	0.8			Volts[1]
TA	Ambient air temperature				°C
TJ	Junction temperature				°C

[1] With clock-aligner enabled; with clock-aligner in bypass mode, input levels must be LVTTL.

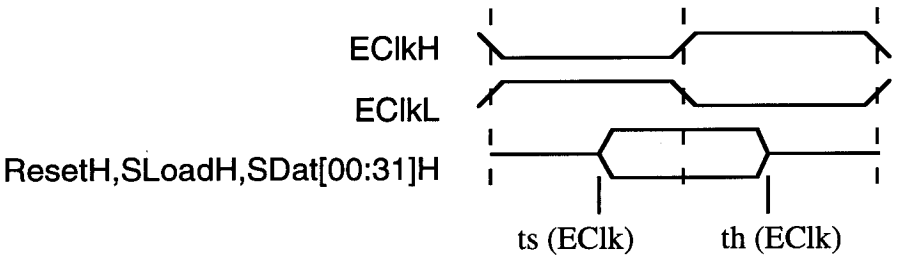
DC Electrical Characteristics (Vdd = 3.3 volts)						
Parameter		**Conditions**	**Min**	**Nom**	**Max**	**Unit**
IL	Input leakage current, all inputs except TDI, TMS, TRST	0 ≤ Vin ≤ Vdd			1	µA[1]
PD	Power dissipation	SClk,EClk = 100 MHz		2	3	Watt

[1] TDI, TMS, and TRST have internal pullup resistors to Vdd, which source up to 150 µa at Vin = 0v.

Timing Requirements					
Parameter		**Min**	**Nom**	**Min**	**Unit**
tP (EClk)	Clock period (EClk)	8			ns
tPH (EClk)	Clock HI period (EClk)	3.5			ns
tPL (EClk)	Clock LO period (EClk)	3.5			ns
tP (SClk)	Clock period (SClk)	10		1.5*EClk	ns[1]
tPH (SClk)	Clock HI period (SClk)	4			ns
tPL (SClk)	Clock LO period (SClk)	4			ns
ts (TCK)	Setup time, TDI and TMS		10		ns[2]
th (TCK)	Hold time, "		10		ns[2]
ts (EClk)	Setup time, ResetH, SLoadH, SDat[00:31]H	0.5			ns[3]
th (EClk)	Hold time, "	2.5			ns[3]
ts (SClk)	Setup time, St{0:2}H, Test{0:5}H,V[RT]SeqInH	2.0			ns[4]

th (SClk)	Hold time,	"	2.5			ns4

[1] In other words, EClk frequency can be no higher than 1.5 times SClk frequency (ask Eyles for exceptions).
[2] Referenced to rising edge of TCK.
[3] See figure:



[4] See figure:

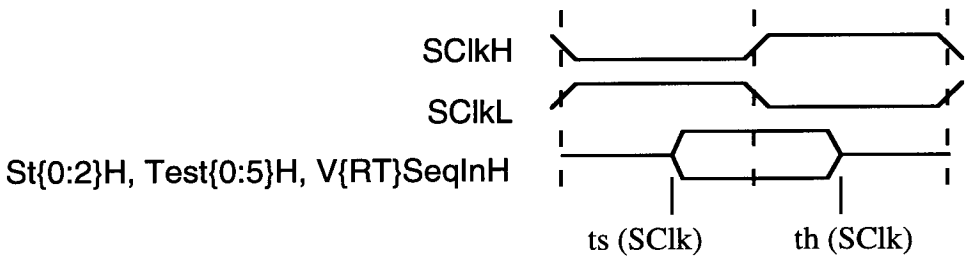


Table values are with clock-aligner enabled.
Add X ns to max times with clock-aligner in bypass mode.

Timing Characteristics

Parameter		Conditions	Min	Nom	Max	Unit
tp(TTL)	Propagation time, LVTTL outputs	No Load				ns[1]
tp(TTL)	Propagation time, LVTTL outputs	CL=50Ω to 1.4v				ns[1]
tp(GTL)	Propagation time, GTL outputs	No Load				ns[1]
tp(GTL)	Propagation time, GTL outputs	CL=50Ω to 1.2v				ns[1]
tp(TDO)	Propagation time, TDO output					ns[3]
tp(ClkOut)	Propagation time, ClkOut1 and ClkOut2	CL=1MΩ ‖ 9pf	1.0	1.5	2.0	ns[2]
tp(ClkOut)	Propagation time, ClkOut1 and ClkOut2	100Ω toGND ‖ 25pf to 1.4 v	1.3	1.8	2.4	ns[2]

[1] With SClk clock-aligner enabled. Add X ns to max values with clock-aligner in bypass mode.

[2] Measured from positive-going crossing of SClkH/SClkL to rising edge of ClkOut{1:2}, with clock-aligner enabled. Add X ns to max values with clock-aligner in bypass mode.

[3] Referenced to falling edge of TCK.

Timing of External Interface Signals

The IGC is controlled by two clocks, which are asynchronous with respect to each other. One is defined by the differential input pair **SClkH/SClkL**, and the other is defined by the differential input pair **EClkH/EClkL**. For each clock, the "primary edge of **Clk**" is defined as the point at which **ClkH** goes high with respect to **ClkL**. All outputs are synchronous with respect to **SClkH/SClkL**, with propagation referenced to the primary edge of **SClk**. Chip inputs are synchronous with respect to one of the two clock regimes, as specified above; setup and hold times are specified with respect to the primary edge of the appropriate clock.

Input Protocol

The IGC accepts a stream of 32-bit instruction words. An instruction word is written by asserting **SLoadH** with the desired data present on **SDat[0:31]H**. IGC commands are variable in length, consisting of 1 to 8 instruction words (discussed below). Each command is parsed and written into one of the two input FIFOs (the RFIFO or the TFIFO); each FIFO entry consists of a single command (regardless of its size), as opposed to one instruction word. Each FIFO has a programmable almost-full flag, the **SRFullH** and **STFullH** outputs, for the RFIFO and TFIFO resepectively. When either of these goes high, this means that the corresponding FIFO has reached its "fullness limit" (set using a special command described below). The device feeding commands to the IGC must monitor **SRFullH** and **STFullH**, to avoid oveflowing either FIFO. There is a pipeline delay between the input port and **S{RT}FullH**, as shown in Figure 2.

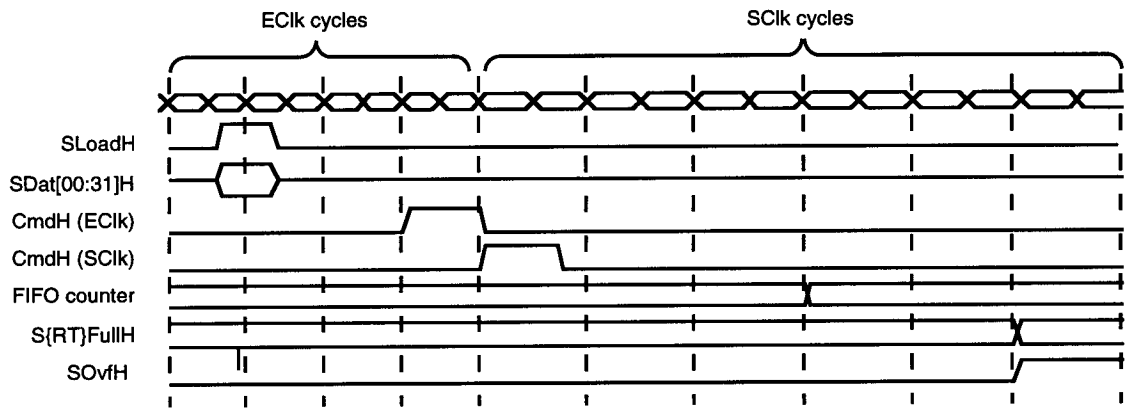


Figure 2: Timing of Full and Overflow Signals

In this figure, the **SLoadH** pulse represents writing the single instruction word of a one-word command (see below). The FIFO write-port is in the **EClk** domain, and the pulse **CmdH(EClk)** represents the command being written into the FIFO. This write command is synchronized into the **SClk** domain andbecomes **CmdH(SClk)**; there is some uncertainly in this synchronization, depending upon the relative timing of the **EClk** and **SClk** edges. Three **SClk** cycles later, the FIFO counter changes value, and two cycles after that the **S{RT}FullH** flags may change and the **SOvfH** oveflow flag may go high.

In the worst-case, a continuous stream of one-word commands can be written into the input port. Thus, by the time the almost-full flag goes high, there may have been 8 additional commands written into the input port, perhaps more if the **EClk** rate is higher than the **SClk** rate. By the time the inputting device reacts to the high-going transition of the almost-full flag and ceases inputting commands into the IGC, even more additional commands may have been written. Thus, the fullness-limits must be set considerably below the full size of each FIFO, to avoid overflows.

An overflow-flag, **SOvfH**, provides some indication of a FIFO overflow; once it goes high, **SOvfH** remains high until **ResetH** is asserted. Unfortunately, this overflow-flag is unreliable: overflowing a FIFO does not necessarily result in **SOvfH** going high, again due to the pipeline delay. For example, if the FIFO is full, and a new command is written into the FIFO, this oveflows the FIFO and corrupts data; but if a command is read from the FIFO during the pipeline delay before the FIFO counter is incremented, then **SOvfH** is never asserted because the counter never oveflows.

Command Formats

Commands consist of an opcode followed by optional ABC coefficients. The opcode and coefficients may be 32-bit, or 64-bit written as two separate words. The format of the opcode is shown in Figure 3.

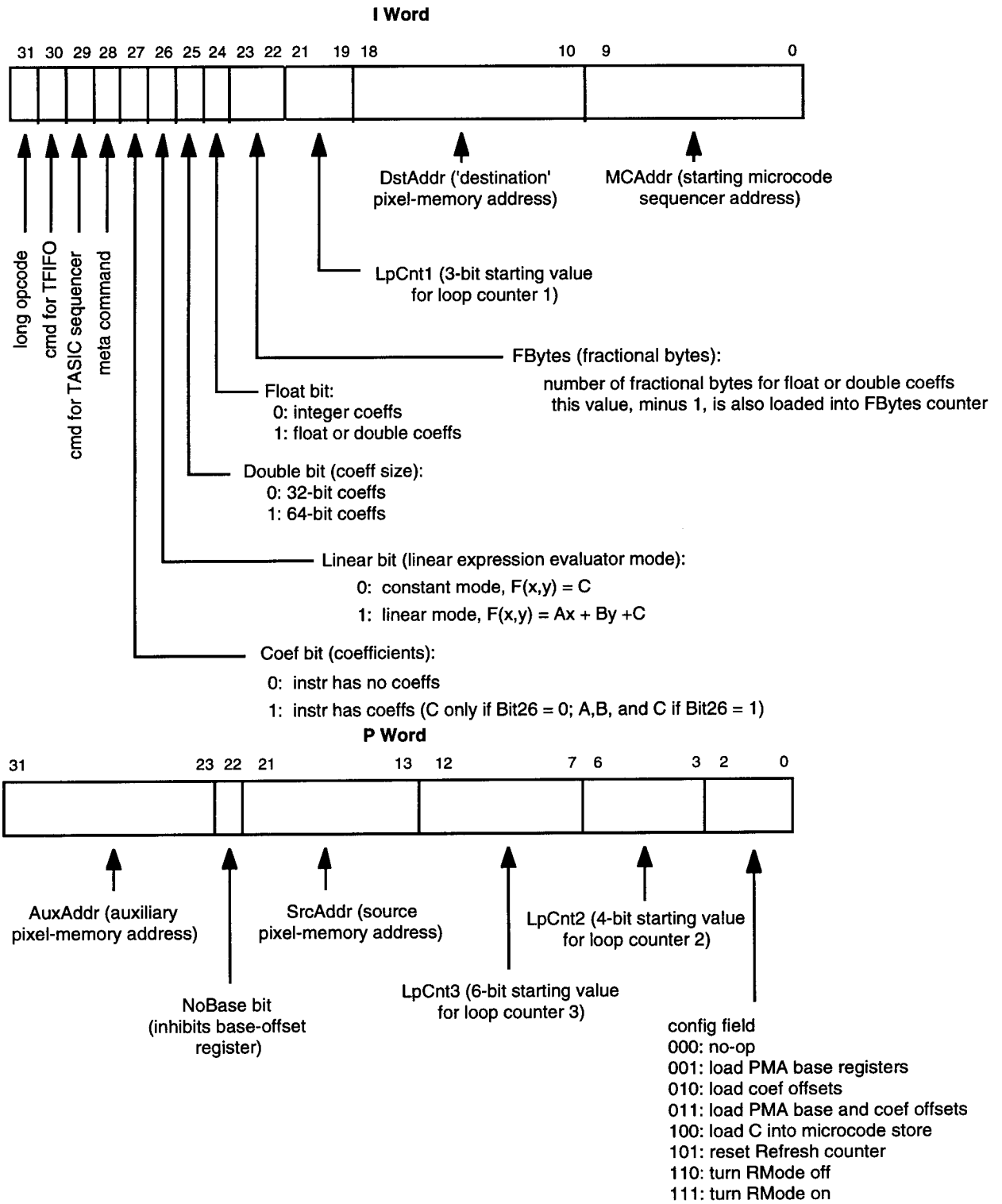


Figure 3: Opcode Formats

The first word of an instruction is always the *I-Word*, the low-order 32-bits of the opcode. The I-Word determines the number of additional instruction words, and their meaning, as described below.

Bits 0-9 (the *MCAddr* field) define the entry point in the microcode sequencer for execution of the instruction.

Bits 10-18 (the *DstAddr* field) specify the *destination* pixel-memory operand

Bits 19-21 (the *LpCntl* field) specify the 3-bit initial count value for Loop Counter 1.

Bits 22-23 (the *FBytes* field) specify the number of fractional bytes (0 to 3) when floating-point or fixed-point LEE coefficients are converted to fixed-point form for the EMCs. This field is also used as the initial count value for the 2-bit FBytes counter, but its value is decremented as it is loaded into the counter; thus the starting count value is the FBytes field minus 1, modulo 3.

Bit 24 (the *Float* bit) is set to 1 to indicate the the LEE coefficients are floating-point, to 0 if they are integer.

Bit 25 (the *Double* bit) is set to 1 to indicate the the LEE coefficients are double-length, that is, if integers they are 64-bits rather than 32-bits, and if floating-point they are 64-bit double-precision rather than 32-bit single-precision.

Bit 26 (the *Linear* bit) is set to 0 to indicate the instruction has only a C coefficient and the LEE uses constant mode (F(x,y) = C), or 1 to indicate the instruction has ABC coefficients and the LEE uses linear mode (F(x,y) = Ax + By + C

Bit 27 (the *Coefs* bit) is set to 1 to indicate the instruction has coefficients. The type of coefficients and their use by the LEE is defined by the *Linear*, *Double*, *Float*, and *FBytes* fields. If this bit is 0, bits 19-26 are interpreted as a one-byte immediate operand which can be passed through the LEE.

Bit 28 (the *Meta* bit) is set to 1 to indicate this is a meta-command. If this bit is set, the other bits of the opcode are interpreted in an alternate fashion. See "**Meta Commands**" below.

Bit 29 (the *ETSel* bit) is set to 0 to indicate that the instruction is for the EIGC (assumes input IGCID is logic-zero), to 1 to indicate that the instruction is for the TIGC (asumes input IGCID is logic-one)

Bit 30 (the *TCmd* bit) is set to 0 to indicate that the instruction should be placed into the RFIFO, or to 1 to indicate that the instruction should placed into the TFIFO.

Bit 31 (the *Long* bit) is set to 1 to indicate the opcode is 64-bits rather than 32-bits.

The number and ordering of additional words in the instruction are defined by several of the fields in the low-order word of the opcode. When set, the *Long* bit (bit 31) indicates that the

next word is the high-order word of the opcode; the *Coef* bit (bit 27 = 1) indicates that the instruction has instruction has coefficients; the *Linear* bit (bit 26 = 1) indicates that the instruction has three coefficients (ABC) rather than one (just C); the *Double*bit (bit 25 = 1) indicates that the coefficients are double-size (64 bits) instead of single (32-bits). This information can be summarized as follows:

I-Word Bit				Additional Command Words (Ordered)	Total Words
31	27	26	25		
0	0	X	X	—	1
1	0	X	X	P	2
0	1	0	0	C	2
1	1	0	0	P, C	3
0	1	0	1	C0, C1	3
1	1	0	1	P, C0, C1	4
0	1	1	0	A, B, C	4
1	1	1	0	P, A, B, C	5
0	1	1	1	A0, A1, B0, B1, C0, C1	7
1	1	1	1	P, A0, A1, B0, B1, C0, C1	8

In this table, *P* refers to the upper 32-bits (i.e.: the P-word) of the opcode; *A*, *B*, and *C* refer to the single 32-bit word of a 32-bit integer or single-precision float coefficient; *A0*, *B0*, and *C0* refer to the first 32-bit word of a 64-bit integer or double-precision float coefficient; and *A1*, *B1*, and *C1* refer to the second word of a 64-bit integer or double-precision float coefficient. If the *Endian* bit in the Interface Control Register (see below) is 0, *A0*, *B0*, or *C0* should be the low order 32-bit word of the 64-bit coefficient, and *A1*, *B1*, or *C1* should be the high-order 32-bit word of the coefficient. If the *Endian* bit is 1, then the ordering of the high-order word and low-order word are reversed.

If the *Long* bit is set, the second word (the P-Word) of the instruction contains the supplementary 32-bits of the opcode; if not, the upper 32-bits of the opcode are all set to 0. The format of the high-order word of the opcode is also shown in Fig xxx. The bits are interpreted as follows:

Bits 0-2 (the *Cfg* field), is used to perform configuration functions in the IGC. These bits cause the following actions:

P-Word Bit			Action
2	1	0	
0	0	0	no-operation
0	1	1	load address base-offset registers
0	1	0	write A and B coefficients to Serializer offset registers
0	1	1	load address base-offset registers *and* Serializer offset registers
1	0	0	write C coefficient to microcode store address specified by *MCAddr*
1	0	1	clear address refresh counter
1	1	0	cause Sequencer to exit RMode
1	1	1	cause Sequencer to enter RMode

Bits 3-6 (the *LpCnt2* field) specify the 4-bit initial count value for Loop Counter 2.

Bits 7-12 (the *LpCnt3* field) specify the 6-bit initial count value for Loop Counter 3.

Bits 13-21 (the *SrcAddr* field), specify the *source* pixel-memory operand.

Bit 22 (the *NoBase* bit) inhibits applying the base-offset register to the pixel-memory operands (see below).

Bits 23-31 (the *AuxAddr* field), specify the *auxiliary* pixel-memory operand.

Meta commands. When IWord bit 28 (the *Meta* bit) is set, the command is a special type of command called a *meta* command; for "meta" commands, some of the Iword and Pword opcode bits have an alternate function, as follows:

I-Word Bit	Action When Meta Command is Executed (I-Word bit 28 must also be set)
26	This command is a special type of *meta* command called an *ignore* command. *Ignore* commands are not loaded into either FIFO; they are used to pad the instruction stream and to load the Interface Control register.
25	This modifies the meaning of some of the other *meta* command opcode bits. When bit 26 is set, it enables loading of the Interface Control Register, used for setting the Endian bit and the FIFO fullness limits. For semaphore 'P' commands, it causes control to defer to the other FIFO if the'P' command causes blocking of the FIFO.
24	Causes loading of the alive and ends registers (for the Image Composition ready/go controller.
18	Causes arming of the R2L path in the ready/gp controller.
17	Causes arming of the L2R path in the ready/gp controller.
16	Causes the FIFO to block until any current Image Composition transfer operation is completed.
15	Causes a 'V' operation (increments the semaphore counter) on the sequencer-blocking semaphore for the corresponding FIFO in the other IGC. This asserts a one-cycle pulse on the **VRSeqOutH** or **VTSeqOutH** output pin.
14	Causes a 'V' operation (increments the semaphore counter) on the fifo-blocking semaphore for the other FIFO.
13	Causes a 'P' operation (blocks until the semaphore counter is non-zero and then decrements it) on the sequencer-blocking semaphore.
12	Causes a 'P' operation (blocks until the semaphore counter is non-zero and then decrements it) on the fifo-blocking.
11	Causes the FIFO to become the *favorite* FIFO.
10	Causes the other FIFO to become the *favorite* FIFO.

P-Word Bit	Action When Meta Command is Executed (I-Word bit 28, and I-Word bit 17, 18, or 24 must also be set)
7	This value is loaded into the *alive* register if I-Word bit 24 is also set.

8	This value is loaded into the *left end* register if I-Word bit 24 is also set.
9	This value is loaded into the *right end* register if I-Word bit 24 is also set.
13	This value is loaded into the L2R path *first* register if I-Word bit 17 is also set.
14	This value is loaded into the L2R path *last* register if I-Word bit 17 is also set.
15-19	This value is the byte-count for L2R path transfers (loaded into counter if Iword bit 17 also is 1); bit 15 is LSB.
23	This value is loaded into the R2L path *first* register if I-Word bit 18 is also set.
24	This value is loaded into the R2L path *last* register if I-Word bit 18 is also set.
25-29	This value is the byte-count for R2L path transfers (loaded into counter if Iword bit 18 also is 1); bit 25 is LSB.

Coefficients. The opcode word(s) are followed by optional coefficient words as shown above in Table X.

Use of these coefficients by the LEE, and their data types, are specified by the *Coef*, *Linear*, *Double*, and *Float* bits in the IWord, as shown in the following table:

Bit 27	Bit 26	Bit 25	Bit 24	F(x,y)=	Coefficients
0	X	X	X	C	C = byte defined by bits 26-19
1	0	0	0	C	C = 32-bit integer
1	0	0	1	C	C = single-precision float
1	0	1	0	C	C = 64-bit integer
1	0	1	1	C	C = double-precision float
1	1	0	0	Ax + By + C	ABC = 32-bit integers
1	1	0	1	Ax + By + C	ABC = single-precision floats
1	1	1	0	Ax + By + C	ABC = 64-bit integers
1	1	1	1	Ax + By + C	ABC = double-precision floats

When bit 26 is set, the LEE computes the full bi-linear expression; if bit 25 is zero, the LEE's compute just the constant portion, C.

Within the IGC, the 32-bit integer type is converted to 64-bit integer type by sign-extension, and the single-precision float type is converted to a double-precision float using a standard single-to-double conversion. Coefficients are converted to byte-serial fixed-point numbers with adjustable numbers of fractional bytes, before being use to compute linear expressions (see below).

Single-precision floats should conform to the standard IEEE format, in which bits 0-23 represent the fractional part of the mantissa with understood 1 to the left of the radix point, bits 23-30 represent the exponent in excess-127 form, and bit 31 is the sign-bit (the representation is sign/magnitude). Double-precision floats should conform to the standard IEEE format, in which bits 0-51 represent the fractional part of the mantissa with understood 1 to the left of the radix point, bits 52-63 represent the exponent in excess-1023 form, and bit 63 is the sign-bit (the representation is sign/magnitude).The IEEE standards

specify several *exceptions*, with exponent fields either all 0's or all 1's; the exceptions in which the exponent field is all zeroes (zero, and denormalized numbers) are treated as zero, and the exceptions in which the exponent field is all ones (NANs and infinities) are also treated as zero. Because the coefficients are converted to fixed-point numbers, precision is often lost . If the exponent is less than -8*fbytes, the fixed-point equivalent is zero. If the exponent is very large, the significant bits of the mantissa may be shifted so far to the left that the coefficient is effectively zero. The precision lost is determined by the magnitude of the coefficient and the setting for the "number of fractional bytes" (FBytes field of the low opcode). The floating-point value is *truncated*, not *rounded*, when the conversion is made, and the truncation is towards zero. Further, although FBytes fractional bit of the coefficients are generated, the corresponding fractional bytes of the LEE results are not available. Only the integer portion of the LEE results are available at the pixel-ALU and pixel-memory. Thus, two truncations occur when floating-point coefficients are supplied to the EMC's, which are essentially integer devices: (1) the coefficients are truncated to fixed-point, (2) the LEE result is truncated downwards (not towards zero) to an integer.

Semaphores

Both have a digital filter which ignores consecutive assertions; that is, the signal must have been low on the previous cycle for an assertion on the current cycle to affect the semaphore.

Loading the Interface Control Register

After reset, the IGCs input interface is in an undefined state. The "fullness" limits, that is, the comparison values for the "almost-full" flag outputs **SRFullH** and **STFullH**, are undefined. Each flag is asserted only if its respective FIFO contains a number of commands *greater* than this fullness limit; since the FIFO counters are initialized to zero on reset, the full-flags are guaranteed to be de-asserted. However, since the fullness limits are undefined, entering a single command can potentially cause **SRFullH** or **STFullH** to be asserted, thereby hanging any software or hardware interface that requires the full-flags to be de-asserted before writing additional commands to the IGC. Thus, immediately after reset is over (and **ResetH** is de-asserted), the Interface Control Register should be loaded, using the command:

Bit 29 specifies which IGC the command is for. Bits 28 and 26 specify an "ignore" type command, which is not written into either FIFO, so bit 30 is ignored. Bit 25 enables loading the Interface Control Register (for an "ignore" type command). Bits 7 - 15 specify the "fullness" limit for the TFIFO (bit 7 is LSB) and bits 0 - 6 specify the "fullness" limit for the RFIFO (bit 0 is LSB). Bit 16 specifies the "endian-ness" for 64-bit coefficients: if it is 1, the high-order 32-bit word is the first of the two 32-bit command words of a 64-bit coefficient; it is ignored except for commands containing 64-bit coefficients. The Interface Control Register can be reloaded at any time, regardless of whether the IGC is in **RMode**

or not.

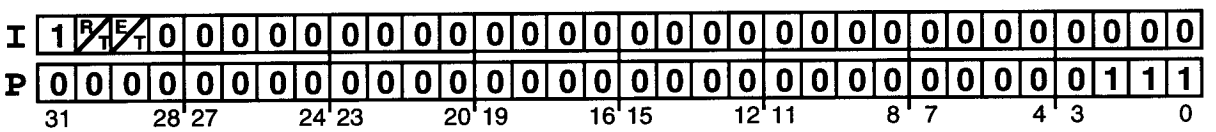
Resetting the IGC, and RMode

The reset signal, **ResetH**, must always be asserted after the IGC is powered-up, or to reset a hung IGC; the reset pulse should be at least two **EClk** cycles long. Reset initializes the Input Interface and resets the RFIFO and TFIFO. Reset also puts the Sequencer into a special mode called *RMode*, used for writing and reading the microcode store.

During RMode, the micro-instruction outputs **Instr{00:23}{HL}** are de-asserted. The address outputs **Addr{0:8}{HL}** cycle through all 512 possible address values; the initial value is undefined after Reset. The external operation outputs **ExtOp{0:7}Hf** also are de-asserted, provided the test-mode inputs **Test{0:5}H** are zero; otherwise, the **ExtOp{0:7}Hf** outputs view internal state (as described below).

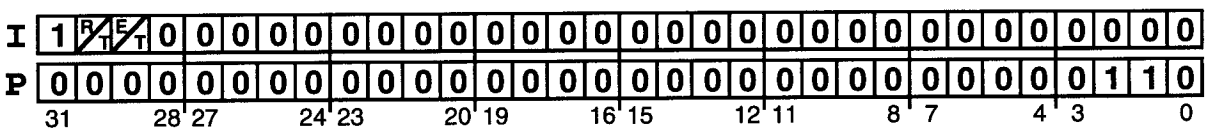
In RMode, the Sequencer always accepts new commands *immediately*,and executes them in a single clock cycle. The microcode sequencer is forced to the starting address of the new command, and is held there until a new command arrives. This behavior allows writing and reading of the Sequencer microcode store, as shown below.

The Sequencer can also enter RMode by executing the command:



where bits 29 and 30 specify which IGC, and which FIFO on that IGC, the command is meant for.

After being loaded with microcode, the Sequencer exits RMode by executing the command:

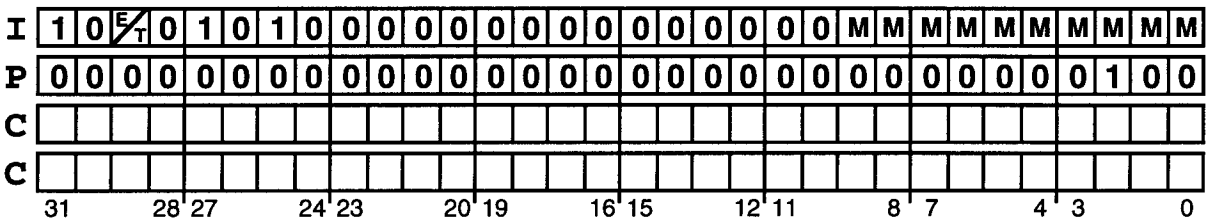


This causes the sequencer to exit RMode and begin executing microcode at address 0.

Loading the Microcode Store

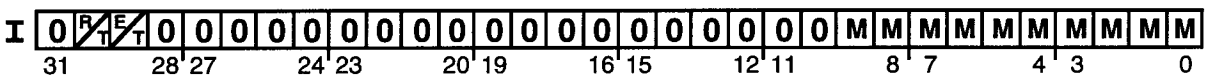
To load microcode into the sequencer, the sequencer is first put into RMode, as described above.

Each microcode location is loaded using a 4-word command:



Bits 0 - 9 of the I-opcode specify the microcode store location to be written. Bits 0 - 2 of the P-Word specify the "write microcode store" code for the *Cfg* field. The two C-coefficient words specify the 64-bit microcode word to be loaded; the ordering of the hi- and lo-order words is defined by the *Endian* bit in the Interface Control Register. This command can only be specified for the RFIFO, since it contains coefficients.

After all desired microcode locations are loaded, the microcode can be read back by sending a command:



Bits 0 - 9 specify the microcode store location to be read. Assuming the sequencer is in RMode, this microcode store location is repeatedly read until a new command is available. The test port, input pins **Test{0:5}H** and output pins **ExtOp{0:7}H**, is used to read the current microcode word (see below).

After microcode is loaded, the Sequencer can be taken out of RMode (see above), and it begins executing microcode at address 0.

Using the Test-Mode Port

For normal operation, the test-mode inputs **Test[5:0]H** are low, and the **ExtOp[7:0]H** outputs are generated directly by bits in the currently executing Sequencer microcode word (or held low while the Sequencer is in RMode). By setting **Test[5:0]**H to other values, the **ExtOp[7:0]H** outputs provide visibility of internal state within the IGC, as shown in Table:

Test{5:0}H						Data Output on ExtOp{7:0}H
5	4	3	2	1	0	
0	0	X	X	X	X	normal operation
0	1	0	d2	d1	d0	diagnostics (see table)
0	1	1	b2	b1	b0	current microcode word
1	0	0	b2	b1	b0	bytes of Opcode in Command Latch
1	0	1	b2	b1	b0	bytes of A-coefficient in Command Latch
1	1	0	b2	b1	b0	bytes of B-coefficient Command Latch
1	1	1	b2	b1	b0	bytes of C-coefficient in Command Latch

The *Command Latch* settings show the command currently residing in the Command

Latch. This is either the currently-executing or last to execute command, if no additional commands are available (**IPHf** is low), or the next command that will be executed after the current command finishes (if **IPHf** is high). For these settings, *b2/b1/b0* specify one of the 8 bytes of the opcode or coefficient command word (*b0* is LSB of byte-number, byte 0 is least-significant byte of command word). Note that: for commands with no P-Word, the upper 32-bits of the Opcode are set to zero; for commands with 32-bit coefficients, the 64-bit coefficient word is either sign-extended for integer coefficients, or double-precision converted for floating-point coefficients; and for a "no coefficients" (or "C-coefficient only") type command, the coefficent words (or the A and B coefficients) are garbage.

The *current microcode word* setting outputs the specified byte of the currently executing microcode word. It is used primarily for reading microcode store (after putting the Sequencer into RMode) or providing a byte-wide window of a sequence of microcode being executing (when not in RMode).

The *diagnostics* settings provide visibility of various control signals within the IGC. These are shown in the following table:

Test{i}H 2	1	0	ExtOp {i}H	SIGNAL NAME	DESCRIPTION	RESET VALUE
0	0	0	0	RRdyHf	asserted if R "ready" latch contains cmd	0
			1	TRdyHf	asserted if T "ready" latch contains cmd	0
			2	REmptyLf	asserted when RFIFO is empty	0
			3	TEmptyLf	asserted when TFIFO is empty	0
			4	DoneLf	asserted for *done* bit of current ucode word	X
			5	IPHf	asserted if Command Latch has command	0
			6	NewHf	asserted on cycle Sequencer starts cmd	0
			7	RModeH	asserted when Sequencer is in RMode	1
0	0	1	0	BStH	asserted if Sequencer test St-input branch	X
			1	BranchH	asserted if Sequencer branch or in RMode	1
			2	MWriteHf	asserted on cycle microcode store loaded	0
			3	ResetHf	latched version of chip input ResetH	0
			4	DoneLf	asserted for *done* bit of current ucode word	X
			5	IPHf	asserted if Command Latch has command	0
			6	NewHf	asserted on cycle Sequencer does new cmd	0
			7	RModeH	asserted when Sequencer is in RMode	1
0	1	0	0	TCFHf	asserted when FBCnt loop counter is 0	X
			1	TC1Hf	asserted when LpCnt1 loop counter is 0	X
			2	TC2Hf	asserted when LpCnt2 loop counter is 0	X
			3	TC3Hf	asserted when LpCnt3 loop counter is 0	X
			4	Cnt2Hf	asserted on cycle LpCnt2 is decremented	X
			5	Cnt3Hf	asserted on cycle LpCnt3 is decremented	X
			6	—	<UNUSED>	0
			7	—	<UNUSED>	0
			0	LoadIFHe	asserted on cycle Interface Ctrl Reg loaded	0
			1	SLoadHe	latched version of chip input SLoadH	0
			2	State2H	three state bits of Stream Parser	0

0	1	1	3	State1H	state machine (bit 0 is LSB)	0
			4	State0H		0
			5	FTCmdHe	asserted cycle command written to TFIFO	0
			6	FRWrtHe	asserted cycle inst-word written to RFIFO	0
			7	FRCmdHe	asserted cycle command written to RFIFO	0
1	0	0	0	RBlockH	RFIFO blocked (if RRdyHf asserted too)	X
			1	TBlockH	TFIFO blocked (if TRdyHf asserted too)	X
			2	FavRHf	asserted if RFIFO is "favorite" FIFO	0
			3	DoMetaHf	asserted cycle that "meta" command done	0
			4	RFifZerHf	asserted if RFIFO fifo-semaphore is 0	1
			5	TFifZerHf	asserted if TFIFO fifo-semaphore is 0	1
			6	RSeqZerHf	asserted if RFIFO seq-semaphore is 0	1
			7	TSeqZerHf	asserted if TFIFO seq-semaphore is 0	1
1	0	1	0	AliveHf	asserted if Image-Comp is "alive"	0
			1	ICPendHf	asserted if Image-Comp transfer "pending"	0
			2	LoadMacHf	asserted on cycle alive and ends loaded	0
			3	L2RRunHf	precursor of ICL2R{HL} GTL outputs	0
			4	L2RArmHf	asserted on cycle L2R path is "armed"	0
			5	R2LArmHf	asserted on cycle R2L path is "armed"	0
			6	L2RZerHf	asserted when L2R counter is zero	X
			7	R2LZerHf	asserted when R2L counter is zero	X
1	1	0	0	AuxHiLf		1
			1	AuxLoLf		1
			2	SrcHiLf		1
			3	SrcLoLf		1
			4	LoadBaseHf	asserted on cycle PMA base-reg is loaded	0
			5	TreeStepHf	asserted on cycle Serializer is "stepped"	X
			6	Mux0Hf		0
			7	Mux1Hf		0
1	1	1	0	REmptyLf	asserted when RFIFO is empty	0
			1	TEmptyLf	asserted when TFIFO is empty	0
			2	RReadDecLf	asserted on RFIFO read cycle	1
			3	TReadDecLf	asserted on TFIFO read cycle	1
			4	RGrey0Hf	2 LSBs of number of RFIFO commands	0
			5	RGrey1Hf	since Reset, in Grey-code (bit 0 is LSB)	0
			6	TGrey0Hf	2 LSBs of number of TFIFO commands	0
			7	TGrey1Hf	since Reset, in Grey-code (bit 0 is LSB)	0

There is a 3 cycle delay bewteen applying inputs to the **Test[5:0]H** inputs and the corresponding effect on the **ExtOp{0:7}H** outputs.

The Idle signal. The output **IdleH** is asserted when both FIFOs and their read latches are empty, no instruction is pending, and the currently-executing Sequencer microcode word is asserting *done*. **IdleH** *may* be asserted if an Image Composition transfer is pending. **IdleH** is valid only the when **Test[0:2]H** inputs are low.

IV.2.2 IMAGE GENERATION CONTROLLER LOGIC DESIGN

The IGC is divided into the following top-level modules:

(1) StrPsr: latches for input interface, stream parser
(2) FIFO: dual buffers for input commands
(3) RTCntl: R/T controller, arbitrates between RFIFO and TFIFO commands, contains semaphores, muxes R and T FIFOs
(4) ICCntl: Image Composition controller, handles ready/go chain and composition control
(5) Cntl: the sequencer, address generators, and coefficient serializers
(6) OutputLatch: output delays, output multiplexer, and output pads

IV.2.2.1 StrPsr Module

StrPsr contains the input pads and single output pad for the instruction input interface, logic for parsing the instruction stream, logic for converting all input opcodes and coefficients into the 64-bit format used within the remainder of the IGC, and logic and latches for controlling writes into the FIFOs and buffering the input comands.

The input interface consists of the following inputs: the instruction data word **SDat[0:31]H**, the instruction write enable signal **SLoadH**, the reset signal **ResetH**, and the IGC identifier **IGCID**; outputs are **SRFullH** and **STFullH**, the almost full signals for the two FIFOs. All inputs are expected to meet setup and hold times referenced to the "rising" edge of the off-chip salphasic clock (defined by **SClkH** and **SClkL**) and therefore to be latchable in the input pads on the falling edge of the on-chip system clock **ClkL**, and the outputs are guaranteed to meet minimum and maximum propagation delays referenced to the salphasic clock and therefore latchable on the rising edge of the off-chip system clock **ClkH**.

SDat[0:31]H, **SLoadH**, and **ResetH** are latched in their input pads. The resulting signals have similar signal names, but with the **-H** suffix replaced by **-He**. **ResetHe** acts as the reset input to the Stream Parser Finite State Machine (**SPFSM**); it is passed to **FIFO**, which synchronizes it into the SClk domain and passes it to **RTCntl**, **ICCntl**, and **Sequencer**. **SLoadHe** and several bits of the data word **SDat[0:31]He** are also inputs

to **SPFSM**.

SDat[0:31]He passes into two unclocked logic blocks, **CnvLo** and **CnvHi**, which perform conversions of the various input data types into double-length (64-bit) integers and floating-point numbers used within the IGC. When **CnvHe** and **HiZHe**are both low, **CnvLo** and **CnvHi** simply pass **SDat[0:31]He** through to **LoDat[0:31]He** and **HiDat[0:31]He**; this mode is used for both words of a double-length (integer or floating-point) input coefficient. When **HiZHe** is high, **SDat[0:31]He** is passed through to **LoDat[0:31]He** and **SDat31He** is passed through to **HiDat[0:31]He** (sign-extension). This mode is used for sign-extending a 32-bit integer integer coefficient to the 64-bit integer format used within the IGC, as well as for converting a 32-bit opcode to 64-bit format; the sign-extension means that for *Long* commands, where the MSB of the low-word of the opcode is 1, the high-word of the opcode will be set to 0xffffff, but this is ok since the high-word will be overwritten by the next input word. When **CnvHe** is high (and **HiZHe** is low), **SDat[0:31]He** is assumed to be a single-precision float and the double-precision equivalent value is passed passed through to **LoDat[0:31]He** and **HiDat[0:31]He**.

The Stream Parser Finite State Machine (**SPFSM**) parses the input instruction stream. It decodes the bits of the opcode which specify which coefficients follow the opcode and what data-type they are, and generates the signals **CnvHe** and **HiZHe** to convert data types as described above, the signals **SaveLoHe** and **SaveHiHe** which disable latching of the converted data **LoDat[0:31]Hf** into the output latches, and the signals **FRWrtHe**, **FRCmdHe**, **FTCmdHe**, and **FIABC[0:1]He**, which control writing into the FIFOs. The function of these control signals is described below in the **FIFO** Module description.

When bits 28 and 26 of opcode are set, this defines a special kind of command for the input interface. If bit x is zero, this is an "ignore" command, meaning that **StrPsr** simply flushes the command; this is normally used for a spacer word to pad to 8-byte address boundaries in the GP. If bit x is one, then the command loads the Interface Specification Register. Bits 0-6 are loaded into the register **RLim[0:6]He**, bits 7-15 are loaded into the register **TLim[0:8]He**, and bit 16 is loaded into the register **EndianHe**. Both types of command will can have a 64-bit opcode and/or coefficient(s) if bit 31 or 27 is set, although they are not used; neither command causes **FRWrtHe**, **FRCmdHe**, or **FTCmdHe** to be asserted, so nothing is loaded into the FIFOs. **RLim[0:6]He** and **TLim[0:8]He** define the "high-water" marks for the RFIFO and TFIFO, respectively, and are passed to **FIFO**. **EndianHe** defines which word of a 64-bit coefficient comes first, and is an input to **SPFSM**.

IV.2.2.2 FIFO Module

This module consists of a single memory system, used to implement the RFIFO and TFIFO first-in/first-outbuffers for instruction input, and associated logic for keeping track of pointers into the memory and full and empty status.

Physically, the memory is arranged as 256 rows by 256 columns. The cells are dual-ported (two pairs of bit-lines and two word-lines, per cell). The memory system has a write-only port and a read-only port. The write port accesses a fourth of a row (a 64-bit word) on each write cycle. The read port accesses an entire 256-bit row on each read cycle.

Logically, the memory consists of the RFIFO, consisting of 128 entries each consisting of four 64-bit words, and the TFIFO, consisting of 512 entries each consisting of a single 64-bit word. Each RFIFO entry uses one row of the memory, and four TFIFO entries use one row of memory.

The four 64-bit words of an RFIFO entry include the opcode (I word) and the three coefficients A, B, and C (the A, B, and C words). Since TFIFO commands cannot include coefficients, a TFIFO entry consists of just the 64-bit opcode. For RFIFO commands which do not have coefficients, or which have only the C coefficient, the words in the RFIFO corresponding to the missing coefficients will not be written; this means that the missing coefficients will be undefined for these commands.

Each write cycle writes one of the four words of an RFIFO entry, or a entire TFIFO entry. Each read cycle reads an entire RFIFO entry, or four separate TFIFO entries.

Each logical FIFO has two pointers for write and read access. These pointers reset to zero on reset, and incremented each time a read of write occurs. The RFIFO pointers simply indicate which row to read or write. The upper 7 bits of the TFIFO pointers indicate which row to access (since each TFIFO row includes four entries). Within **FIFOMemory**, the wordline drivers for the memories are qualified by **[RT]ReadHf** and **[RT]WriteHe** to select which of the pointers to use.

FRWrtHe is asserted for one clock cycle to cause a 64-bit word to be written from **Str[0:63]He** into the RFIFO. Module FIFO has two write input strobes (**RWriteHe** and **TWriteHe**), and two read input strobes (**RReadHf** and **TReadHf**), each causing a write

or read to occur in the specified FIFO. Since the FIFO is implemented as a single memory system with one write-only portand one read-only port, **RWriteHe** and **TWriteHe** are mutually exclusive (only one FIFO can be written on a given clock cycle) and **RReadHf** and **TReadHf** are also mutually exclusive (only one FIFO can be read on a given clock cycle); however, the dual ports allow a write and a read on the same cycle, as long as they are not to the same FIFO location.

Flag Logic

FlagLogic contains a counter for each FIFO; these counter produce full flags (**RFullHf** and **TFullHf**), empty flags (**REmptyHf** and **TEmptyHf**), and oveflow signals (**ROvfHf** and **TOvfHf**). The empty signals become **FIFO** outputs are go to the **RTCntl** logic. The full signals are driven become the chip outputs **RFullH** and **TFullH**. The overflow signals are or'ed together, and go into a sticky-register, and produce the chip output **SOvfH**. **FlagLogic** also contains the synchronizers and logic for producing the "write" signals for the flag counters.

GreyCntr is a 2-bit Grey Code counter. It has a synchronous reset, **ResetHe**, and a count-enable, **CmdHe**. It produces the sequence 00, 01, 11, 10 on the outputs **Grey[1:0]He**. Note that the latches are clocked by **EClk**, not by **Clk**.

Mouse is a two-stage synchronizer. It synchronizes the input into an output signal which is synchronous to **Clk**. It containsa a sequence of 4 transparent latches, buitl as mouse-traps, alternately clocked by non-overlapping clocks **MClkH** and **MClkL**. These clocks are produced from **Clk** using a pair of cross-coupled NOR gates.

GreyComp compares the Grey Code value on the current clock cycle with that of the previous clock cycle, to determine how many counts have occurred. The output is a 2-bit binary (not Grey-coded) number. The **ResetHf** input clears all 4 latches. (This reset is not really necessary, since the **ResetHe** inputs to the **GreyCntr**'s could be held high for many cycles, and the zero outputs allowed to propagate through the synchronizers and the **GreyComp**'s).

The flag counter, which is cleared on reset, incremented when an entry is written to the FIFO, and decremented when an entry is read. Thus the counter value represents the number of entries in the FIFO. The zero-condition signal for each counter is computed in look-ahead fashion to generate the two "empty" signals, **REmptyHf** and **TEmptyHf**.

The RFIFO counter value, **RCount[0:7]Hf**, is compared to the limit value **RLim[0:6]Hf** (MSB is understood 0) to produce an almost-full signal **RFullHf**; **RFullHf** is asserted whenever the value represented by **RCount[0:7]Hf** is greater than the value represented by **RLim[0:6]Hf** (**RLim7Hf** is understood 0). Similarly, **TFullHf** is asserted whenever the value represented by **TCount[0:9]Hf** is greater than the value represented by **TLim[0:8]Hf** (**TLim9Hf** is understood 0). Note that the comparison is "greater-than" instead of "greater-than or equal to"; this insures that the "full" signals are de-asserted after a reset, because the counters are 0, so the comparison fails even if the limit values have not been initialized. **RFullHf** and **TFullHf** go to latching output pads to produce the chip outputs **SRFullH** and **STFullH**.

IV.2.2.3 RTCntl Module

This module controls the read port of the FIFOs, arbitrates between RFIFO commands and TFIFO commands and passes the command to the sequencer, and processes "meta" commands. In order to perform this arbitration, **RTCntl** contains a number of semaphore counters, which are P'ed (wait and then decrement) or V'ed (increment) by special "meta" commands, which may be placed in either FIFO. It also includes the multiplexer which selects between RFIFO and TFIFO output, a latch for storing the next command for **Cntl**, and handshaking logic.

One of the FIFOs is designated the "favorite" FIFO; if **FavRHf** is asserted, the RFIFO is the favorite FIFO, otherwise the TFIFO is favored. **FavRHf** is cleared on power-up, but it can be changed by meta commands. Commands are read from the favored FIFO, until it is blocked by a defer-type block command. If this occurs, commands can be read from the non-favored FIFO; if it is also blocked, then no commands are read until one FIFO becomes unblocked. Each FIFO can be blocked by one of two possible semaphore P commands, are by a wait-for-IC command. These block commands include a "defer" bit which is set to indicate that data should be read from the non-favored FIFO.

The opcodes from the two FIFOs, **RI[0:63]Hf** and **TI[0:63]Hf**, are multiplexed based on which FIFO is to be used; this opcode, and the 3 coefficients, are latched into the command latch. The command latch outputs **[IABC]Word[0:63]Hf** are the inputs to **Cntl**. Each of the two FIFO Read Latches has a "ready" signal, **RRdyHf** and **TRdyHf**, which indicates that there is an unprocessed instruction currently in the Read Latch. Each instruction is of one of two types, as determined by *meta* bit of the opcode, bit 28.

IV.2.2.4 ICCntl Module

ICCntl contains the logic for controlling image composition transfers.

It consists of registers for configuring the machine, two instances of **ICCntlPath**, and four bi-directional pads.

The machine configuration register generates the signals: **AliveHf** which is 1 to indicate that a board is active, **LeftEndHf**, which is 1 to indicate that a board lies at the left-hand end of a virtual machine, and **RightEndHf**, which is 1 to indicate that a board lies at the right-hand end of a virtual machine. **ResetHf** initializes this register to 0, so all boards are "dead" after a reset.

One instance of **ICCntlPath**, named **L2RCntl**, controls the "left-to-right" pathway in the Image Composition network; the other instance, **R2LCntl**, controls the "right-to-left" pathway. Each **ICCntlPath** propagates the "go" chain in the direction of its name (e.g. **L2R**- or **R2L**-); the two "go" chains share two bi-directional input pads. The IO pad **ICLGoH** is connected to both **LGoInHr**, the "go" input to **L2RCntl**, and **LGoOutHf**, the "go" output from **R2LCntl**; similarly, the IO pad **ICRGoH** is connected to both **RGoInHr**, the "go" input to **R2LCntl**, and **RGoOutHf**, the "go" output from **L2RCntl**. Similarly, the IO pads **ICLRdyH** and **ICRRdyH** are connected to the "ready" inputs and outputs of **L2RCntl** and **R2LCntl**. However, each **ICCntlPath** propagates the "ready" chain in the direction opposite the direction the "go"chain is propagated.

An Image Composition network transfer is initiated by asserting the input **MetaOpHf**, with the correct bits of **CmdIHf** set. If bit X is set, then **L2RArmHf** is asserted; this loads the regsisters **L2RFirstHf** and **L2RLastHf**, and "arms" a transfer in **L2RCntl**. Similarly, if bit X is set, then **R2LArmHf** is asserted; this loads the regsisters **R2LFirstHf** and **R2LLastHf**, and "arms" a transfer in **R2LCntl**.

Within **ICCntl**, **ArmHf** loads the 13-bit counter with the appropriate cycle count, and sets the **PendHf** flip-flop. This allows the "ready" signal to propagate through this board. The ready signal can propagate from **RdyInHr** to **RdyOutHr** if three conditions are met: (1) the previous board must be "ready" (the **RdyInHr** input is asserted), or this board is the last board in this transfer and so there is no previous board (**LastHf** is low); (2) this board must be ready, meaning it has a transfer pending which has not already begun (**PendHf** is high and **RunHf** is low), or else this board is "dead" (**AliveHf** is low); and (3) the "go" signal is not being asserted (**GoOutHf** is low).

When the "ready" signal has propagated to the beginning of the chain, to a board with **FirstHf** asserted, **ICCntl** generates the output **GoOutHf**, which is asserted for two cycles, before **GoOutHf** inihibits **RdyOutHf**, which in turn inhibits **GoOutHf**. Then each board propagates "go" from **GoInHr** to **GoOutHf**, since its **RdyOutHf** is asserted. As the "go" signal propagates down the chain, is causes the "ready" signal to recede, as **GoOutHf** inhibits **RdyOutHf** ; on "alive" boards, **GoOutHf** also sets the **RunHf** flip-flop, thereby disabling **RdyOutHf** until **PendHf** goes high again when the next transfer is initiated.

When **GoOutHf** is asserted, the **RunHf** flip-flop is set. This causes the counter to begin running. When the transfer was "arm"ed, this counter was loaded with the number of cycles required for the transfer, minus one (**NBytes[0:4]Hf** *128 + 127); when the counter conuts down to zero, the **PendHf** and **RunHf** flip-flops are cleared. Thus the signal **RunHf** is asserted for exactly the number of cycles required for the transfer. When **PendHf** goes low, this allows another transfer to be initiated.

The chain of boards (from "first" to "last") involved in a transfer may all lie in a straight line, that is, they do not loop around the end of the virtual machine; if this is the case ...

Initializing this mechanism during system start is tricky. When the system is reset, **ResetHf** causes **AliveHf**, **LeftEndHf**, and **RightEndHf** to be cleared. Thus each board starts out "dead". It a board is to be left "dead", nothing more needs to be done to it; it will behave transparently to the Image Composition network, provided that **MetaOpHf** is never asserted, and so none of **[L2R,R2L][Arm,First,Last]Hf** are ever asserted. If the IGC or GP is so brain-dead that this cannot be assured, then the board must be removed from the system. After initialization, software loads the machine configuration registers on all boards that are not to be left for dead; the **AliveHf** register is set to 1, **LeftEndHf** is set to 1 if this board lies at the left-hand end of the virtual machine, and **RightEndHf** is set to 1 if this board lies at the right-hand end of the virtual machine. Naturally, the left-hand-end board must lie to the left of the right-hand-end board, or they may be the same board in a one board virtual machine. **LeftEndHf** and **RightEndHf** *must* remain 0 on all dead boards. After the virtual machine is configured by setting **AliveHf**, **LeftEndHf**, and **RightEndHf** on *all* "alive" boards, some time must be allowed for the "ready" and "go" chains to initialize. Software must wait at least *N* clock cycles, where *N* is the number of boards in the virtual machine, before initiating any transfers. The "ready" chain is guaranteed to be cleared in this time because at least one alive board will be emitting 0's from its two "ready" outputs (since **AliveHf** is 1, and **PendHf** is 0 since no transfers have been initiated) , and these 0's will propagate around

the chain in *N* cycles.

IV.2.2.5 Cntl Module

This module is the sequencer which generates the cycle by cycle micro-instructions outputs of the IGC. It also produces the address outputs, and the coefficient byte-stream and associated control bits (unused for the IGC which drives the TASICs). It also produces several strobes which become IGC outputs and are used for controlling other logic in the rasterizer.

Cntl contains several modules:

(1) Sequencer: microcode memory and sequencer

(2) LpCnt: counters used by Sequencer for counting loops (and as flags)

(3) PixMemAddr: counters for generating address outputs

(4) Serializer: logic for byte-serializing ABC tree coefficients

An instruction is latched into the modules of **Cntl** when exeuction of the previous instruction has ended. At this point, **Sequencer** asserts **NewHf** and begins executing the microcode sequence for the instruction. **NewHf** causes the tree coefficients, and the loop count and address fields from the opcode to be loaded into **Serializer**, **LoopCount**, and **PixMemAddr**.

IV.2.2.5-1 Sequencer Module of Cntl

Sequencer is the heart of the IGC. It is a microcode engine, with 7 branch conditions, single-entry stack, and no instruction pre-fetch. It contains the **UcodeMemory** sub-module, with 1K (1024) words of 64-bit microcode store, and the **AddrGen** sub-module, which controls program flow by generating microcode addresses.

Sequencer asserts **DoneHf** when it is ready for a new instruction. If **IPHf** is also hi (a new instruction is pending), then **NewHf** is asserted, which causes **Sequencer** and the other submodules of **Cntl** to begin executing the next instruction.

Cntl can also be placed into a special mode called RMode (**RModeHf** is asserted), which

is used primarily for reading and writing the **Sequencer** microcode store. RMode can be entered by either (1) asserting the hard-reset signal, **ResetHf**, or (2) writing a special kind of instruction, in which opcode bits **IWord[32:34]Hf** are set to 111. When in RMode, **NewHf** is always asserted. Thus, instructions pass immediately from **RTCntl** into **Sequencer**, without delay. Only a very limited set of instructions (those to read and write a location in microcode memory) should be executed when RMode is in effect.

One of these instructions, in which opcode bits **IWord[32:34]Hf** are set to XXX, is used for writing microcode store. **MWriteHf** is asserted when **Cntl** is in RMode, a new instruction is pending, and the special opcode bits are set.

Sub-module UcodeMemory

UcodeMemory takes the pre-decoded address from **AddrGen** and reads or writes the corresponding microcode word in the memory array.

The microcode word is 64 bits wide, formatted as follows:

0-3	PMAInstr	4-bit control word for pixel-mem address counters
4	TreeStep	step control for Serializer
5	CntF	count-enable for FB Counter
6	Cnt1	count-enable for LoopCounter 1
7	Cnt2	count-enable for LoopCounter 2
8	Cnt3	count-enable for LoopCounter 3
9-16	BrCond	sequencer branch condition
17	BrPol	polarity control for branch test
18	JSR	jump to subroutine (branch and load stack register)
19	RTS	return from subroutine (use stack register address)
20-29	Branch	10-bit branch address (goes to AddrGen)
30-31	Done	Done signal (two copies)
32-63	External	to generate **Instr[0:23]H** and **ExtOp[0:7]H** outputs

The low-order half of the microcode word, bits 0-31, generates control signals for the remainder of **Cntl** . The high-order half of the microcode word, bits 32-63, generates most of the IGC outputs, including **Instr[0:23]** and **ExtOp[0:7]H**.

The microcode memory is 128 rows by 512 columns. The memory word-lines, **Word[0:127]H**, are computed from the 7 MSB's of the address in pre-decoded form:

Hi[0:3]Hf, **Med[0:3]Hf**, and **Lo[0:7]Hf**. The 3 LSB's of the address in pre-decoded form, **RS[0:7]Hf**, select one of 8 rows for each of the 64 microcode bits.

When memory is read, the entire 64-bit word is latched to produce the signals **RdData[0:63]Hf**; the high-order word generates most of the outputs of **Cntl**, via **OutputLatch**, and the lower-order word produces the control signals for **AddrGen** and for the other modules within **Cntl**.

Submodule AddrGen

AddrGen generates addresses for **UcodeMemory** in *pre-decoded* form**.** On any microcycle, 4 possible addreses may be selected: (1) the *new* address, the starting address for the next instruction to be executed, (2) the *incremented* address, the address immediately following the current one, (3) the *branch* address, the address specified in the branch address field, or (4) the *return* address, from the single-level stack register. The branch adress field, **BrAddr[0:9]Hf**, is pre-decoded to produce the signals **BrHi[0:3]Hf**, **BrMed[0:3]Hf**, **BrLo[0:7]Hf**, and **BrRS[0:7]Hf**. Similarly, **NewAddr[0:9]Hf** are pre-decoded to produce the signals **NewHi[0:3]Hf**, **NewMed[0:3]Hf**, **NewLo[0:7]Hf**, and **NewRS[0:7]Hf**.. The incremented address is generated from the current address in pre-decoded form. More or less simultaneously with the pre-decoding, during the first quarter or so of the clock cycle, the address source is selected, based on the sequencer instruction bits of the current microcode word (**RTSHf**, **DoneHf**, **BrCond[0:7]Hf** and **BrPolHf**), the condition codes (the terminal count conditions **TC1Hf**, **TC2Hf**, **TC3Hf**, **TCFHf** from **LoopCount**, and the external condition codes **St0H, St1H**, and **St2H**), and the AddrGen inputs **RModeHf** and **IPHf**. The address source is specified by **Next[H,L]** and **Branch[H,L]**. The four possible addresses are multipexed to get the next address in two stages: first, the *new* or *branch* address is selected according to **Next[H,L]**, and the *incremented* or *retrurn* address is selected according to **RTSH**; next, these two results are selected according to **Branch[H,L]**.

BrCond[0:7]Hf select one of the branch conditions (one of the 4 terminal counts or 3 external condition codes, or a null condition), and **BrPolHf** determines the condition polarity: if **BrPolHf**=0, the branch occurs if condition code is non-zero (the external status input is high, or the terminal count condition is low, meaning the counter value is non-zero); if **BrNZHf**=1, the branch occurs if the external status input or counter value is zero.

Whenever the *Done* bit in the current microcode word is set, so **DoneHf** is asserted, this indicates that the sequencer is ready for a new instruction. If **IPHf** is also asserted (there is

a new instruction ready), then **NextHf** is asserted. The microcode assembler insures that **BrCond[0:7]Hf** are set for an unconditional branch whenever **DoneHf** is asserted, so either the *branch* or *new* address is selected according to **IPHf**. If **IPHf** is asserted, control jumps to the "new" address, specified by the inputs **NewAddr[0:8]Hf**; otherwise control branches to the branch address (usually address 0, the "idle" state). The Done microcode bit is set for the last word of the microcode sequence for each instruction, so control branches directly from the end of one instruction to the first word of the next (if one is available) or to address 0 (if one is not available).

Note that if no branch is desired, **BrCond[0:7]Hf** are all set to 0. If more than one of **BrCond[0:7]Hf** is asseerted, then thre branch occurs if any of the indicated condition codes are asserted (or de-asserted, according to **BrPolHf**).

When RMode is in effect, **RModeHf** causes both **NextHf** and **BranchH** to be asserted, so the *new* address is always selected.

The **AddrGen** output **NewHf** is assered whenever a new instruction is begun; this happens when a new instruction is available (**IPHf** is asserted) and either RMode is in effect (**RModeHf** is asserted) or the previous instruction is done (**DoneHf** is asserted). Note that **NextH** and **NewHf** are similar but not identical (they are the same when RMode is not in effect), and are computed from separate identical copies of **DoneHf**.

If the **MWriteHf** input is asserted, the data on **CWord[0:63]Hf** is written directly into to the microcode memory location specified by **NewAddr[0:8]Hf**. Note that **MWriteHf** can be asserted only when RMode is in effect.

IV.2.2.5-2 LoopCount Submodule of Cntl

LoopCount contains four counters, each with zero-detect, used by **Sequencer** for timing loops, or as flags.

When **NewHf** is asserted, the four loop counters are loaded with the appropriate fields of the opcode of the new instruction, **IWord[0:63]Hf**.

LoopCount has 4 additional inputs, **CntFHf, Cnt1Hf**, **Cnt2Hf**, and **Cnt3Hf** generated directly from bits in the **Sequencer** microcode word. Each of these becomes a count enable signal for one of the counters. When it is asserted on a given clock cycle, the value in the counter is decremented on the next clock edge, and the terminal count signal is

asserted if the new value is zero.

The four terminal count signals, **TCF[HL]f, TC1[HL]f, TC2[HL]f,** and **TC3[HL]f,** are the only outputs of **LoopCount**, and go to **Sequencer** where they can be tested as branch conditions. The terminal count signals are valid even on the first cycle of any instruction (immediately after the counter is loaded by **NewHf**); thus the count field can be used simply as a flag which the microcode can test.

When **NewHf** is asserted, it overrides the count-enable signals for **LpCnter1, LpCnter2,** and **LpCnter3**. However, for **FBCnter**, **NewHf** forces the count-enable; thus, the value in the opcode field is decremented by one (modulo the counter size) when the value is loaded. This is so **FBCnter** can be loaded with the number of fractional bits minus one, thereby facilitating a loop to discard the fractional part of the LEE result; when **FBCnter** is used as an ordinary loop counter, the microcode asembler pre-increments the user-specified starting loop count value so that this effect is transparent to the microcode writer.

The following table summarizes the four loop counters:

	Number Bits	Field in Opcode (LSB - MSB)	Count-Enable Signal	Terminal Count Signal
FBCounter	2	22 - 23	CntFHf v NewHf	TCF[HL]f
LoopCounter1	3	19 - 21	Cnt1Hf ^ !NewHf	TC1[HL]f
LoopCounter2	4	35 - 38	Cnt2Hf ^ !NewHf	TC2[HL]f
LoopCounter3	6	39 - 44	Cnt3Hf ^ !NewHf	TC3[HL]f

Each loop counter consists of a half-adder which adds either 0 or -1 depending upon the **CountHf** input, followed by a latch. The other adder input comes from a two-into-one multiplexer which selects either the value from the latch, **OutHf** , or the new count value, **InHf**. The zero-detect signal is computed in look-ahead fashion by looking at the multiplexer output and the count-enable signal.

There is no logic which causes the loop counters to halt at 0. The microcode writer must take care that a loop counter does not underflow before the terminal count condition is detected by a conditional sequencer instruction.

IV.2.2.5-3 PixMemAddr Submodule of Cntl

PixeMemAddr generates the address outputs of **Cntl**. For any IGC instruction, 3 sequences of addresses may be specified by starting address in fields of the opcode.

PixMemAddr contains three 9-bit adders and base registers (one for each of the 3 pixel-memory address sequences), four 9-bit up/down counters (the three addresses, plus the refresh address counter), logic to decode the address instruction, and a multiplexer for selecting among the counters.

When **NewHf** is asserted, the counters **DstCntr**, **SrcCntr**, and **AuxCntr** are loaded with the appropriate fields of the opcode of the new instruction, **IWord[0:63]Hf**, as follows:

Field in Opcode (LSB-MSB)

DstAddrCntr	bits 10-18
SrcAddrCntr	bits 45-53
AuxAddrCntr	bits 55-63

A fourth counter, **RefCntr**, is loaded with zero using a special command in which **IWord[34:32]Hf** are [101]. Normally this is done only once, at system initialization.

PixMemAddr has 4 additional inputs, **PMAInstr[0:3]Hf**, generated directly from bits in the **Sequencer** microcode word. These are decoded to produce the controls for the counters and multiplexer, as follows:

PMAInstr[3:0]Hf	Selects	Counter Action
0x0	RefCntr	decrement
0x1	"	hold
0x2	"	increment
0x3	"	increment by 2
0x4	DstCntr	decrement
0x5	"	hold
0x6	"	increment
0x7	"	increment by 2
0x8	SrcCntr	decrement
0x9	"	hold
0xA	"	increment
0xB	"	increment by 2
0xC	AuxCntr	decrement
0xD	"	hold
0xE	"	increment
0xF	"	increment by 2

In this table, the action in the "Counter" column refers to the counter selected by the multiplexer (in the "Mux" column). Only the selected counter can be incremented or decremented. The increment or decrement always occurs *after* the value is used; that is, if **PMAInstr[3:0]** = 0xA, then the value from **SrcCntr** is selected by the multiplexer, and **SrcCntr** is incremented, so that the new value is used *next* time **SrcCntr** is selected (within the same instruction).

If the counter-load signal is asserted (**NewHf**, or **NewHf** & **IWord[34:32]Hf**==0X1) the increment or decrement operation is ignored; in other words, the load signal overrides the count-enable signals.

A base-offset register can be applied to the pixel-memory operands, when each counter is loaded. This base is applied to any operand in the range 0x1c0 - 0x1ff (a 64-byte area); the upper 3 bits of the operand are stripped off, and this 6-bit quantity is added to the base-offset register to determine the value to be loaded into the counter. However, if the *NoBase* bit of the new instruction, **IWord54Hf**, is asserted, then the base-offset is not applied. The 9-bit base-offset register is loaded when **LoadBaseHf** is asserted; this occurs when **NewHf** is asserted and **IWord[34:32]Hf** == 001.

The multiplexer produces the final output address, **Addr[0:8]Hf**.

IV.2.2.5-4 Serializer Submodule of Cntl

The **Serializer** is the most complicated of the **Cntl** modules. Its function may be summarized as follows:

(1) convert the 64-bit integer or double-precision floating-point coefficients (A,B,C) supplied with an instruction into byte-serial 2's-complement fixed-point numbers with a specified number of fractional bytes

(2) produce a signal which marks the LSByte of each set of coefficients, which controls pipelining in the linear expression evaluators of the EMCs

(3) adjust the C coefficient value to effectively offset the position of the rasterizer region by both an integer region-offset and a fractional sub-pixel offset

When **NewHf** is asserted at the beginning of a new instruction, the three coefficients, A, B, and C, are latched into three instances of the module **CoefSer**, and several bits of the

opcode which determine the handling of the coefficients are latched into module **ILatch**. Every time **Sequencer** asserts the control signal **TreeStepHf**, each **CoefSer** generates the next byte of its coefficient. Each **CoefSer** generates two byte-streams: **Whole[0:7]Hf**, representing the coefficient value, and **Fract[0:7]Hf**, representing the value of the coefficient divided by 256 (subscript 0 is the LSB of each byte). These two byte streams represent the unsigned values; the **CoefSer**'s also generate the signals **ASignHf**, **BSignHf**, and **CSignHf**, representing the signs of the coefficients.

These byte streams for A and B pass into two instances of the **ABChain** module, which simply two's complements the byte-stream if the sign is negative, and applies a delay to generate the TrA an TrB outputs of **Serializer**. The byte streams for A and B also pass into two instances of the **Adjust** module, which compute the two offsets to be added to the C coefficient. These offsets are added to a signed and delayed version of the C coefficient within **CChain**, which produces the TrC output of **Serializer**.

The region- and subpixel-offset values are loaded into **Serializer** by a special type of instruction which has bits 61-63 of the opcode set to XXX. For this instruction, the C data represents these offset values. The values are latched into module **OffsetLatch**, which feeds the values to the **AAdjust** and **BAdjust** modules.

The module **Timing** generates the signals **Step[0:X]Hf** and **Pipe[0:X]Hf**, which control the flow of the coefficient byte streams through the various other modules of **Serializer**; it also generates the signals **TrStHf** and **TrLSBHf**, the two remaining outputs of **Serializer**.

Submodule ILatch

ILatch handles all the bits of the opcode which specify how the coefficients are used for the new instruction (**IWord[19:27,61:63]Hf**) and produces the appropriate mode signals for the other circuitry in **Serializer**.

CMaskHf is asserted if **IWord27Hf** is low, meaning the new instruction does not use coefficients. **ABMaskHf** is asserted if either **IWord27Hf** or **IWord26Hf** is low, meaning that the new instruction does not use the LEE in linear mode (or at all). **FracEnHf** is asserted if **IWord[27,26,24]Hf** are all three high, meaning the new instruction uses the LEE in linear mode, and the coefficients are floating-point. **Mask[0:7]Hf** are derived from **IWord[19:27]Hf**. These four outputs are latched when **NewHf** is asserted for a new instruction, so are valid during the entire instruction. **FracEnHf** is delayed by a clock cycle to match the circuitry in **AAdjust** and **BAdjust**.

The output **FloatHf** is asserted if **IWord27Hf** and **IWord24Hf** are asserted; this means the incoming instruction uses the coefficients and they are floating-point.

The outputs **FBytes[0:1]Hf** are simply **IWord[22:23]Hf**; they represent the number of fractional bytes to be generated (for floating-point coefficients).

The output **OffEnHf** is asserted when NewHf is asserted if **IWord[61:63]Hf** are 010. This causes the C coefficient values to be loaded into submodule **OffsetLatch**.

Submodule OffsetLatch

OffsetLatch latches the C coefficient values when **OffEnHf** is asserted; this means that this is a special instruction used for loading the region and subpixel offset registers.

The bits are interpreted as follows: ...

Submodule ABCoefSer (instances ACoefSer and BCoefSer)

There are two instances of **ABCoefSer**, one for each of the coefficients A and B; described below is a similar submodule, **CCoefSer**, which handles the C coefficient.

When **NewHf** is asserted, the corresponding coefficient is loaded into **ABCoefSer**. The coefficient value is encoded on the inputs **Word[0:63]Hf** in either IEEE-standard double-precision floating-point format, as a 64-bit integer, or as a 64-bit fixed point number with 16, 24, or 32 fractional bits.The double precision floating-point format is:

bits 0-51 normalized mantissa (understood 1 to left of radix), MSB at bit 52
bits 52-62 exponent in excess-1023 form
bit 63 signbit

Three components of **CoefSer** are loaded: the exponent counter, the mantissa latch, and the signbit latch.

The opcode bits IWord[22:24,26:27]Hf describe the coefficient value as follows:

27	26	24	23	22	Type
0	X	X	X	X	Non-coefficient
1	0	X	X	X	Constant
1	1	0	0	0	Linear, Integer
1	1	0	0	1	Linear, Integer 6 : 2
1	1	0	1	0	Linear, Fixed 5 : 3
1	1	0	1	1	Linear, Fixed 4 : 4
1	1	1	X	X	Linear, Floats

Since integers are in two's-complement form, the signbit register is loaded with 0 for integers; for floats, it is loaded to the signbit (bit 63).

For integers, all 64-bits are loaded directly into the Mantissa Latch. For floats, the understood 1 to the left of the radix point is added to the 52-bit mantissa field, and the reamining 11 bits are set to zero; this mantissa is left shifted if necessary, before being loaded into the Mantissa Latch, so that the radix point is aligned at a byte boundary. The shift amount is determined by adding 5 to the 3 LSB's of the exponent field, to yield a value between 0 and 7.

The most significant 8 bits determine the initial value for the exponent counter. The exponent is represented in excess-1023 form. The *true* exponent is determined by subtracting 1023 from the exponent field. If the 3 LSBs of the true exponent are 4, the mantissa should not be shifted (the understood 1 to the left of the radix point is in bit position 4 of its byte); if true exponent % 8 is 5, the mantissa must be shifted one to the left, and so on so that the mantissa is shifted 7 bits to the left if true exponent % 8 is 3. Thus, the barrel shifter adds 5 to the 3 LSBs of the exponent field (1 to account for the 1023 subtracted to determine the true exponent, and 4 for the offset) to determine the shift to be applied.

The stream of bytes representing the unsigned coefficient is generated by selecting bytes, from LSByte to MSByte, from the mantissa latch. This is done by means of a set of 8 precharge-evaluate bit-lines, one for each bit of the byte. These lines are precharged during **ClkL**, evaluated during **ClkH**, and inverted and latched on the falling edge of **ClkH**. There are two sets of these bitlines, one for **Whole[0:7]Hf**, and one for **Fract[0:7]Hf**. The signals **Select[0:8]Hf** determine which, if any, of the bytes from the mantissa latch is to be asserted onto the bitlines and latched. No more than one of **Select[0:8]Hf** is asserted on any clock cycle; if none is asserted, 0's are latched.

Select[0:8]Hf are generated from **TokEnabHf** and **TokCntr[0:4]Hf**. When coefficients are converted to fixed-point byte serial form, the fixed-point representation can have from 0 to 3 fractional bytes, as determined by **FBytes[0:1]Hf** (bits 22-23 of the opcode). If the exponent is so small that the mantissa lies completely to the right of the least significant fractional byte, or if the exponent is so large that the LSB of the mantissa lies more than 64 bits to the right of the radix point, then the byte-stream should be all zeroes; in the first case, the fixed-point equivalent *is* zero, and in the second case, the non-zero bytes would never appear, since no more than 8 bytes (plus the fractional bytes) of the coefficients are ever generated, since the maximum operand length for instructions which use the LEE is 8.

bits 53 - 63	actual exp	shift	TokEnab	TokCntr[4:0]
0x000 - 0x3fa	< -4	X	0	X
0x3fb - 0x3fe	-4 to -1	0	-31	0
0x3ff	0	4	1	0
0x400	1	5	1	0
0x401	2	6	1	0
0x402	3	7	1	0
0x403	4	0	1	1
0x40A	11	7	1	1

If the coefficient is integer, things are much simpler. The sign is considered to be zero, since the integer is two's-complement. The entire 64-bits of the integer are just latched into the Mantissa Latch with no shifting. A one is loaded into the **TokEnabHf** latch, and 7 into the **TokCntrHf** counter, so that the eight bytes of the integer are output in order.

After the mantissa is shifted and loaded into the mantissa latch, the exponent is decoded and loaded into the **TokEnabHf** latch and **TokCntrHf** counter, and the sign is latched, generation of the unsigned byte-streams can begin. Each time **TreeStepHf** is asserted, the value on **TokCntr[0:4]Hf** is decremented and the next byte of the coefficients is generated.

Since there is no use for fractions of the C coefficient, the bit-lines and latch for **Fract[0:7]Hf** are absent in **CCoefSer**.

TreeStepHf is delayed by a cycle, in **Timing**, to represent the cycle of delay injected by the latch for the **CoefSer** outputs **Whole[0:7]Hf** and **Fract[0:7]Hf**. Thus, **Step0Hf** is asserted once for each new byte which appears on **Whole[0:7]Hf** and **Fract[0:7]Hf**.

If the opcode specifies that the LEE is to be operated in constant mode for this instruction, then the ILatch output **ABMaskEnHf** is asserted; in **ACoefSer** and **BCoefSer**, this causes the **WholeHf** outputs to be zeroed, so the LEE result will be the desired F(x,y) = C.

If the instruction does not use the ABC coefficients, it may use an 8-bit field of the opcode as a single-byte value to be used in the pixel-ALUs. This value is passes down using the LEE. In this case, the ILatch output **CMaskEnHf** is asseerted. This causes **CCoefSer** to generate **Mask[0:7]Hf** for the first byte; any additional bytes are garbage.

If the coefficient is in range, 0's and 1's are produced as the token is shifted to the left in the Token Shifter. When the counter underflows and none of the SelectHf are asserted, only 0's are produced; the reflects the fact that sign-extension in a sign-magnitude representation is realized simply by adding trailing 0's. If the exponent is so large that initially none of the SelectHf are asserted, then the first few bytes of the coefficient will be 0's. This reflects the fact that precision is lost when converting very large numbers to fixed-point. When the exponent field is outside the range 0x3fb - 0x4fa (actual exponent between -4 to 251) then TokEnabHf is zero, the SelectHf are never asserted, and the coefficient is zero. This is the right result for very small coefficients (the fixed point equivalent for any number less than 2^-8*fbytes *is* zero), and is ok for coefficients with exponents above 251, because the mantissa bits all lie more than 128 positions to the left of the radix point, so the first 16 bytes of the fixed-point equivalent are zero, and no instruction can use more than 16 bytes of the LEE result in any case.

Submodules AChain, BChain, and CChain

Submodules **AChain** and **BChain** simply two's-complement the A and B coefficient byte-streams, based on the signbits **ASignHf** and **BSignHf**, and then delay the byte-streams by 8 additional cycles to produce **Serializer** outputs **TrA[0:7]Hf** and **TrB[0:7]Hf**.

Submodule **CChain** is quite similar, but it adds in the multiples of the A and B coefficienst, from **AAdjust** and **BAdjust**, which adjust the C coefficient for the region and subpixel offset. It produces the **Serializer** output **TrC[0:7]Hf**.

Submodule Adjust

There are two instances of **Adjust**, one for the A coefficient (**AAdjust**), and one for the B coefficient (**BAdjust**).

Adjust performs two's compementing on the coefficient bit-stream if the sign is negative,

When a coefficient register is loaded, the signbit of the coefficient is latched into leafcell **WordLatch**. When the coefficients are posted, the signbit is transferred to one instance of **PostCLatch**, and the other instance of **PostCLatch** is loaded with the logical-AND of the latched version **Tok64LSP2** of the out-of-range exponent decoder output, and one of the **ILatch** outputs **ABEnHSP2**, **DEFEnHSP2**, or 0 (depending on which instance of **CoefGate**).

The bit-stream from **CoefSer**, **BitHSP2**, is two's complemented using the following algorithm: "pass the bits unchanged up to and including the first 1 that is seen, then invert all the remaining bits". **SOneHSP2** is cleared when the coefficients are posted, and remains low until the first 1 is seen. If the sign-bit was 1, then **InvLSP2** is asserted when **SOneHSP2** goes high, and the remaining bits of the coefficient will be inverted in the exclusive-OR gate.

Specifying Serializer Control Fields in the Opcode

This section summarizes the bits in the opcode which affect the **Serializer**. These fields are: the *Coefs* bit, the *Linear* bit, the *Float* bit, the *Double* bit, the 8-bit *Mask* field, and the 2-bit *FBCnt* field.

IV.2.2.6 OutputLatch Module

OutputLatch contains the output pads for the **Cntl** outputs as well as some delay and multiplexer logic.

The outputs are of two types. Most of the outputs, including the micro-instruction outputs **Instr[0:23]H**, the address outputs **Addr[0:8]H**, and the tree outputs **Tr*H**, are driven out in complementary pairs: each output signal drives a pair of output pads, both of these latch the signal, one drives an active-high version of the signal, and on drives an active low output. However, the external operation strobes **ExtOp[0:7]H** are driven out by simple latched LVTTL-level output pads.

The outputs **ExtOp[0:7]H** are driven by a multiplexer controlled by the test-mode inputs **Test[0:5]H**. Under normal operating conditions, **Test[0:5]H** are grounded and multiplexer channel 0 is selected.

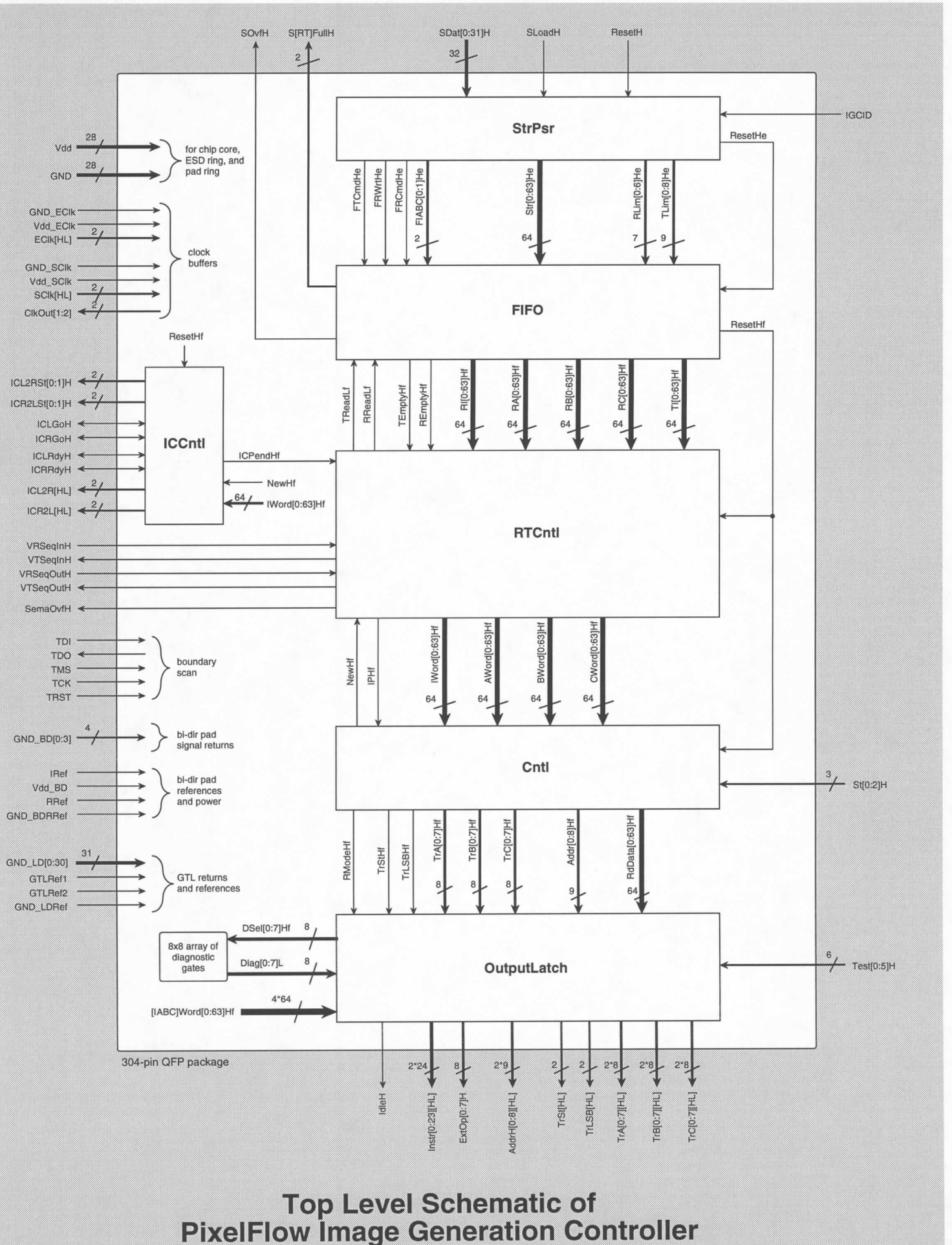
In normal mode, **Instr[0:23]H** and **ExtOp[0:7]H** are generated directly from the high-order 32-bit word of the **Sequencer** output word, **RdData[32:63]Hf**. An additional 10 cycles of delay are applied to generate **Instr[0:23]Hf**; this matches the latency introduced into the **Tr*Hf** signals within **Serializer**. This pipeline delay is not applied to the external operation strobes, **ExtOp[0:7]H**.

In normal mode, **RdData[32:63]Hf** are gated to zero when **RModeHf** is asserted. This prevents the state of the EMCs, TASICs, and external memory from being modified if an IGC is put into RMode to do an "on-the-fly" reload of the sequencer microcode store.

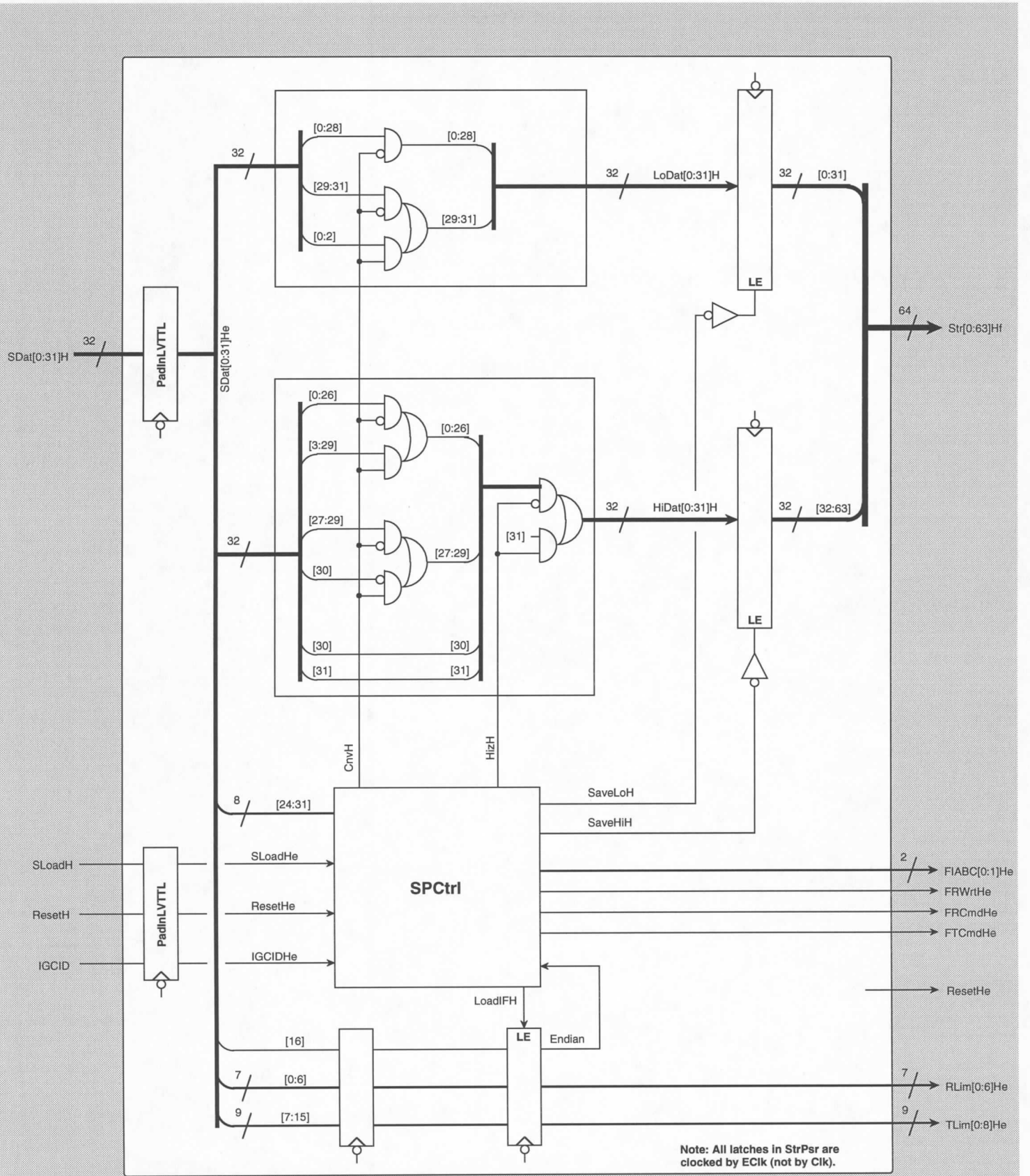
The address outputs **Addr[0:8]Hf** are delayed by only 8 additional cycles; this matches the latency of **Instr[0:23]Hf** and **Tr*Hf**, since **PixMemAddr** introduces two cycles of latency in its input block and output multiplexer.

In the various test modes, the output **ExtOp[0:7]H** are driven by alternate sources. These are summarized in the following table:

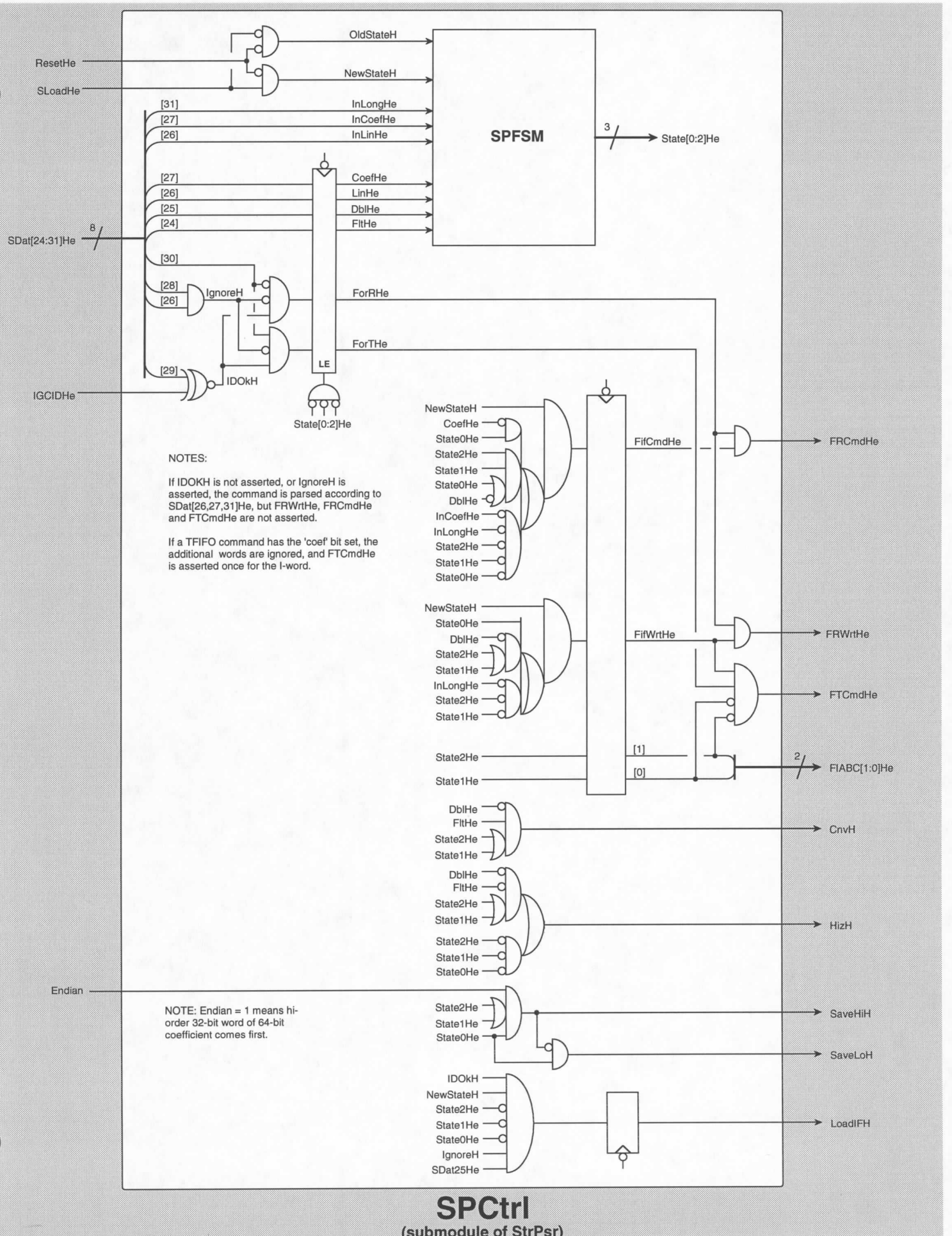
The output **IdleH** is computed from **Diag[0:5]L**. It is asserted when both FIFOs and their read latches are empty, no instruction is pending, and the Sequencer is in the Done state. **IdleH** does not depend on the state of **ICCntl**, so it may be asserted even when an Image Composition transfer is pending. Diag[0:5]L have the correct values for computing IdleH only when DSel0H is asserted. Thus **IdleH** is valid only when the **Test[0:2]H** inputs are low; otherwise its value is nonsense.



Top Level Schematic of PixelFlow Image Generation Controller


SDat[0:31]H
32
PadInLVTTL
SDat[0:31]He
32
[0:28]
[0:28]
[29:31]
[29:31]
[0:2]
32
LoDat[0:31]H
LE
32
[0:31]
64
Str[0:63]Hf
[0:26]
[0:26]
[3:29]
[27:29]
[27:29]
[30]
[31]
32
HiDat[0:31]H
LE
32
[32:63]
[30]
[30]
[31]
[31]
CnvH
HizH
8
[24:31]
SaveLoH
SaveHiH
SLoadH
ResetH
IGCID
PadInLVTTL
SLoadHe
ResetHe
IGCIDHe
SPCtrl
2
FIABC[0:1]He
FRWrtHe
FRCmdHe
FTCmdHe
ResetHe
LoadIFH
[16]
7
[0:6]
9
[7:15]
LE
Endian
7
RLim[0:6]He
9
TLim[0:8]He
Note: All latches in StrPsr are clocked by EClk (not by Clk).

StrPsr



NOTES:

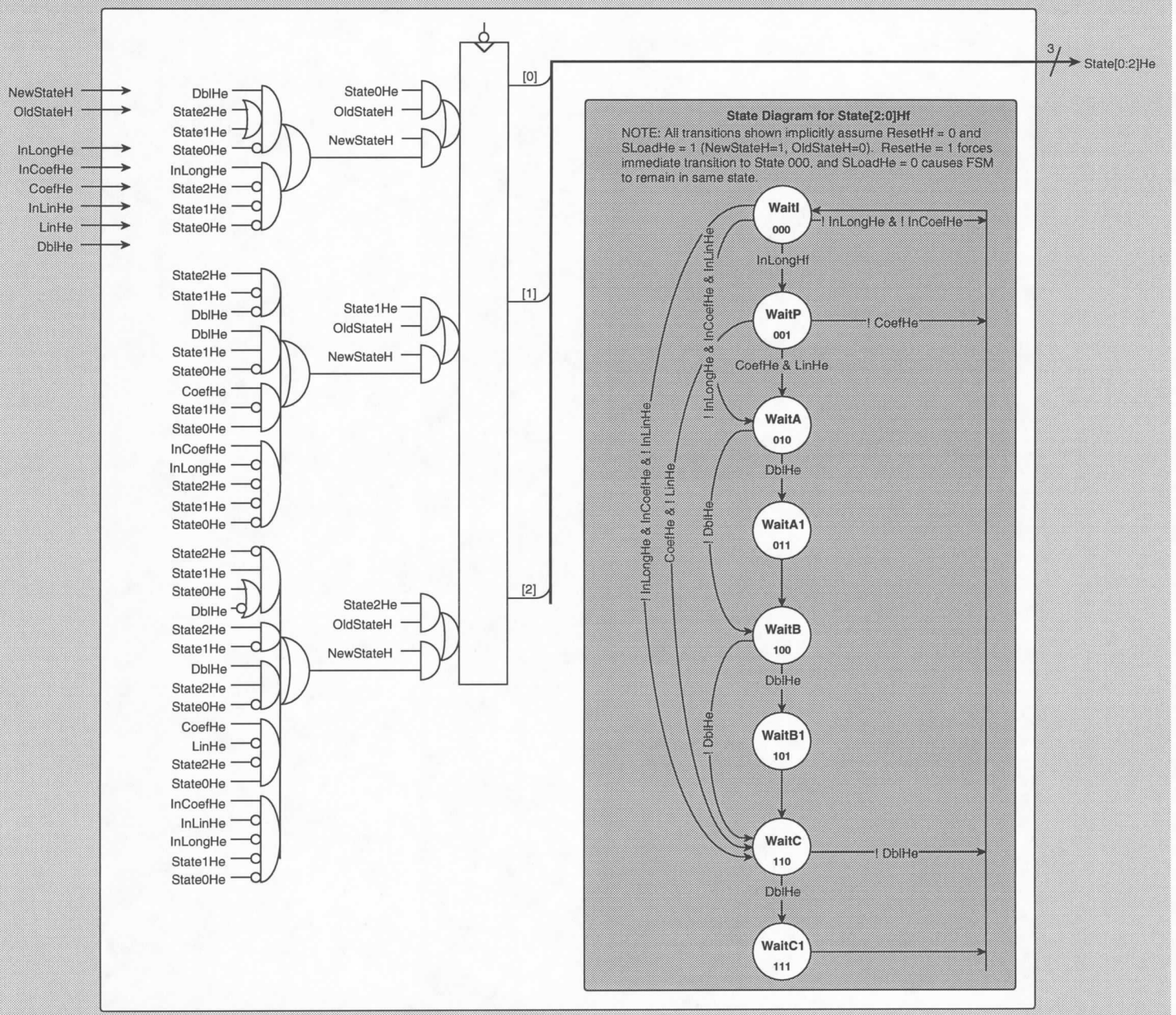
If IDOKH is not asserted, or IgnoreH is asserted, the command is parsed according to SDat[26,27,31]He, but FRWrtHe, FRCmdHe and FTCmdHe are not asserted.

If a TFIFO command has the 'coef' bit set, the additional words are ignored, and FTCmdHe is asserted once for the I-word.

NOTE: Endian = 1 means hi-order 32-bit word of 64-bit coefficient comes first.

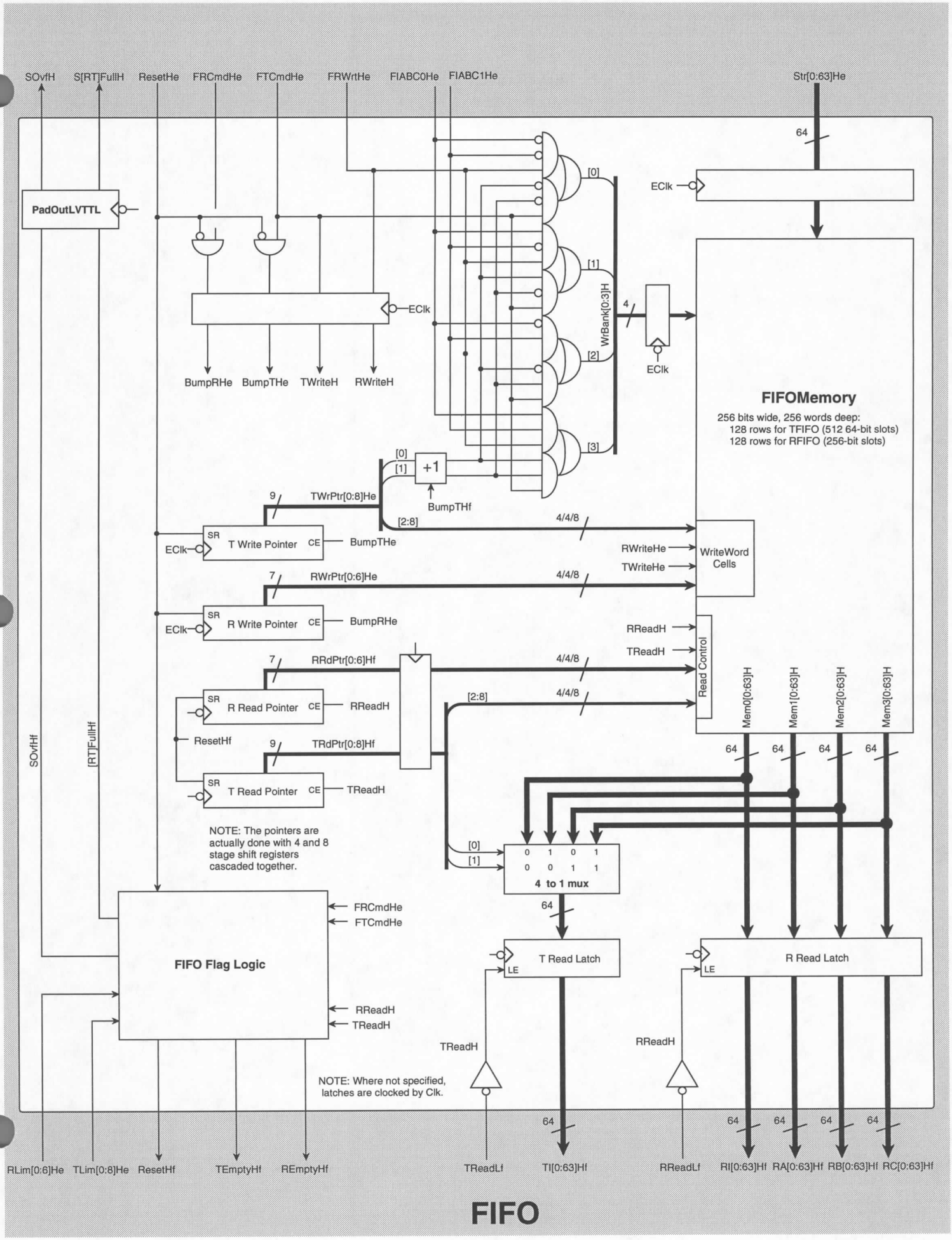
SPCtrl

(submodule of StrPsr)

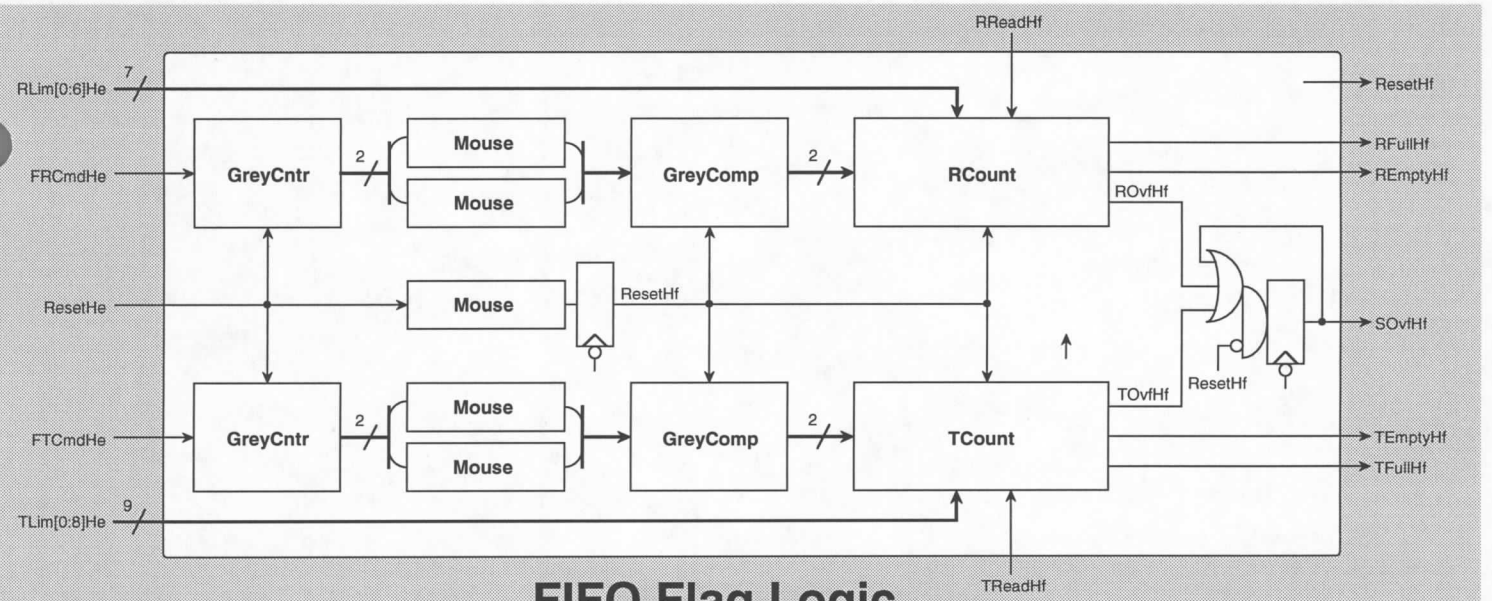


SPFSM

(submodule of SPCtrl)

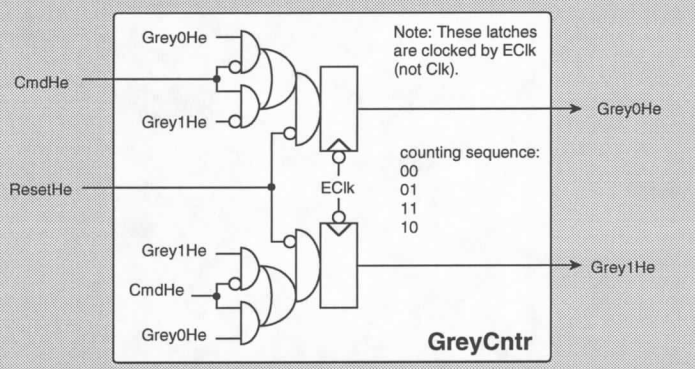

SOvfH
S[RT]FullH
ResetHe
FRCmdHe
FTCmdHe
FRWrtHe
FIABC0He
FIABC1He
Str[0:63]He
64
ECIk
PadOutLVTTL
[0]
[1]
[2]
[3]
WrBank[0:3]H
4
ECIk
ECIk
BumpRHe
BumpTHe
TWriteH
RWriteH
FIFOMemory
256 bits wide, 256 words deep:
128 rows for TFIFO (512 64-bit slots)
128 rows for RFIFO (256-bit slots)
[0]
[1]
+1
BumpTHf
9
TWrPtr[0:8]He
[2:8]
4/4/8
ECIk
SR
T Write Pointer
CE
BumpTHe
RWriteHe
TWriteHe
WriteWord Cells
7
RWrPtr[0:6]He
4/4/8
ECIk
SR
R Write Pointer
CE
BumpRHe
RReadH
TReadH
Read Control
7
RRdPtr[0:6]Hf
4/4/8
SR
R Read Pointer
CE
RReadH
[2:8]
4/4/8
ResetHf
9
TRdPtr[0:8]Hf
SR
T Read Pointer
CE
TReadH
Mem0[0:63]H
Mem1[0:63]H
Mem2[0:63]H
Mem3[0:63]H
64
64
64
64
NOTE: The pointers are actually done with 4 and 8 stage shift registers cascaded together.
[0]
[1]
0 1 0 1
0 0 1 1
4 to 1 mux
64
SOvfHf
[RT]FullHf
FRCmdHe
FTCmdHe
FIFO Flag Logic
RReadH
TReadH
T Read Latch
LE
R Read Latch
LE
TReadH
RReadH
NOTE: Where not specified, latches are clocked by Clk.
64
64
64
64
64
RLim[0:6]He
TLim[0:8]He
ResetHf
TEmptyHf
REmptyHf
TReadLf
TI[0:63]Hf
RReadLf
RI[0:63]Hf
RA[0:63]Hf
RB[0:63]Hf
RC[0:63]Hf

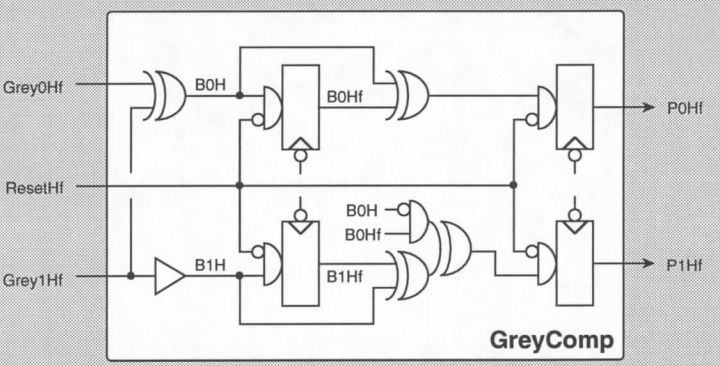
FIFO

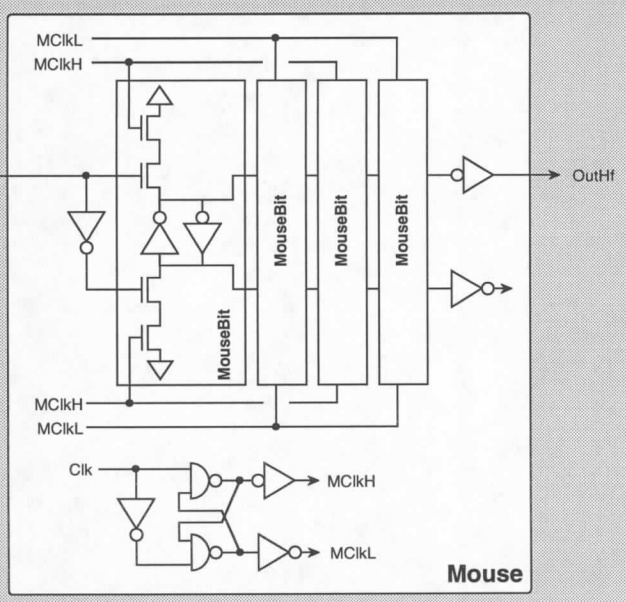


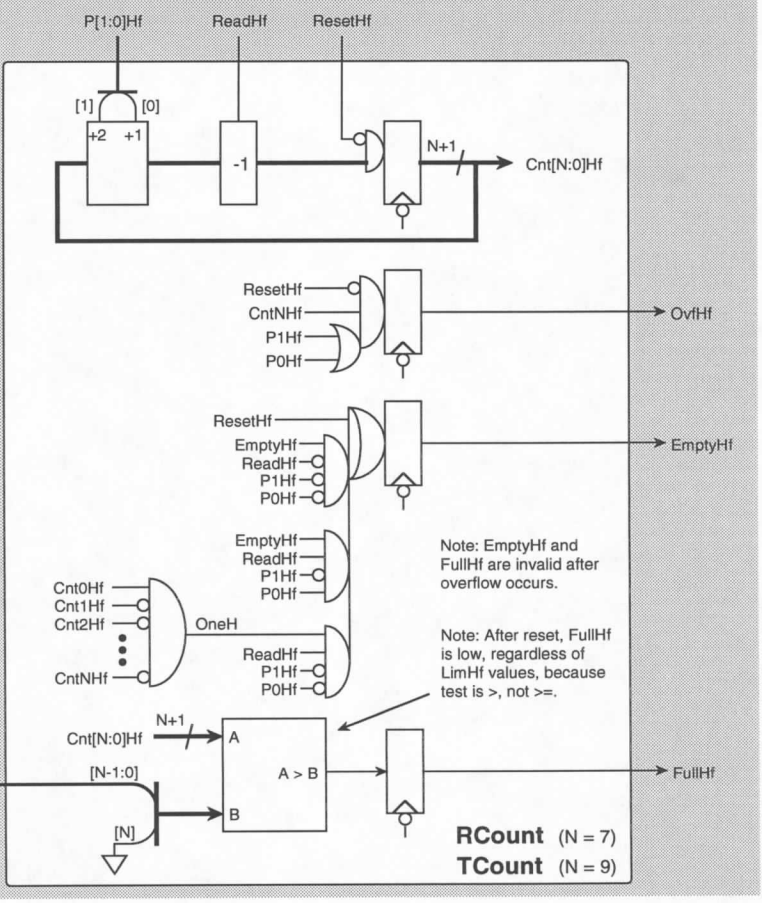
FIFO Flag Logic

(submodule of FIFO)



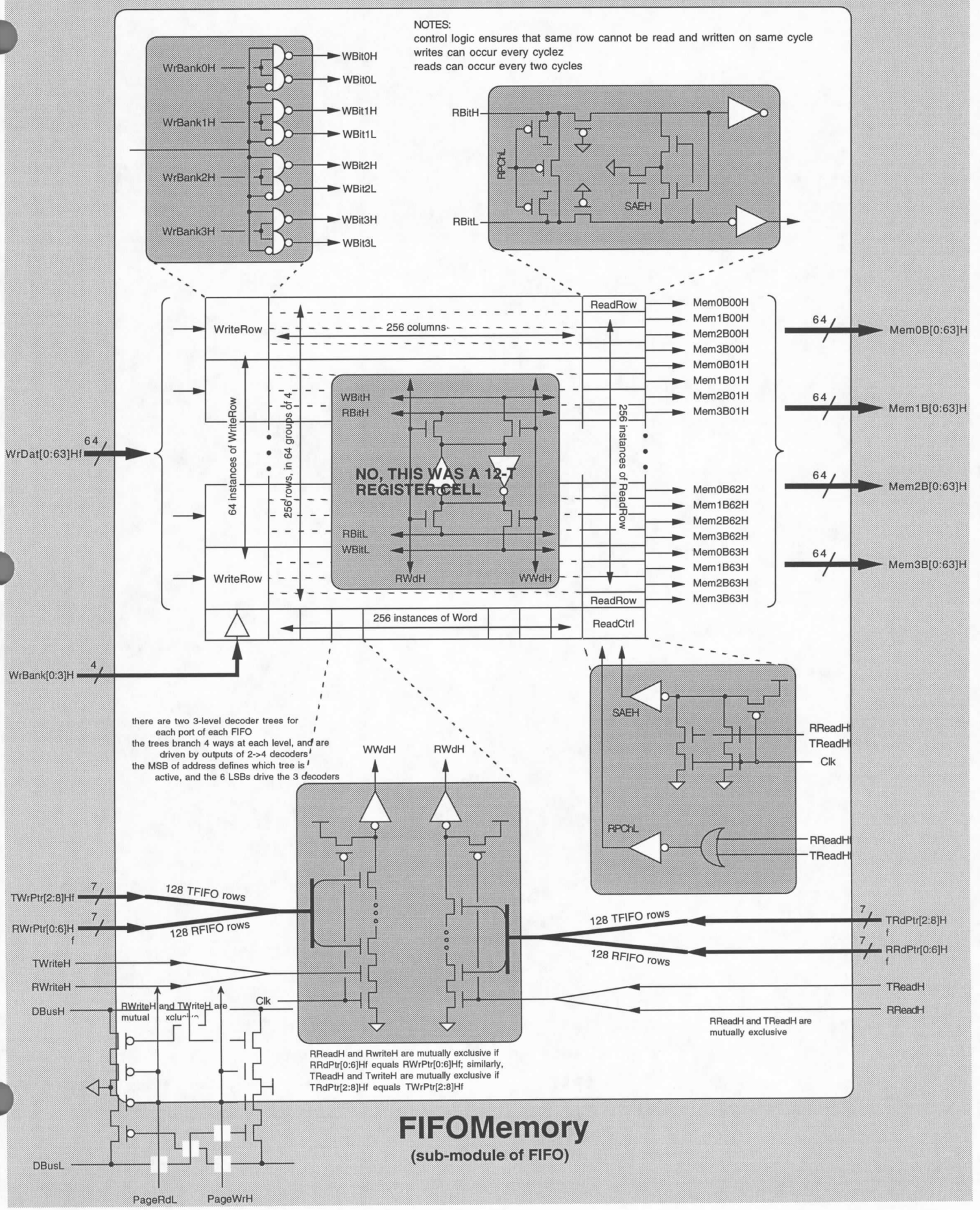






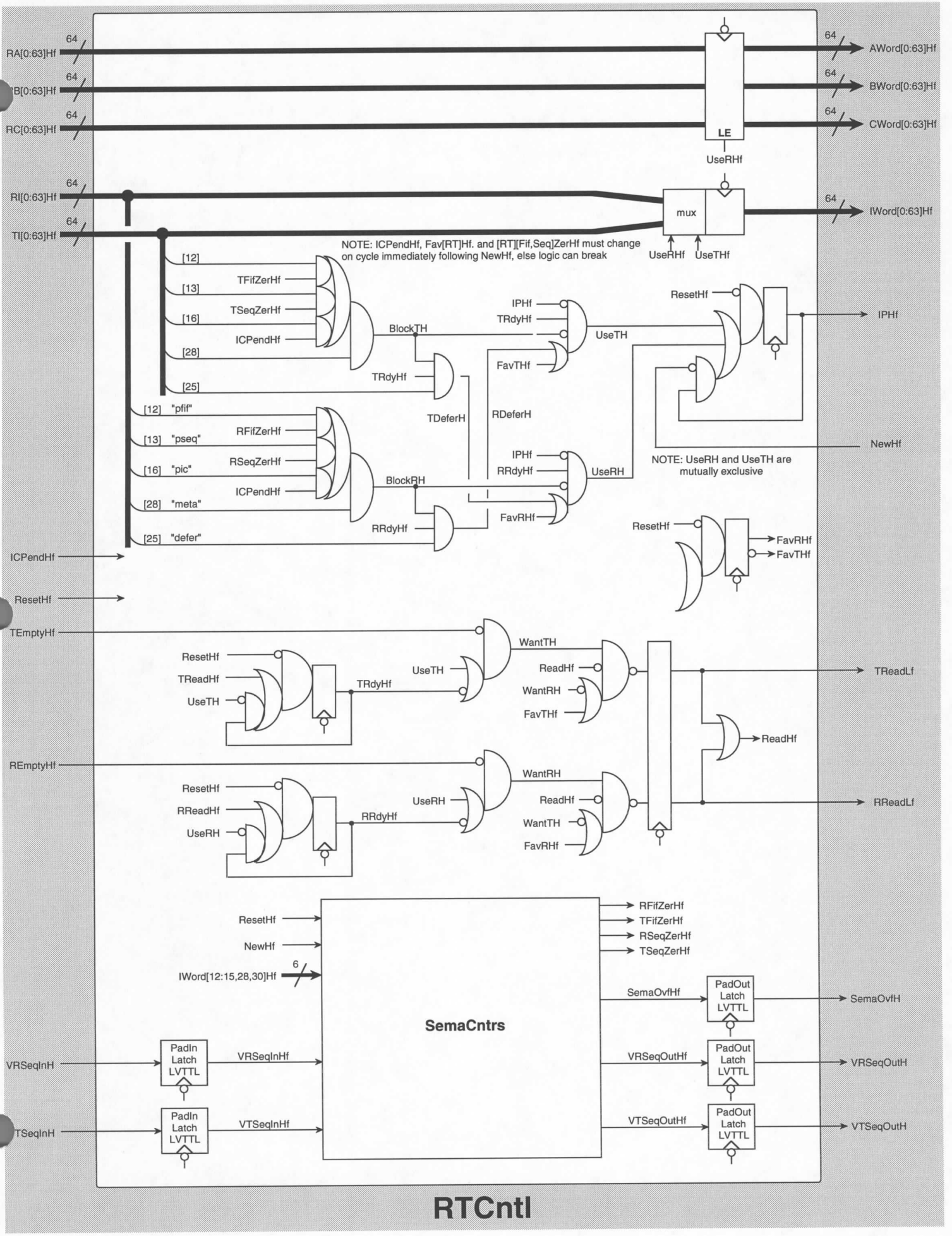
NOTE: Latches clocked by EClk are explicitly indicated; all other latches are clocked by Clk.

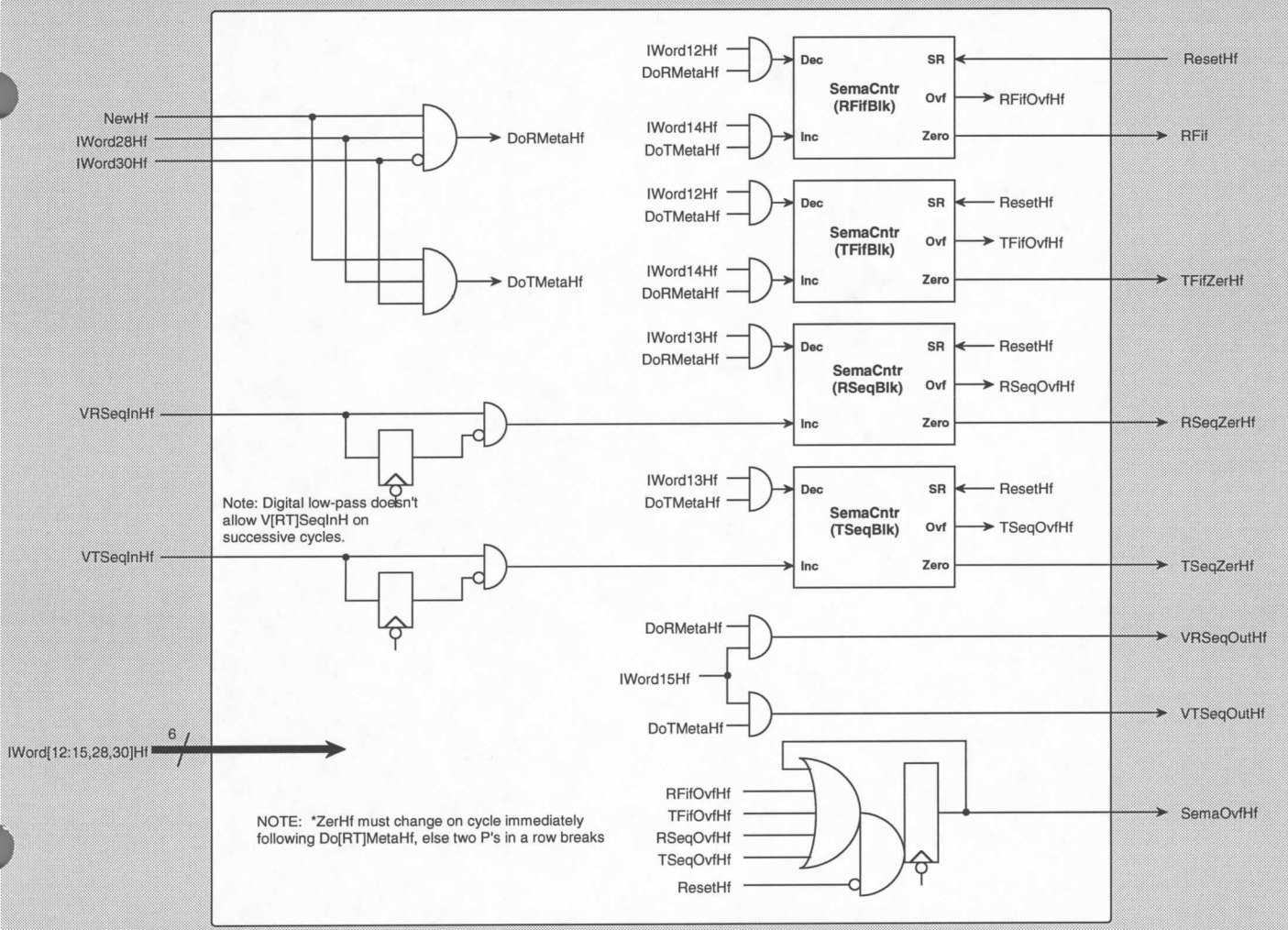
THIS SCEMATIC NEEDS MAJOR REVISIONS !!!



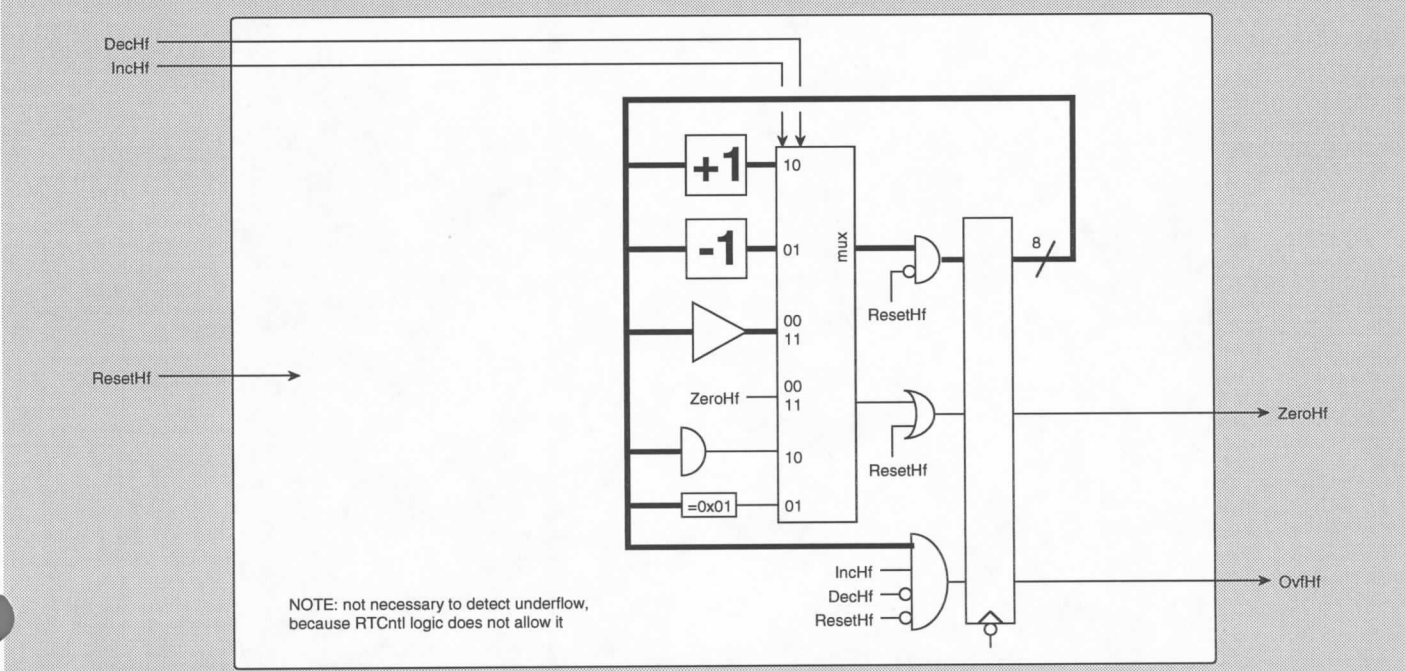
FIFOMemory

(sub-module of FIFO)

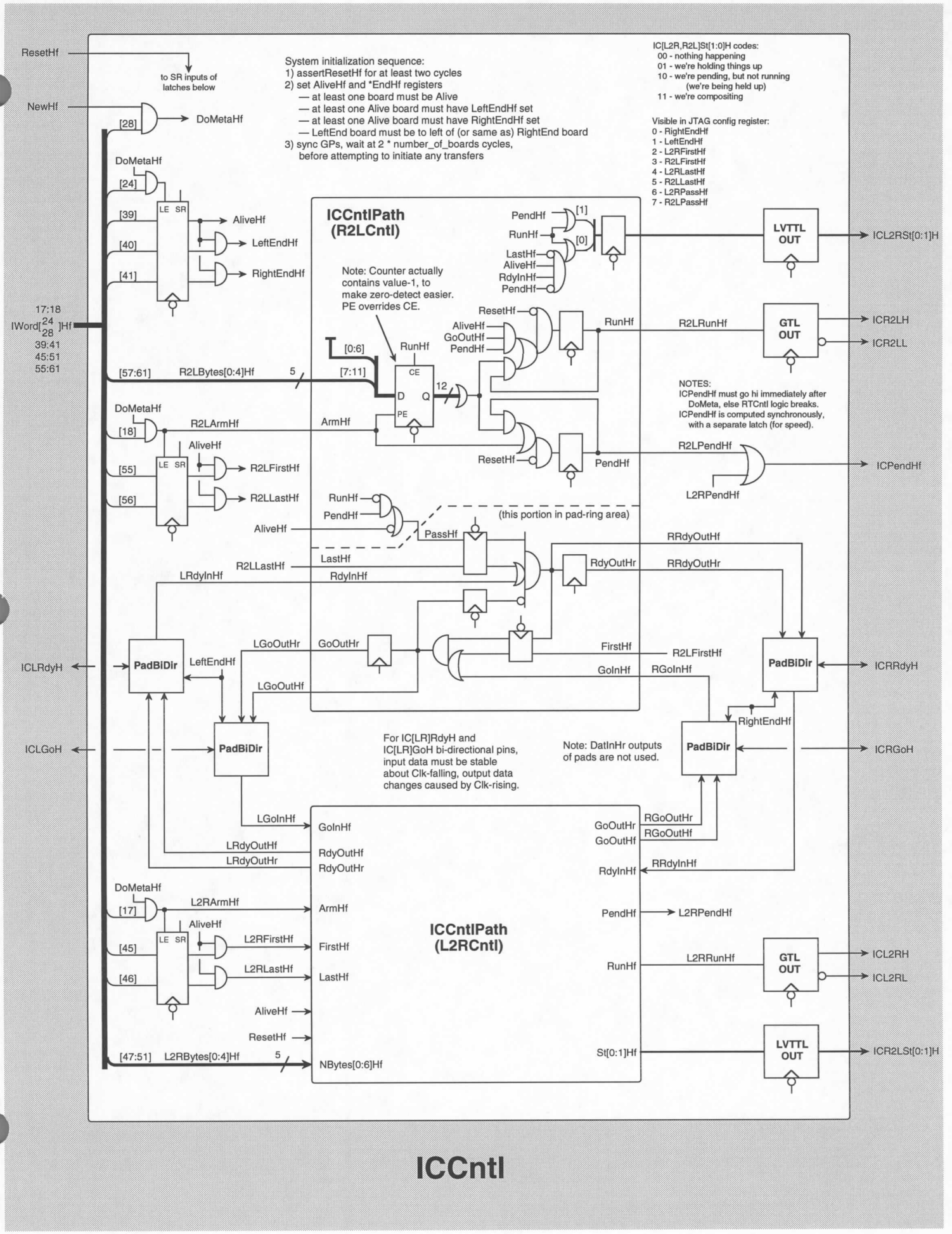

RA[0:63]Hf
RB[0:63]Hf
RC[0:63]Hf
64
LE
UseRHf
AWord[0:63]Hf
BWord[0:63]Hf
CWord[0:63]Hf
RI[0:63]Hf
TI[0:63]Hf
mux
IWord[0:63]Hf
NOTE: ICPendHf, Fav[RT]Hf. and [RT][Fif,Seq]ZerHf must change on cycle immediately following NewHf, else logic can break
UseTHf
[12]
[13]
[16]
[28]
[25]
TFifZerHf
TSeqZerHf
ICPendHf
BlockTH
TRdyHf
IPHf
TRdyHf
FavTHf
UseTH
ResetHf
NewHf
TDeferH
RDeferH
[12] "pfif"
[13] "pseq"
[16] "pic"
[28] "meta"
[25] "defer"
RFifZerHf
RSeqZerHf
BlockRH
RRdyHf
RRdyHf
FavRHf
UseRH
NOTE: UseRH and UseTH are mutually exclusive
FavRHf
FavTHf
ICPendHf
ResetHf
TEmptyHf
REmptyHf
TReadHf
RReadHf
WantTH
WantRH
ReadHf
TReadLf
RReadLf
SemaCntrs
NewHf
6
RFifZerHf
TFifZerHf
RSeqZerHf
TSeqZerHf
SemaOvfHf
VRSeqOutHf
VTSeqOutHf
PadIn Latch LVTTL
PadOut Latch LVTTL
VRSeqInH
TSeqInH
VRSeqInHf
VTSeqInHf
SemaOvfH
VRSeqOutH
VTSeqOutH
RTCntl



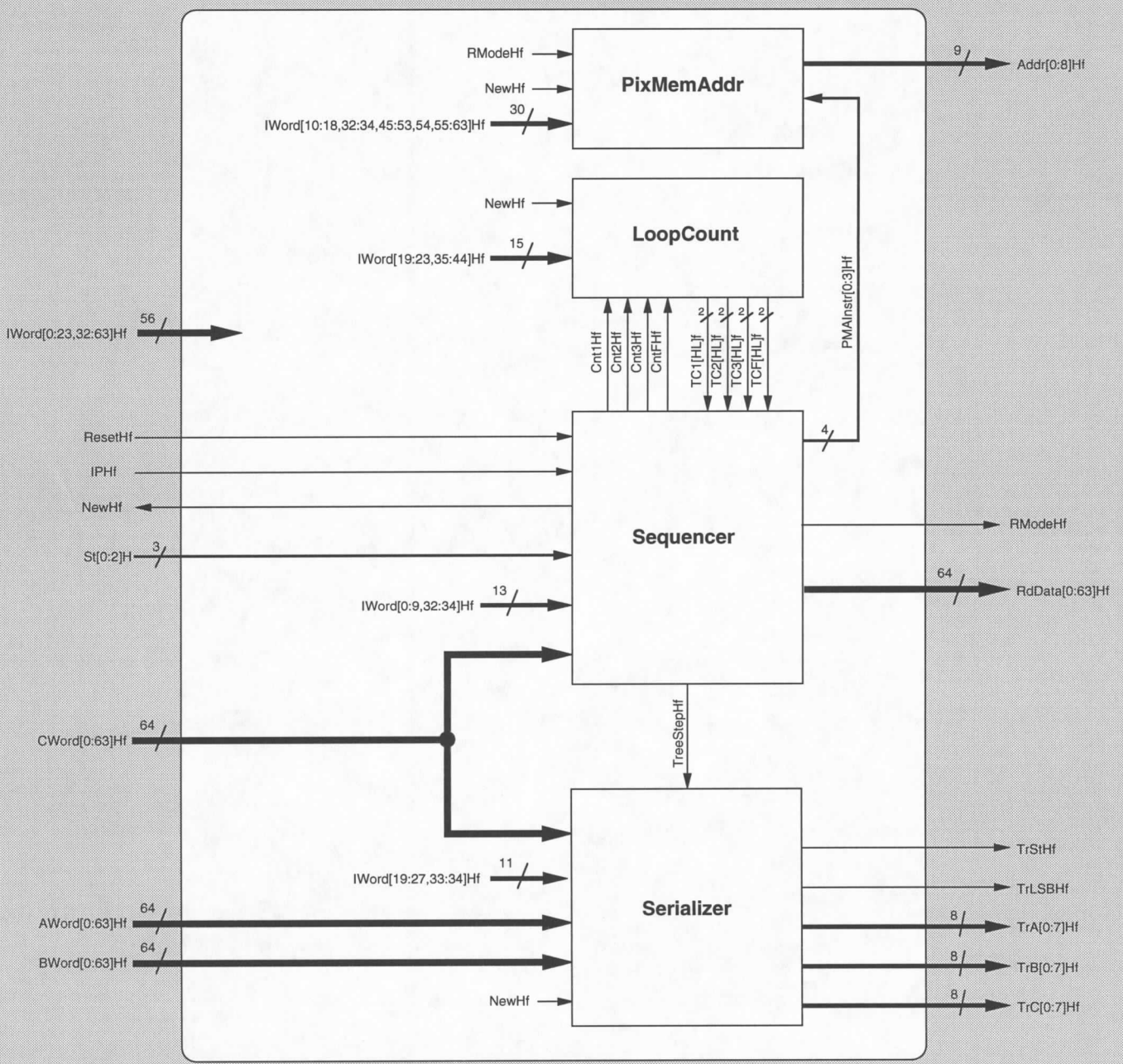
SemaCntrs
(submodule of RTCntl)



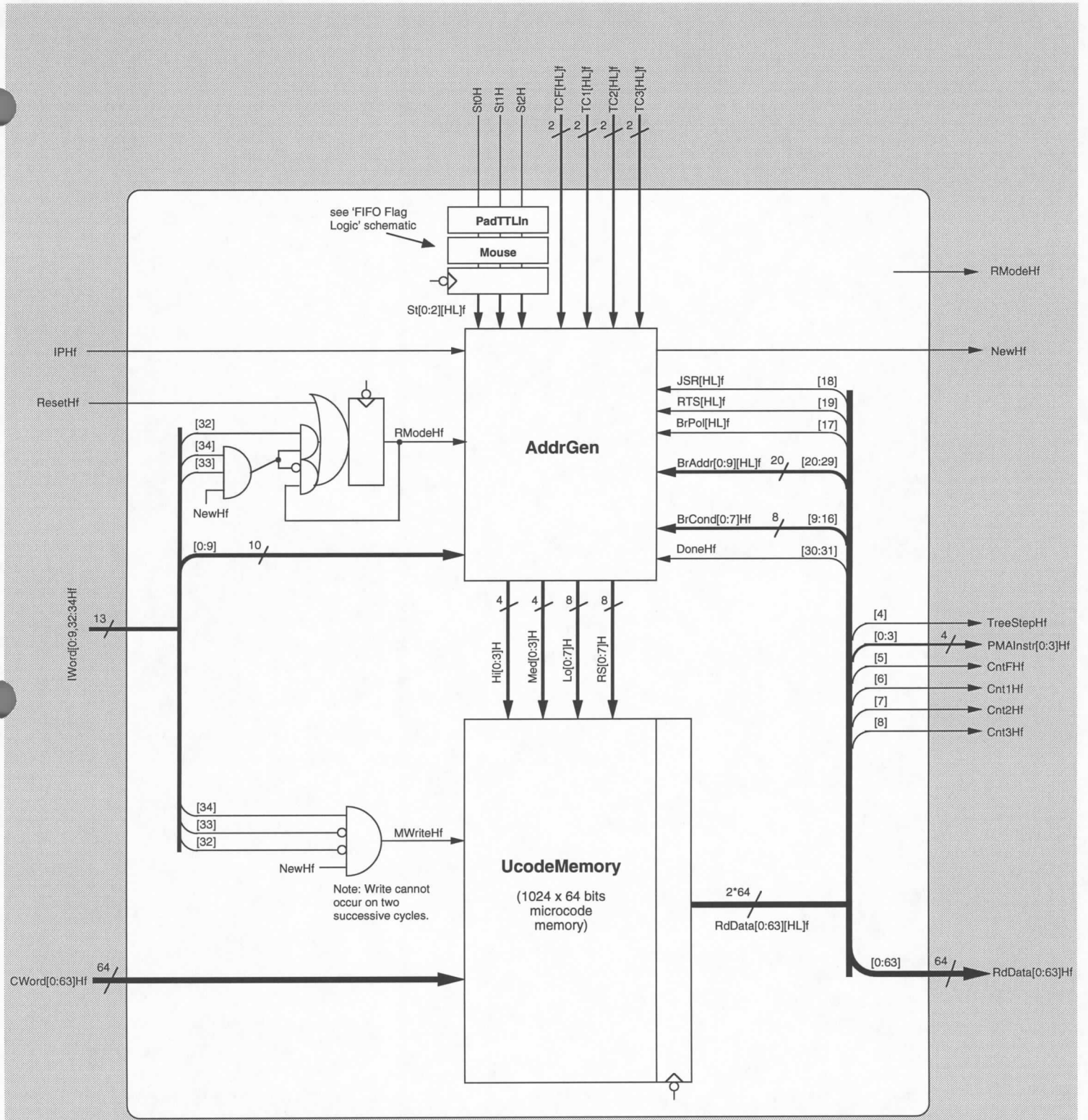
SemaCntr
(submodules of SemaCntrs)


ResetHf
to SR inputs of latches below
NewHf
[28]
DoMetaHf
System initialization sequence:
1) assertResetHf for at least two cycles
2) set AliveHf and *EndHf registers
— at least one board must be Alive
— at least one Alive board must have LeftEndHf set
— at least one Alive board must have RightEndHf set
— LeftEnd board must be to left of (or same as) RightEnd board
3) sync GPs, wait at 2 * number_of_boards cycles, before attempting to initiate any transfers
IC[L2R,R2L]St[1:0]H codes:
00 - nothing happening
01 - we're holding things up
10 - we're pending, but not running (we're being held up)
11 - we're compositing
Visible in JTAG config register:
0 - RightEndHf
1 - LeftEndHf
2 - L2RFirstHf
3 - R2LFirstHf
4 - L2RLastHf
5 - R2LLastHf
6 - L2RPassHf
7 - R2LPassHf
ICCntlPath (R2LCntl)
[24]
[39]
[40]
[41]
LE SR
AliveHf
LeftEndHf
RightEndHf
17:18
IWord[24 28]Hf
39:41
45:51
55:61
Note: Counter actually contains value-1, to make zero-detect easier. PE overrides CE.
PendHf
RunHf
LastHf
AliveHf
RdyInHf
PendHf
[1]
[0]
LVTTL OUT
ICL2RSt[0:1]H
ResetHf
AliveHf
GoOutHf
PendHf
RunHf
R2LRunHf
GTL OUT
ICR2LH
ICR2LL
[0:6]
[7:11]
CE
D Q
PE
12
[57:61]
R2LBytes[0:4]Hf
5
NOTES:
ICPendHf must go hi immediately after DoMeta, else RTCntl logic breaks.
ICPendHf is computed synchronously, with a separate latch (for speed).
[18]
R2LArmHf
ArmHf
[55]
[56]
R2LFirstHf
R2LLastHf
R2LPendHf
ICPendHf
L2RPendHf
(this portion in pad-ring area)
PassHf
RRdyOutHf
RRdyOutHr
RdyOutHr
LastHf
R2LLastHf
RdyInHf
LRdyInHf
ICLRdyH
PadBiDir
LeftEndHf
LGoOutHr
GoOutHr
FirstHf
R2LFirstHf
GoInHf
RGoInHf
LGoOutHf
ICRRdyH
RightEndHf
ICLGoH
For IC[LR]RdyH and IC[LR]GoH bi-directional pins, input data must be stable about Clk-falling, output data changes caused by Clk-rising.
Note: DatInHr outputs of pads are not used.
ICRGoH
LGoInHf
GoInHf
GoOutHr
RGoOutHr
GoOutHf
RGoOutHf
LRdyOutHf
LRdyOutHr
RdyOutHf
RdyOutHr
RdyInHf
RRdyInHf
[17]
L2RArmHf
ArmHf
PendHf
L2RPendHf
[45]
[46]
L2RFirstHf
L2RLastHf
FirstHf
LastHf
ICCntlPath (L2RCntl)
RunHf
L2RRunHf
GTL OUT
ICL2RH
ICL2RL
AliveHf
ResetHf
[47:51]
L2RBytes[0:4]Hf
NBytes[0:6]Hf
St[0:1]Hf
LVTTL OUT
ICR2LSt[0:1]H

ICCntl

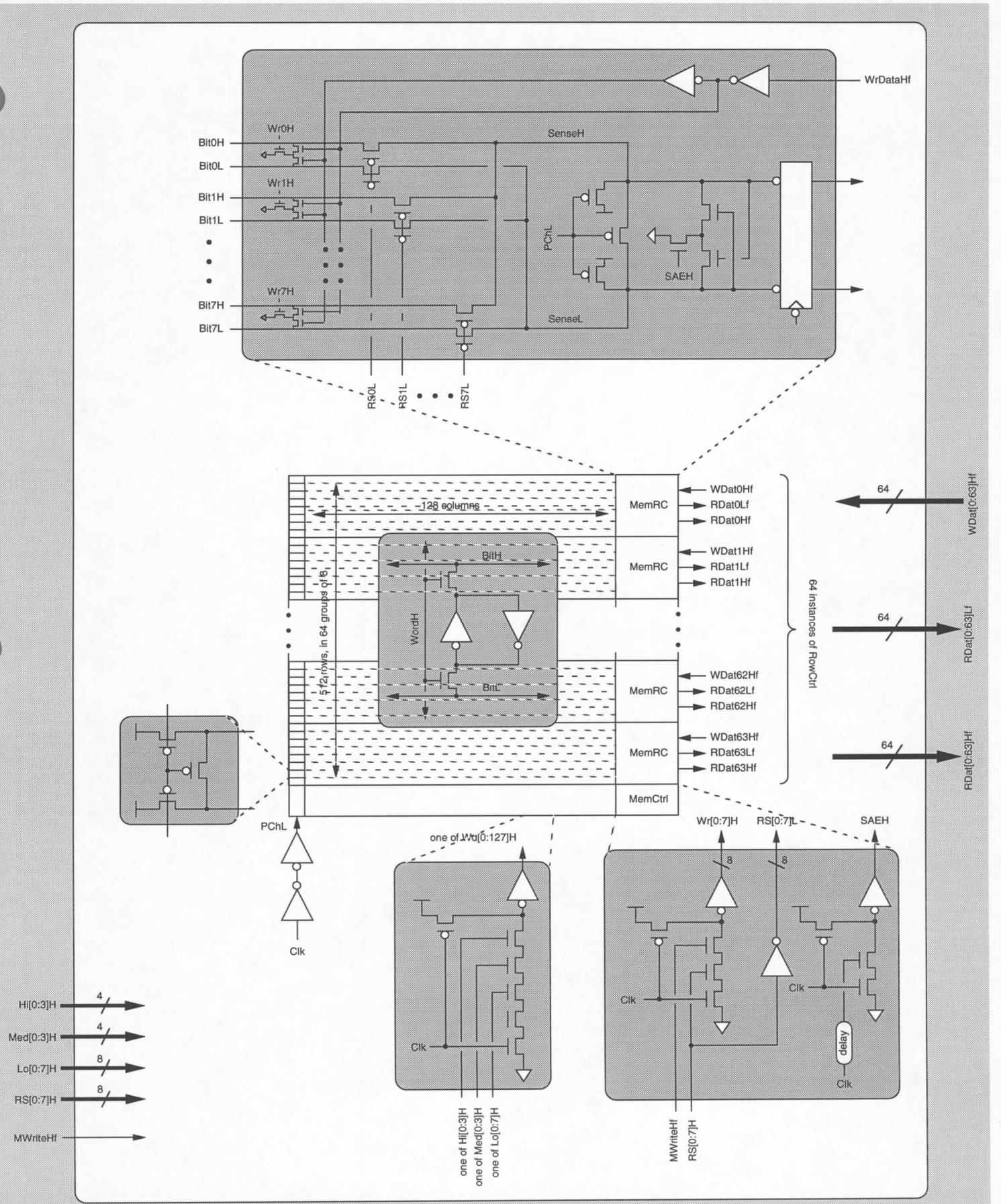

RModeHf
NewHf
30
IWord[10:18,32:34,45:53,54,55:63]Hf
PixMemAddr
9
Addr[0:8]Hf
NewHf
15
IWord[19:23,35:44]Hf
LoopCount
PMAInstr[0:3]Hf
56
IWord[0:23,32:63]Hf
Cnt1Hf
Cnt2Hf
Cnt3Hf
CntFHf
2
TC1[HL]f
TC2[HL]f
TC3[HL]f
TCF[HL]f
ResetHf
IPHf
NewHf
St[0:2]H
3
4
Sequencer
RModeHf
64
RdData[0:63]Hf
13
IWord[0:9,32:34]Hf
64
CWord[0:63]Hf
TreeStepHf
11
IWord[19:27,33:34]Hf
Serializer
TrStHf
TrLSBHf
64
AWord[0:63]Hf
8
TrA[0:7]Hf
64
BWord[0:63]Hf
8
TrB[0:7]Hf
NewHf
8
TrC[0:7]Hf

Cntl



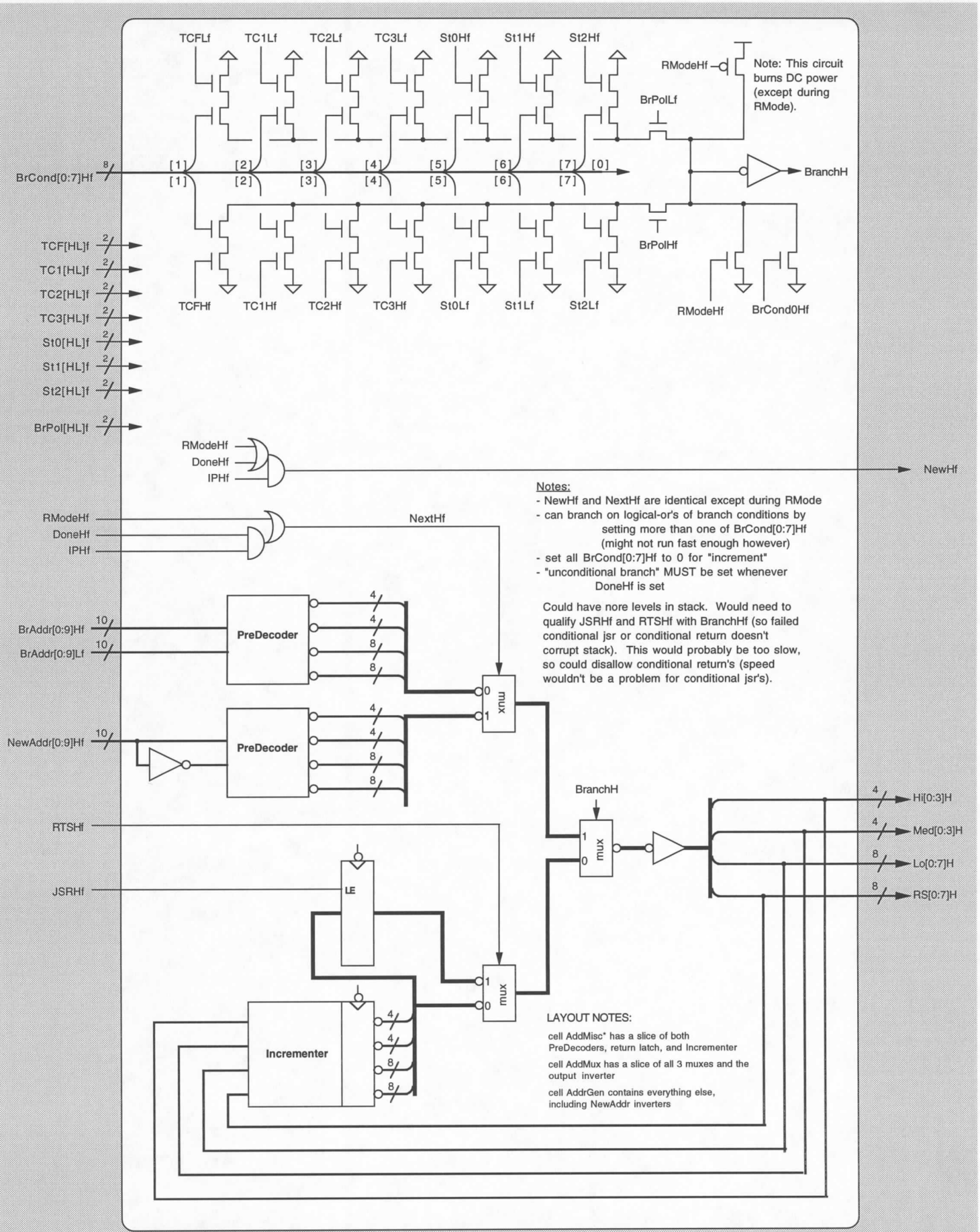
Sequencer

(submodule of Cntl)



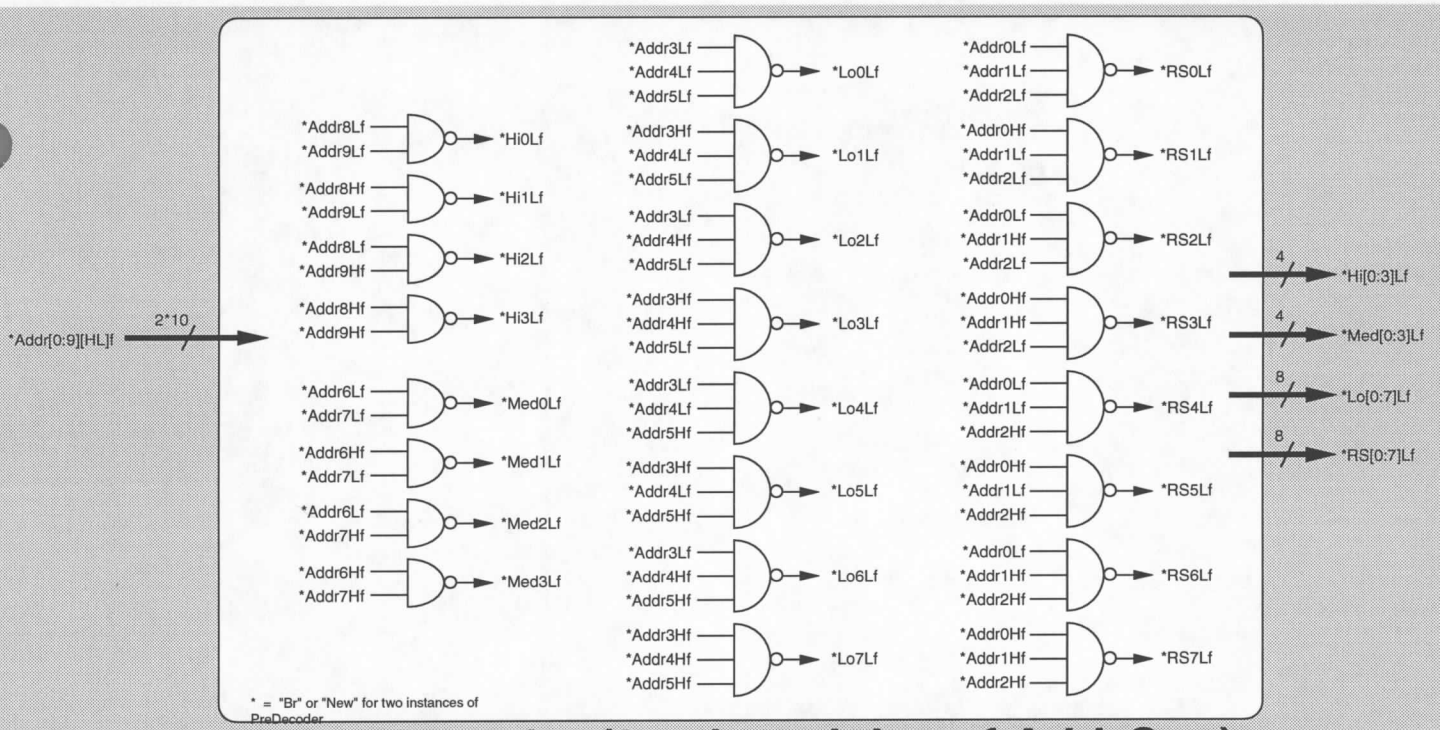
UcodeMemory

(sub-module of Sequencer)

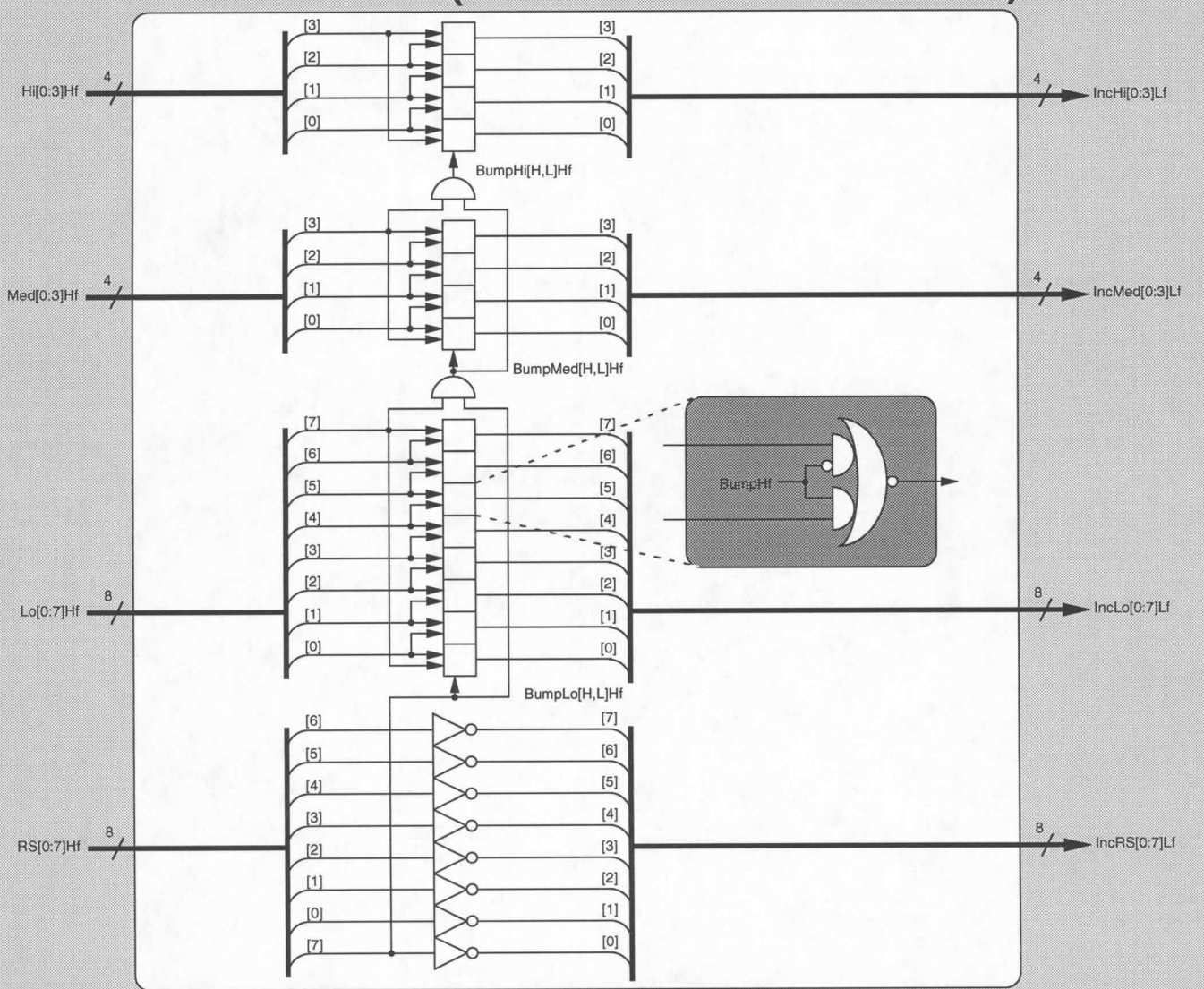


AddrGen

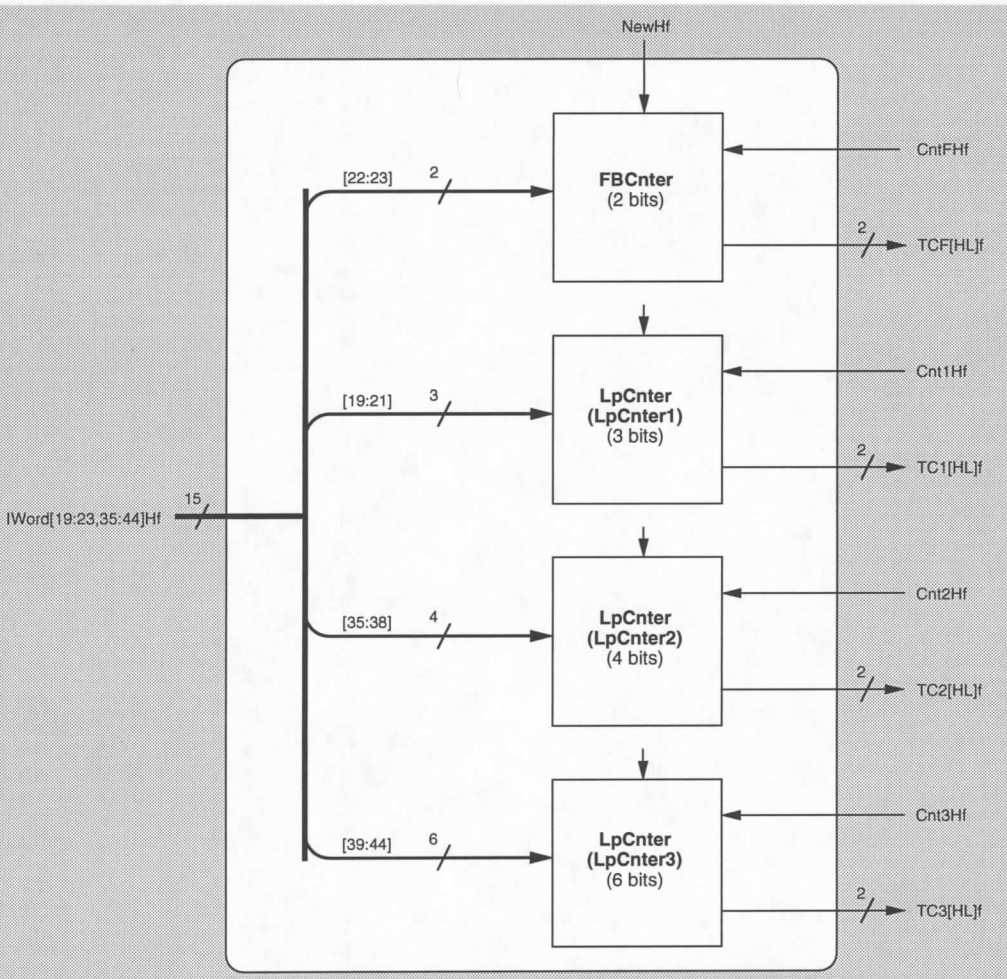
(submodule of Sequencer)



PreDecoder (2 submodules of AddrGen)

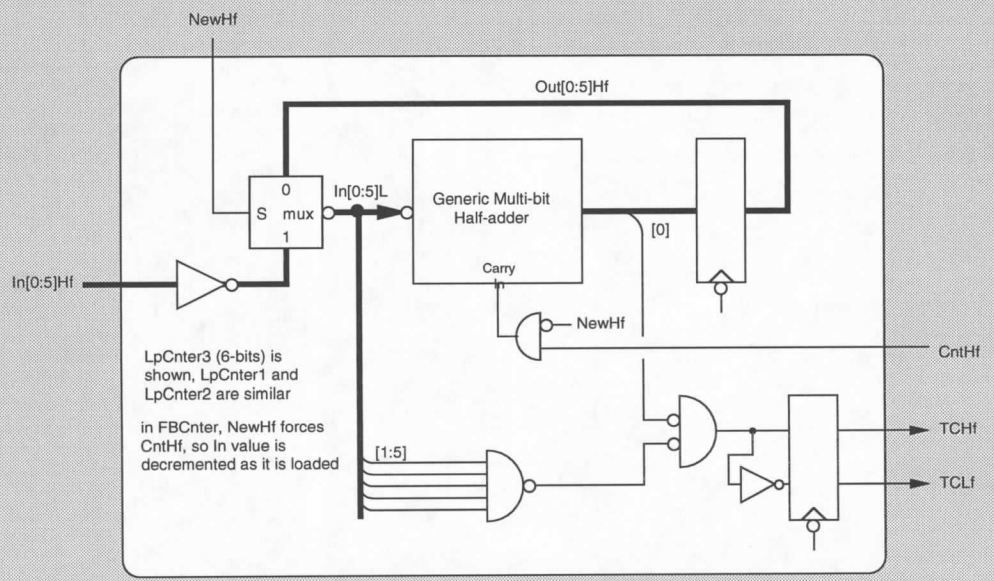


Incrementer (submodule of AddrGen)



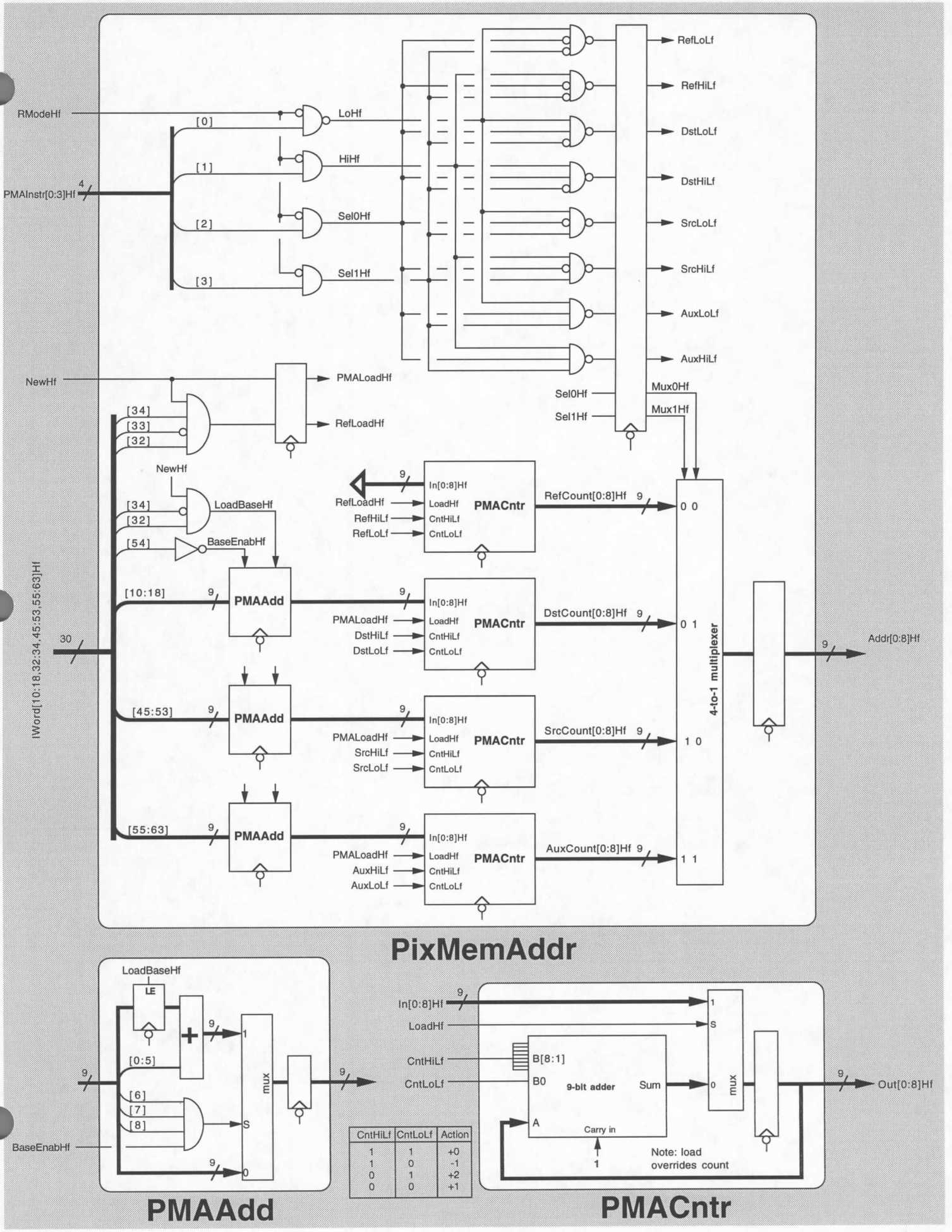
LoopCount

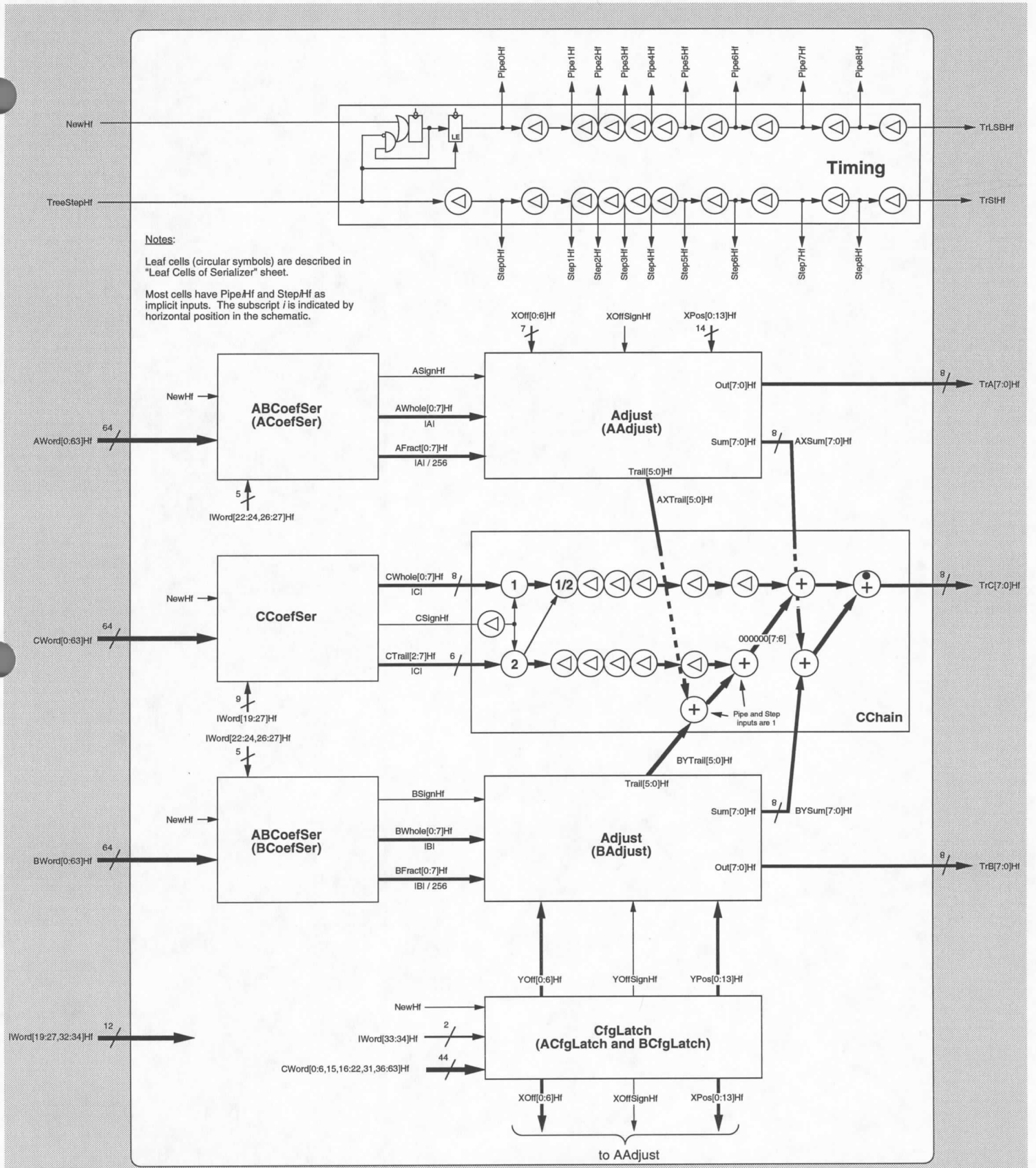
(submodule of Cntl)



LpCnter

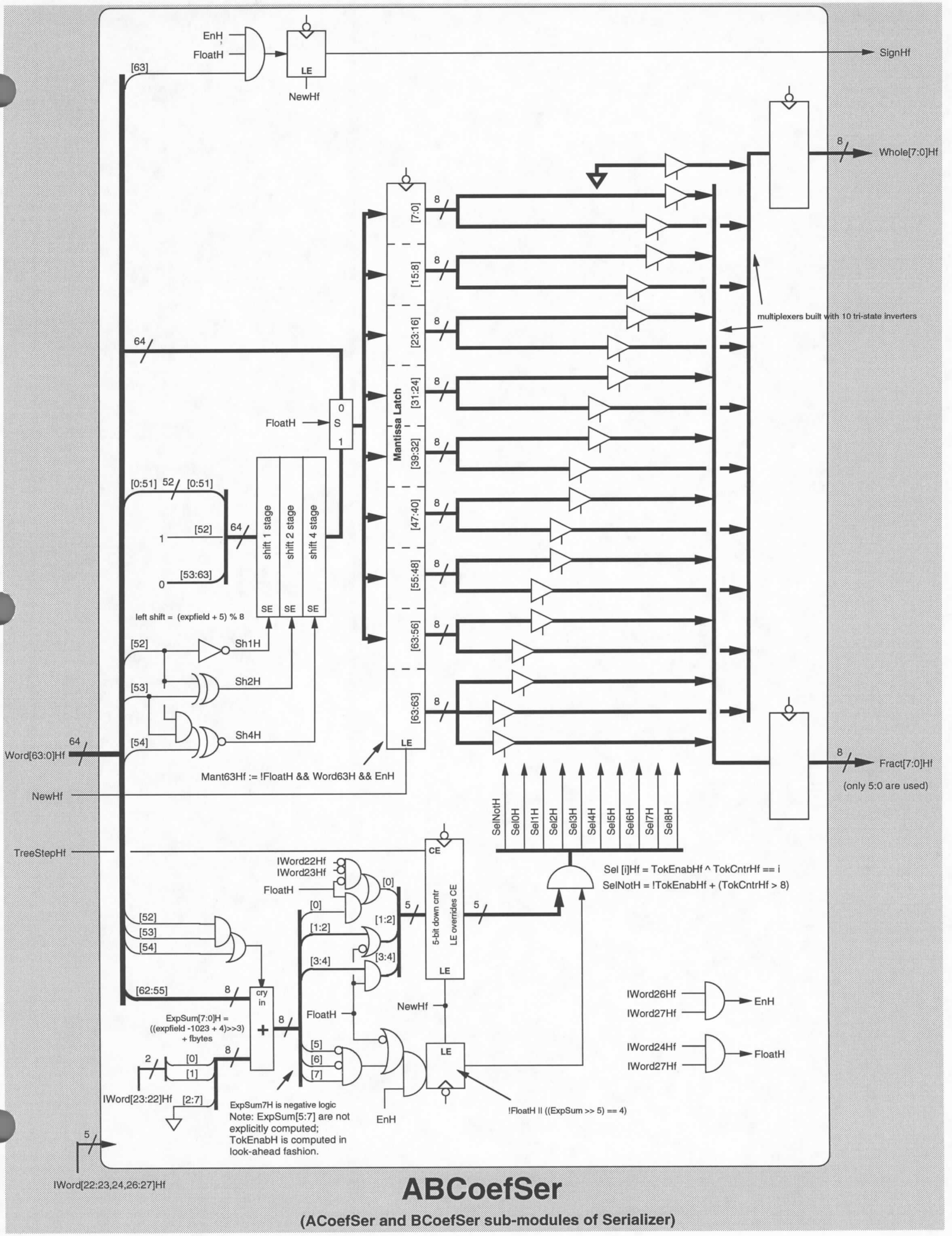
(4 submodules of LoopCount)


RModeHf
PMAInstr[0:3]Hf
4
[0]
[1]
[2]
[3]
LoHf
HiHf
Sel0Hf
Sel1Hf
RefLoLf
RefHiLf
DstLoLf
DstHiLf
SrcLoLf
SrcHiLf
AuxLoLf
AuxHiLf
NewHf
PMALoadHf
RefLoadHf
[34]
[33]
[32]
Sel0Hf
Sel1Hf
Mux0Hf
Mux1Hf
NewHf
LoadBaseHf
[34]
[32]
[54]
BaseEnabHf
9
In[0:8]Hf
LoadHf
CntHiLf
CntLoLf
PMACntr
RefLoadHf
RefHiLf
RefLoLf
RefCount[0:8]Hf 9
0 0
[10:18]
PMAAdd
PMALoadHf
DstHiLf
DstLoLf
DstCount[0:8]Hf 9
0 1
30
IWord[10:18,32:34,45:53,55:63]Hf
[45:53]
PMAAdd
PMALoadHf
SrcHiLf
SrcLoLf
SrcCount[0:8]Hf 9
1 0
[55:63]
PMAAdd
PMALoadHf
AuxHiLf
AuxLoLf
AuxCount[0:8]Hf 9
1 1
4-to-1 multiplexer
9
Addr[0:8]Hf
PixMemAddr
LoadBaseHf
LE
+
9
1
[0:5]
[6]
[7]
[8]
S
mux
0
BaseEnabHf
9
In[0:8]Hf
LoadHf
CntHiLf
CntLoLf
B[8:1]
B0
9-bit adder
Sum
A
Carry in
1
Note: load
overrides count
S
mux
Out[0:8]Hf
CntHiLf | CntLoLf | Action
1 | 1 | +0
1 | 0 | -1
0 | 1 | +2
0 | 0 | +1
PMAAdd
PMACntr



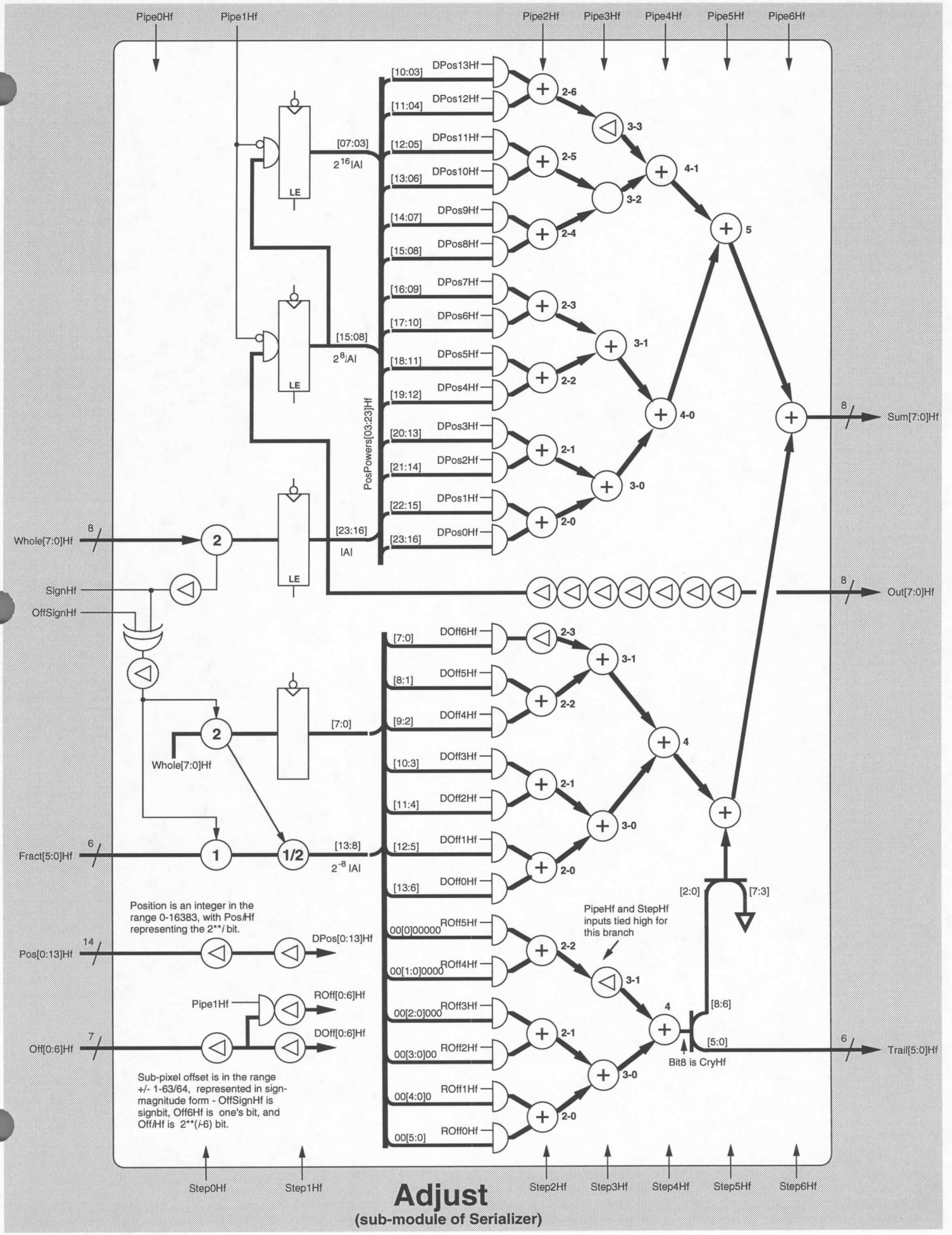
Serializer

(submodule of Cntl)



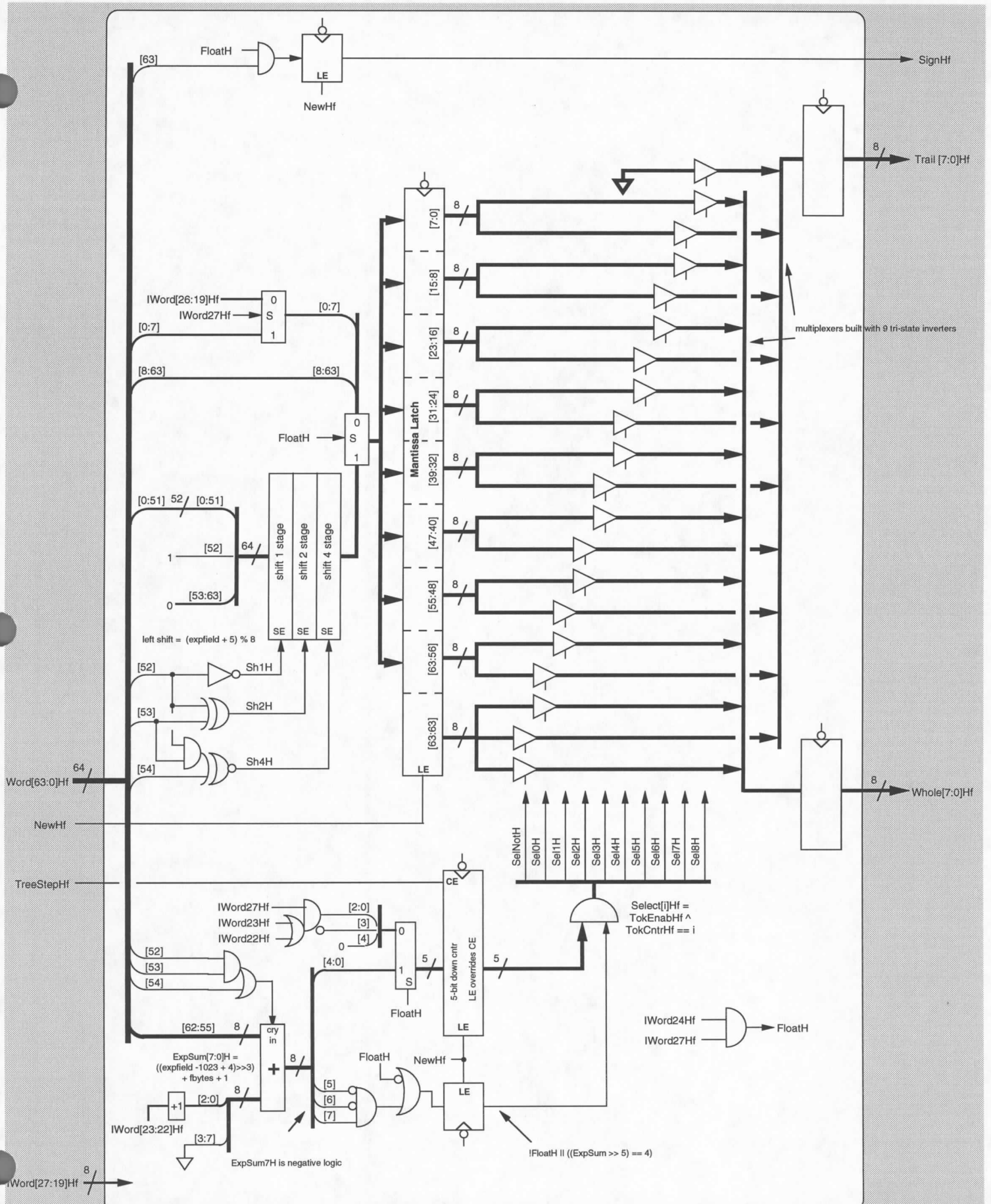
ABCoefSer

(ACoefSer and BCoefSer sub-modules of Serializer)



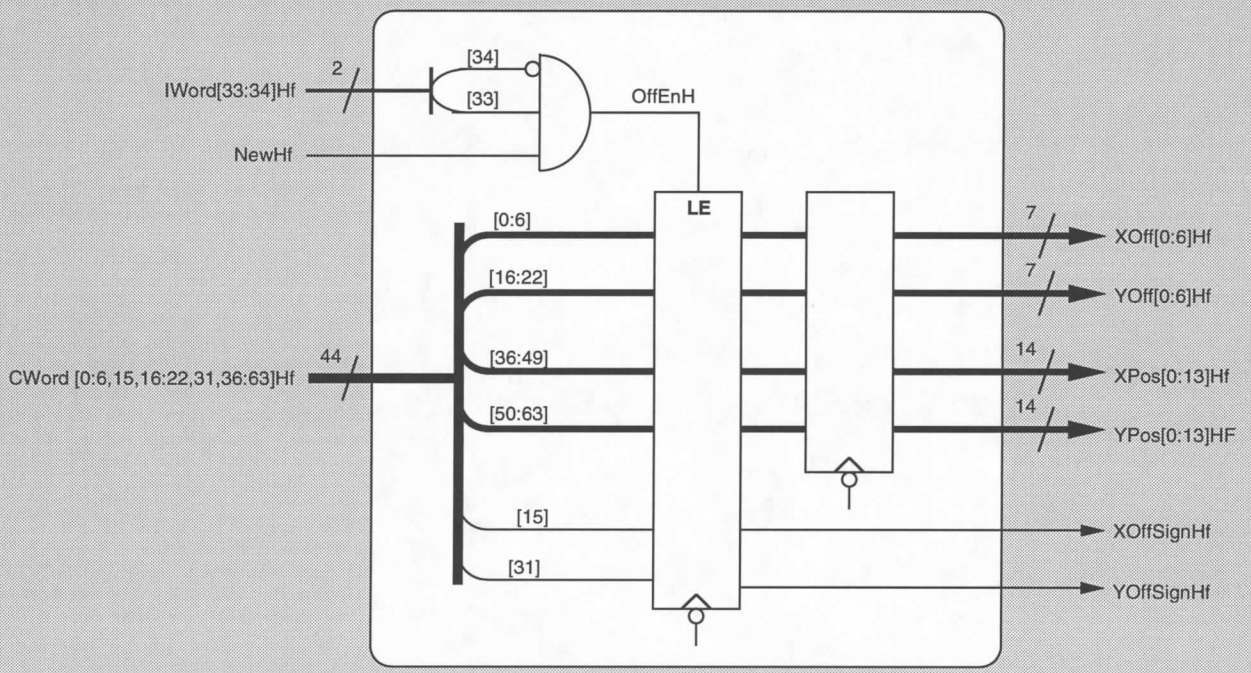
Adjust

(sub-module of Serializer)



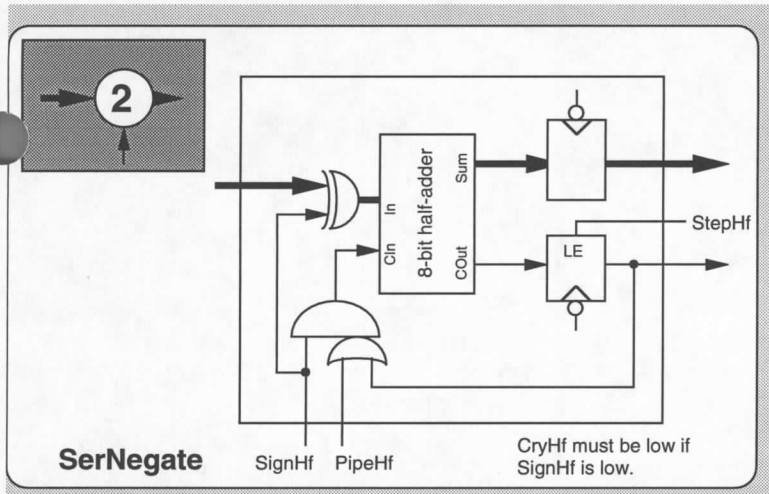
CCoefSer

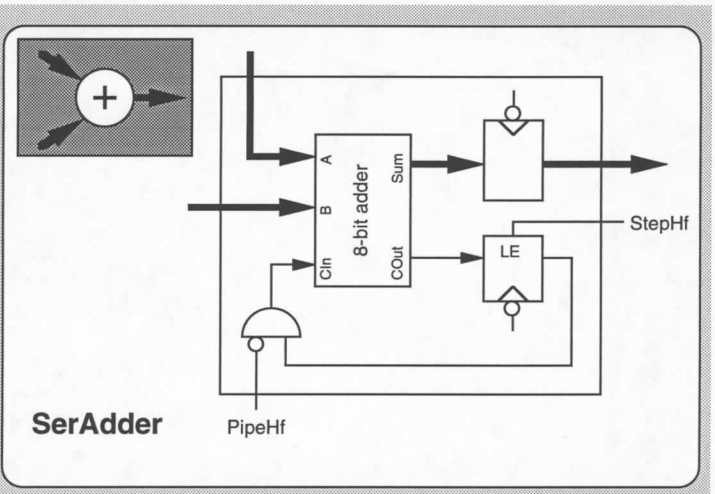
(submodule of Serializer)

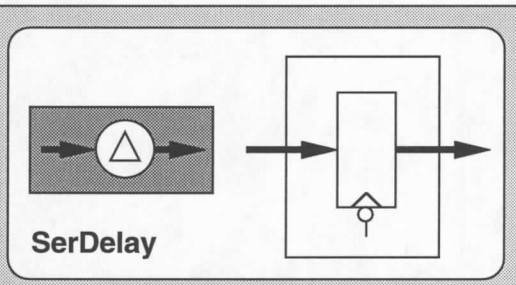


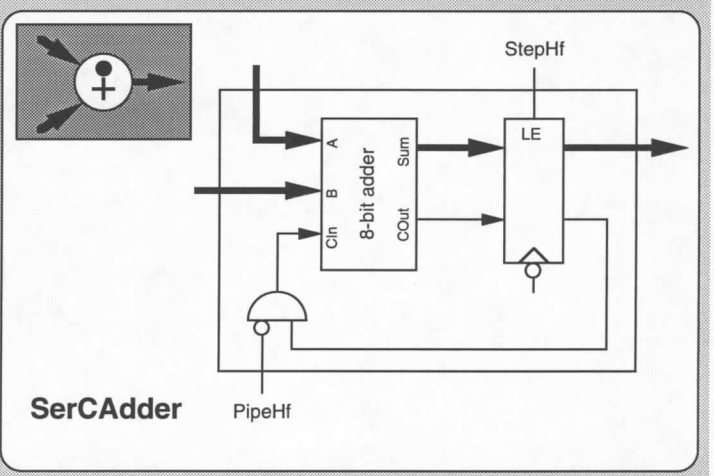
CfgLatch

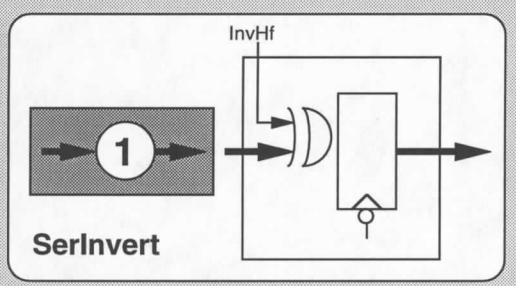
(submodule of Serializer)

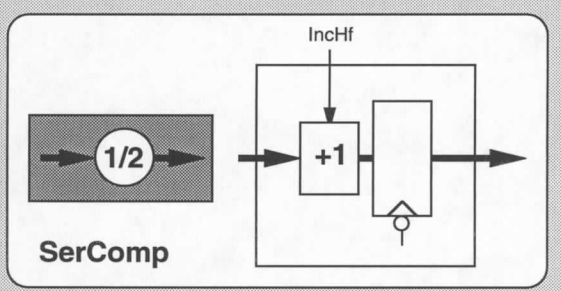




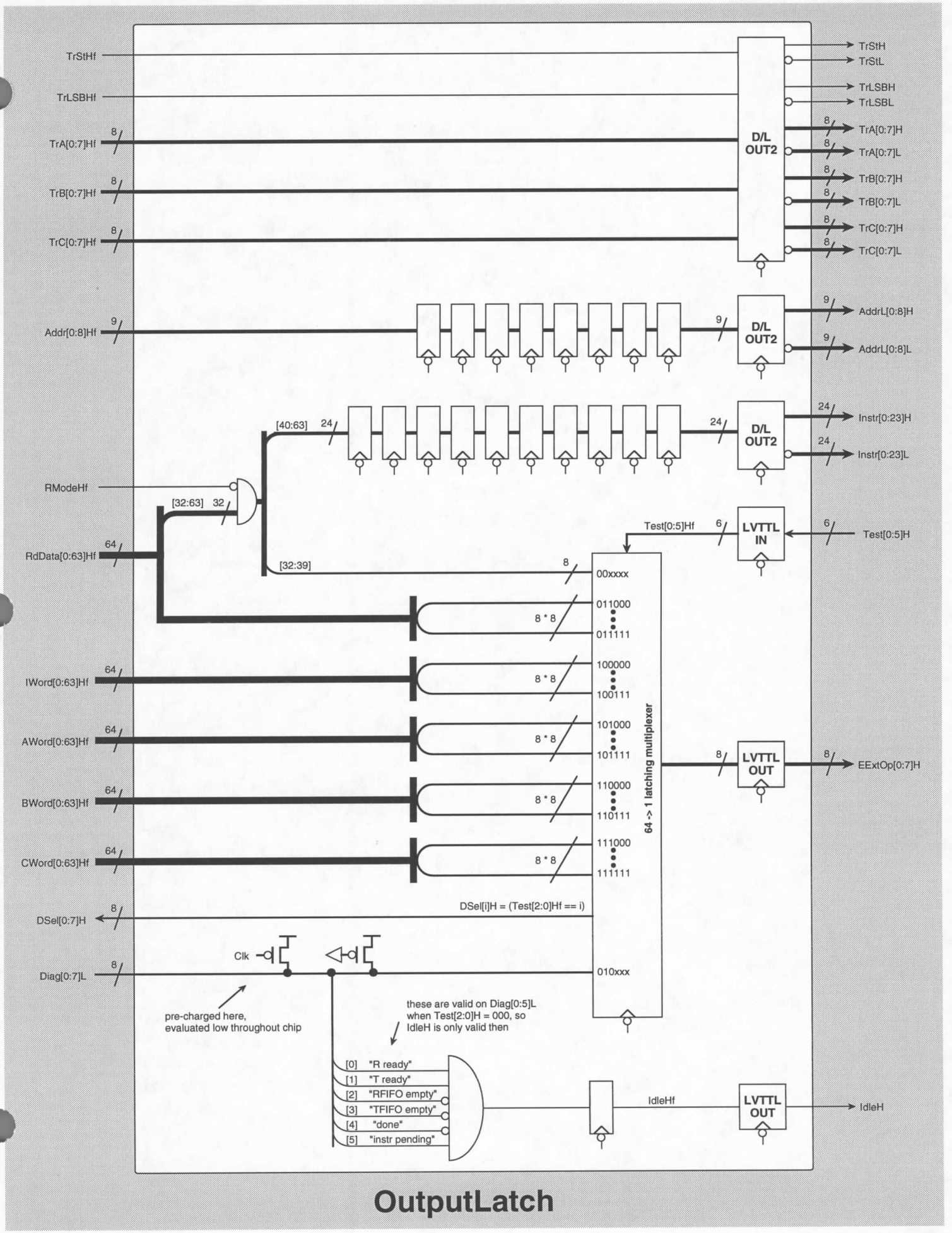




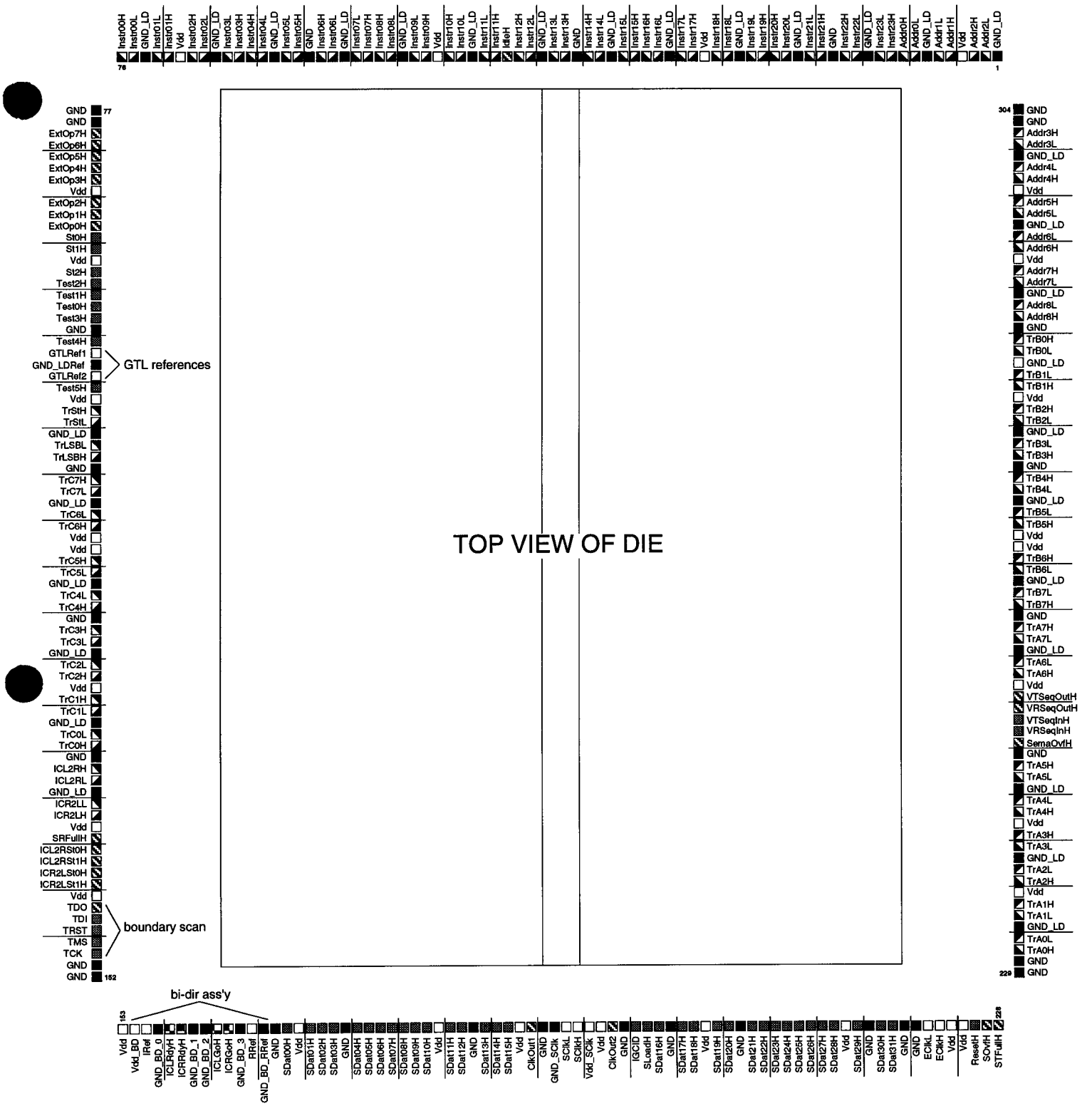




Leaf Cells of Serializer



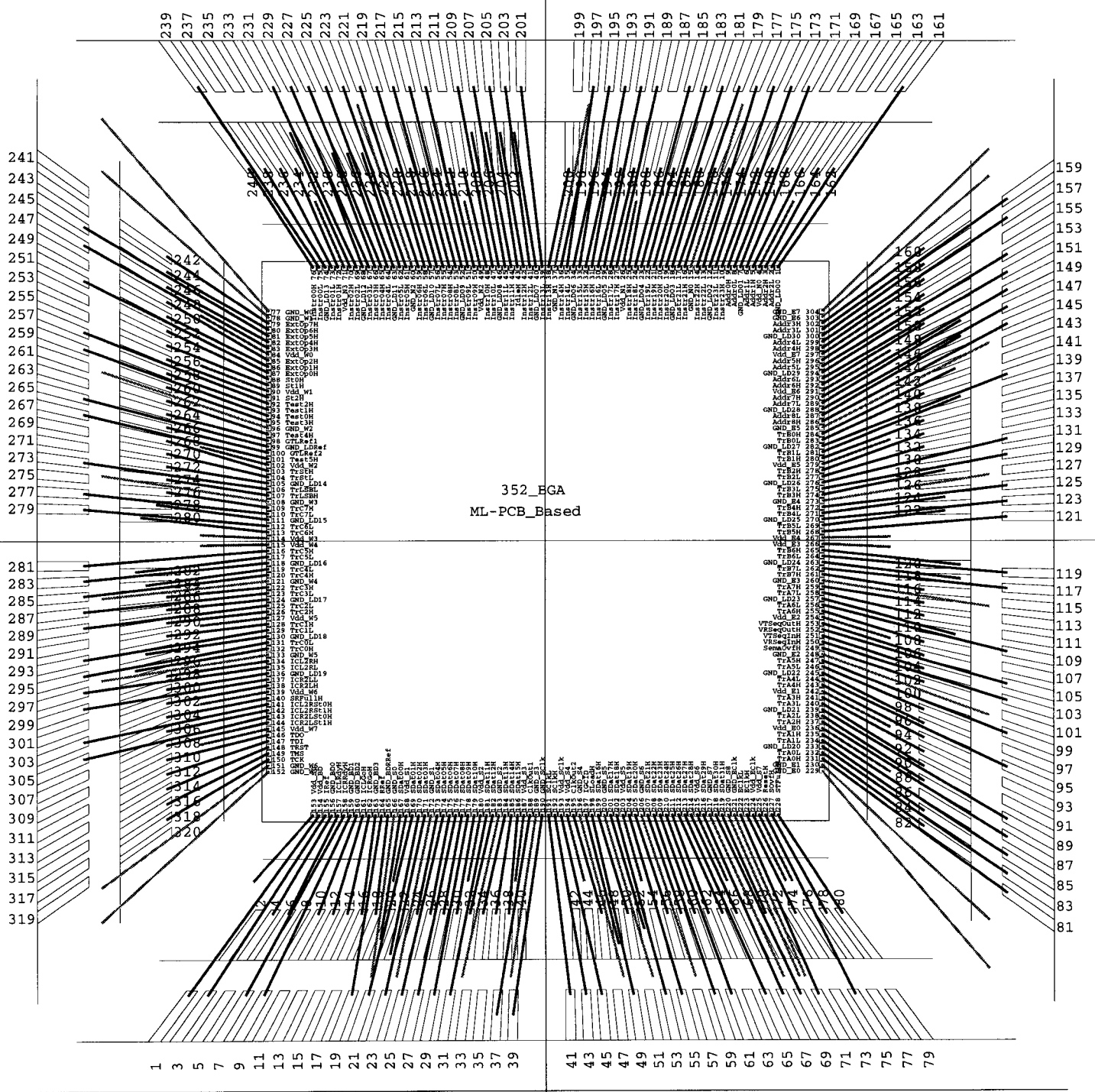
OutputLatch



Pinout for PixelFlow Image Generation Controller

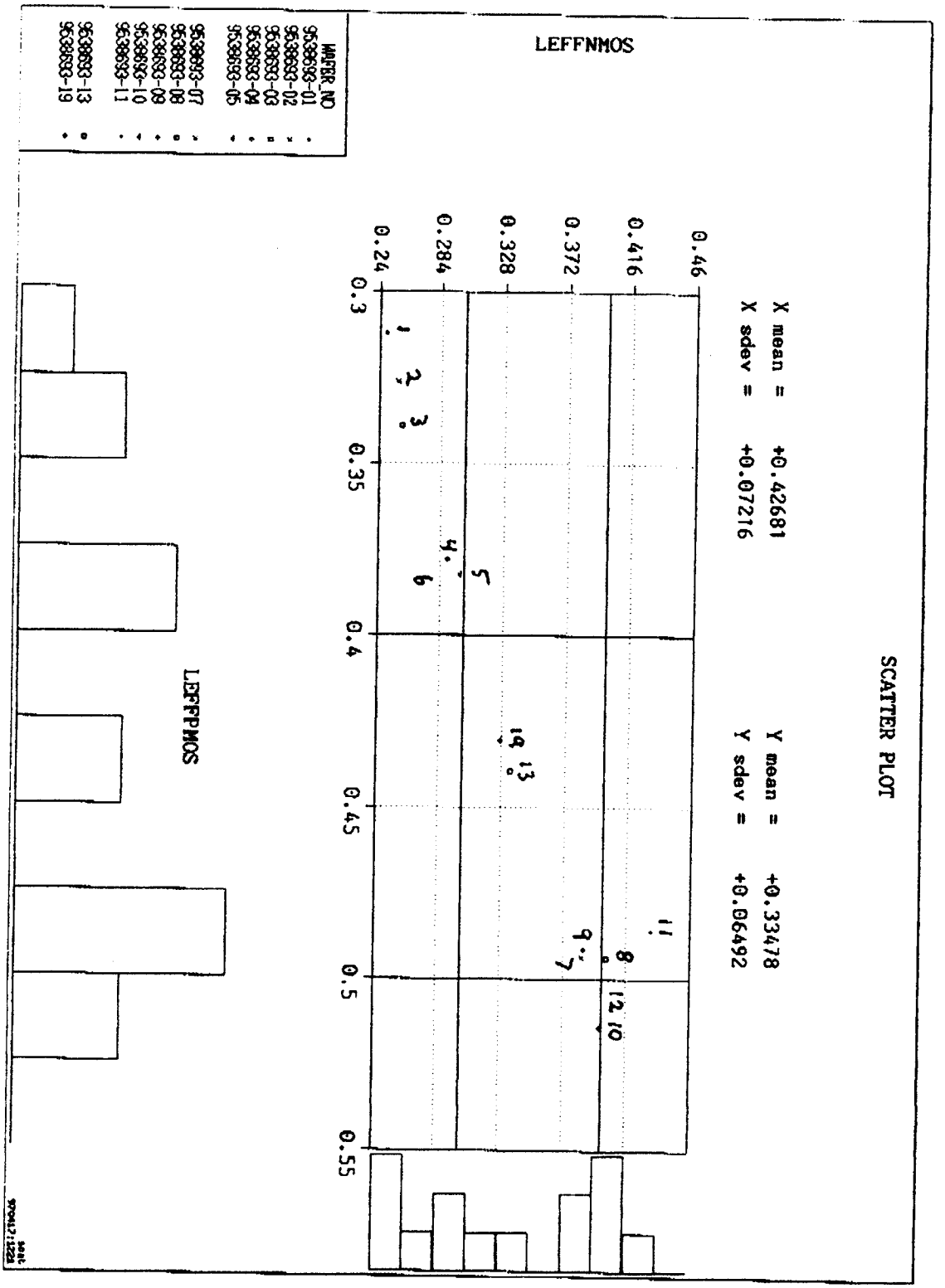
jge/24-Aug-95

PIXELFLOW EGC BONDING DIAGRAM



SCATTER-PLOT FOR 1ST IGC RUN

TO: JOHN EYLES



IGC LFET 9538693

TOTAL P.01

IV.2.5 IGC MICROCODE ASSEMBLER

The sequencers on the IGC are programmed using the IGC microcode asemblers, *asmEMC* and *asmTAS*. Each assembler processes an input file *<root>.ucode*, containing microcode source described below, and generates three output files: *<root>_microcode.h*, defines an array *static int root_microcode[]* containing the data to be loaded into the sequencer microcode store. *<root>_opcodes.h*, contains macro definitions which generate command opcodes for specific instructions. The third, *<root>_commands.h*, contains inline C++ functions built on top of the *<root>_opcodes.h* macros, which allow the user to specify an IGC command with a single function call. The file *<root>_microcode.h* is included in application code which initializes the IGC; *<root>_commands.h* and *<root>_opcodes.h* are included in application code which generates IGC commands. *AsmEMC* is used to generate microcode for the EIGC (which generates micro-instructions for the EMC array), and *asmTAS* is used to generate microcode for the TIGC (which generates micro-instructions for the TASICs and connected texture/frame memory).

EMC Instructions

EMC instructions may be divided into 2 basic types:

1) those which do not use the linear expression evaluator
2) those which use the linear expression evaluator, in one of three ways:
    (a) a one byte 'mask' field is passed down as C, tree compute F(x,y) = C
    (b) tree computes F(x,y) = C from supplied C coefficient
    (c) tree computes F(x,y) = Ax + By + C from supplied ABC coefficients

For instruction types 2a and 2b, the coefficients may be of 4 types:

1) 32-bit integers
2) 64-bit integers
3) 32-bit single-precision floats
4) 64-bit double-precision floats ('doubles' in C)

Input File for Microcode Assembler.

Each line in the input file falls in one of four categories:

(1) a template for specifying the opcode for an instruction

(2) a specification of the rules the arguments to an arguments must obey
(3) a microcode word
(4) comment lines

Lines beginning with '%' define an instruction, and a template for building the specific opcode. These lines are interlaced with the microcode word specification lines, so that the placement of an opcode specification line relative to the microcode word specification lines defines the entry point for the instruction (unless another entry point is explicitly specified as described below). Multiple instructions may use the same entry point.

Lines beginning with '^' define rules for the arguments to an instruction (the arguments in the opcode specification line). Each rule applies to all subsequent opcode specifications, until a new rule is given.

All other lines (except comments) define microcode words. The microcode words are assigned to microcode store locations in ascending order, starting with address 0, with a 1-1 correspondence between input lines and microcode store locations.

Any portion of a line lying to the right of a semi-colon (';') (including lines beginning with a semi-colon, and empty lines) is treated as a comment.

Opcode specification.

Opcode template lines begin with the special character '%'. They are of the form:

```
%NEM(arg1,arg2,...) [MCAddr:addr] [DstAddr:dst] [SrcAddr:src] [AuxAddr:aux] [nobase]
                    [LpCnt1:lpcnt1] [LpCnt2:lpcnt2] [LpCnt3:lpcnt3] [FBCnt:fbcnt]
                    [con | lin] [int32 | int64 | flt32 | flt64]
                    [FBytes:fbytes] [mask:mask] [A:avalue] [B:bvalue] [C:cvalue]
                    [meta [ignore] [maux] [pseq] [pfif] [vseq] [vfif] [pic] [grab] [yield]
                    [loadmac [alive:ALIVE] [leftend:LEFTEND] [rightend:RIGHTEND] ]
                    [l2rarm:L2RARM [l2rfst:L2RFST] [ l2rlst:L2RLST] [l2rbytes:L2RBYTES] ]
                    [r2larm:R2LARM [r2lfst:R2LFST] [ r2llst:R2LLST] [r2lbytes:R2LBYTES] ]
                    [cfg:CFG] [Ronly] [Tonly]
```

NEM is a mnemonic name for the instruction, and *arg1*, *arg2*, etc. are a set of optional mnemonics for instruction arguments. The remaining fields are also optional. There must be no spaces in the list *arg1*, *arg2*, ...

MCAddr specifies an alternate entry point for the instruction.

DstAddr, *SrcAddr*, and *AuxAddr* specify starting values to be used for the three 9-bit pixel-memory address counters. Thus 3 separate pixel-memory operands (3 sequences of pixel-memory addresses) can be used in an instruction. The token *nobase* causes the base-offset register to be inhibited.

LpCnt1, *LpCnt2*, and *LpCnt3* are starting values to be used for the loop counters used by the microcode sequencer. *FBCnt* specifies the starting value for the FBCnter, a 2-bit counter. *FBCnt* may be specified only for instructions which do not use coefficients or the one-byte mask value.

FBytes specifies the number of fractional bytes for floating-point or fixed-point coefficients. When the tree result is clocked out of the LEE, the first *fbytes* bytes will represent the fractional portion of the LEE result. In addition, the FBCnter is loaded with the value ((*fbytes*-1) modulo 4). (For *FBCnt*, the value put in the opcode field is incremented, so this decrement is neutralized). A tight loop on the FBCnter is used to clock out the fractional bytes; note that the FBCnter will be loaded with 3 when *fbytes*=0, so this case cannot be handled in the loop.

The *con* and *lin* tokens are mutually exclusive and indicate that the instruction includes linear expression evaluator coeffcients, either just C, or A, B, and C, respectively.

Together, the *con*, *lin*, *int32*, *int64*, *flt32*, and *flt64* tokens indicate whether or not the instruction uses the linear expression evaluator result, the mode of the LEE, and the type of coefficients to be used.

The presence of the *mask:* token means that the instruction uses a special one-byte value as the C coefficient and that the LEE works in constant mode. The first byte of the LEE result is just the specified byte, and any additional bytes are garbage (so only one byte should be used).

The *A:avalue*, *B:bvalue*, and *C:cvalue* tokens indicate that the specified coefficient should be set to a constant value or a value which is an expression involving the arguments. (If *int64* is also specified, the coefficient expression must be of the form *lo@hi*, where *lo* and *hi* are expressions for the low and high-order 32-bits of the 64-bit integer coefficient).

The arguments *dst*, *src*, *aux*, *lpcnt1*, *lpcnt2*, *lpcnt3*, *fbcnt*, and *fbytes* must be constant expressions involving the 'args' which are interpretable by the C pre-processor, and should

use parentheses liberally to avoid incorrect evaluation if the 'args' are complex expressions in invocations of the instructions. Valid ranges for the actual arguments passed in the opcode are as follows:

FIELD	ARGUMENT	MIN	MAX
MCAddr	addr	0	1023
DstAddr	dst	0	255
SrcAddr	src	0	255
AuxAddr	aux	0	255
LpCnt1	lpcnt1	0	8 †
LpCnt2	lpcnt2	0	16 †
LpCnt3	lpcnt3	0	64 †
FBytes	fb	0	3

† range is 0 to MAX-1 or MIN+1 to MAX for any given instruction, depending on the how the microcode is written

<< DISCUSS THE PMA BASE_REGISTER OFFSET FUNCTION >>

For each opcode specification line, *asmEMC* places a macro definition in the output file *EMC_opcodes.h* of the form

I_NAME(args) ...

This macro defines the low 32-bit word of the opcode. If the instruction requires the long 64-bit opcode (if any of the tokens *LpCnt2:*, *SrcAddr:*, or *AuxAddr:* are used) *asmEMC* generates a second macro of the form

P_NAME(args) ...

which defines the high 32-bit word of the opcode.

Rules specification.

Since the specification for the arguments generally are expressions involving the 'args', the assembler cannot do error checking; rather, the code generating IGC commands must insure that valid values will result. Failure to detect errors can result in the IGC hanging. The microcode assembler provides some assistance in providing valid arguments. For each

nstruction, a "rule" can be specified, which is a test included in the in-line function call, and applied at run-time (if the compiler is allowed to select this feature).

Lines specifying rules must begin with '^', followed by a logical expression (in C). The following macros are provided (in *IGCChk.h*) to aid in writing rules:

IGCMEM(lsb,len)	the specified memory segment lies wholly within one of the areas of pixel-memory
IGCRANGE(val,min,max)	*val* lies between *min* and *max*, inclusive
IGCNOVERLAP(lsb1,len1,lsb2,len2)	the two memory segments do not overlap at all
IGCNPARTIAL(lsb1,len1,lsb2,len2)	the two memory segments do not overlap, or else they have the same LSB (not necessarily the same length)
IGCMULT(val,dic)	*val* is an exact multiple of *div*

An example rule is as follows:

```
^ IGCMEM(dst,len) && IGCRANGE(len,1,8) && (tmp > 1)
```

This says that *mem[dst:len]* must be a valid memory segment, *len* must be in the range 1 through 8, and the argument *tmp* must be greater than 1.

Each rule applies to all subsequent instructions. If a rule refers to operands that do not exist for a given instruction, the generated header files will end up causing multiple errors at compile time; this must be avioded. The rule can be set back to "null", using a single line consisting of '^'.

Microcode word specification.

All other lines in the input file, except blank lines and comment lines (beginning with a semicolon ";") are specifications of microcode words, one line to a microcode word. A microcode specification lines contains a number of optional but inter-dependent fields.They are divided into several groups:

(1) sequencer control - control the sequencer branching
(2) pixel-memory control (address and read/write)

(3) linear expression evaluator function
(4) pixel-ALU instruction

Sequencer control is specified using one of the following tokens:

<default>	unconditionally increment (go to next address)
done	terminate this instruction (begin next instruction, otherwise idle)
done?	terminate this instruction if new one waiting (otherwise continue)
done:N	terminate this instruction if new one waiting (otherwise branch)
br:n	unconditionally branch to offset n (positive or negative)
bzX:N	branch if condition X is zero (continue otherwise)
bnzX:N	branch if condition X is non-zero (continue otherwise)
jsr:N	jump to subroutine (unconditionally)
jzX:N	jump to subroutine if condition X is zero (continue otherwise)
jnzX:N	jump to subroutine if condition X is non-zero (continue otherwise)
rzX	return from subroutine if condition X is zero (continue otherwise)
rnzX	return from subroutine if condition X is non-zero (continue otherwise)
rzX:N	return from subroutine if condition X is zero (branch otherwise)
rnzX:N	return from subroutine if condition X is non-zero (branch otherwise)

The branch condition X may be one of the following:

1	condition is contents of Loop Counter 1
2	condition is contents of Loop Counter 2
3	condition is contents of Loop Counter 3
f	condition is contents of FBytes counter
ext[012]	condition is external condition input St[0:2]

Complex branches, based on logical-or's of the branch conditions, are also possible; talk to Eyles for details.

The branch address, or subroutine jump address, N, is one of: =N

absolute address *N*

[+-]n	relative address *n* (offset +/- *n* from current address)
label	address specified by label from label table

Labels are specified with tokens of the form:

Label:label

The label must begin with a letter of the alphabet.

The *done* token is special. It specifies termination of the instruction; if the next instruction is available, control will jump to the starting microcode address for that instruction, otherwise control branches to 0, the idle address. For the *done?* token, if another instruction is available, control will jump to the starting microcode address for that instruction, otherwise controls jumps to the next address; for the *done?:N* token, control branches to the specified branch address if no new instruction is available.

Additional optional tokens control the loop counters (in module **LoopCount**):

cnt1	decrement loop counter 1 (ignored if 'done' is also specified)
cnt2	decrement loop counter 2 (ignored if 'done' is also specified)
cnt3	decrement loop counter 3 (ignored if 'done' is also specified)
cntF	decrement FBytes counter (ignored if 'done' is also specified)

If a loop counter decrement is specified for a word containing the *done* token, it is ignored since the instruction ends or control branches to idle address 0. If a loop counter decrement is specified for a word containing the *done?* token, the decrement occurs if and only if the conditional done fails and control continues to the next microcode address.

Control of pixel-memory address is by the following tokens:

ref	use Refresh address counter
ref+	use Refresh address counter, then increment it
ref++	use Refresh address counter, then increment it by 2
ref-	use Refresh address counter, then decrement it (DEFAULT)
dst	use Destination address counter
dst+	use Destination address counter, then increment it
dst++	use Destination address counter, then increment it by 2
dst-	use Destination address counter, then decrement it
src	use Source address counter
src+	use Source address counter, then increment it
src++	use Destination address counter, then increment it by 2
src-	use Source address counter, then decrement it
aux	use Auxiliary address counter, then increment it
aux+	use Auxiliary address counter

aux++	use Destination address counter, then increment it by 2
aux-	use Auxiliary address counter, then decrement it

If any increment or decrement is specified in a microcode word containing the *done* token, it is ignored since the counters will be loaded with the respective starting addresses from the next instruction, or control jumps to idle. For the conditional done (*done?*), the counter will be incremented or decremented if and only the conditional done fails and controls continues to the next microcode address.

Control of pixel-memory read/write is by the tokens **read**, 'wrtR, 'wrtS, and fwrtR. If it is not given, the specified pixel-memory address is read on this cycle. If it is given, the value on the 'sum' output of the pixel-ALU is written to the specified pixel-memory location provided either (1) the pixel -ALU Enable register is set, or (2) the ALU instruction specified FrcEn on the previous microcycle (see below). Note that pixel-memory is always read, even on a write cycle.

Linear expression evaluator function is controlled by the token:

tree

This causes a value of the linear expression evaluator output to appear at the input to the pixel-ALU. This value will remain valid until the next **tree** token is encountered. When **tree** is asserted on the same word as the **done** token, it causes the final byte of the LEE result to be generated; that byte (or the byte computed by passing that byte through the ALU) will not be available until the next clock cycle, which will be after the instruction has completed. Thus, normally **done** and **tree** would not be asserted in the same microcode word, unless as part of a special instruction meant to be used in conjunction with another instruction which processes the final byte.

The pixel-ALU instruction is specified by several sets of tokens, each controlling one of the functional blocks of the pixel-ALU. These are as follows:

The pixel-ALU is based on a standard ALU core, consisting of two universal function generators to produce a "propagate" and "generate" function, a sum-generator exclusive-or gate, and a Manchester carry-chain. The propagate and generate function blocks can generate any of the possible 16 logical functions of the byte-wide A and B inputs. The carry-in for the carry chain can be either a logical one, a logical zero, the Carry register

input, or the inverse of the Carry register.

The core is fed by a multiplexer which selects two of three possible inputs: the contents of the R register, the contents of the M register, or the value from the linear expression evaluator tree; the multiplexer also has a mode, for accelerating multiplies, in which it selects the R register and the M register logically-and'ed with the LSB of the S register. It also has 4 modes which allow accessing the R register of the neighboring PEs; using this requires knowledge of how the PEs are mapped to pixels.

On each cycle, the core generates the standard set of four status flags, which can be conditionally latched into CCLatch. The flags are:

Z	asserted when output is zero
N	asserted when MSB of output is one
L	asserted if result of two's complement operation is negative
V	overflow flag (exclusive-or of N and L flags)

The Enable register is used to selectively Enable pixel processors in the very small grain SIMD engine represented by the rasterizer. It can be set, its value inverted, its value pushed onto or popped from a one level deep stack, and its value logically-and'ed with one of the four ALU status flags, or the Carry register, or the inverse of any of these five flags.

The Sum and Carry outputs of the ALU core go into a shifter whose outputs are latched into the S and R registers and the Carry register. The shifter can simply pass the core output to the S,R, and Carry registers, or it can perform one bit left or right shifts involving the R register and the Carry register, or it can perform a 16-bit right shift using both registers. The R, S, and Carry registers can all be conditionally loaded.

The memory bus, DBus, interfaces to the pixel-ALU via the M, S, and R registers. The S and R registers can drive the DBus, either unconditionally, or qualified by the Enable register (the register contents are driven onto the DBus only if Enable=1). The M register can latch the DBus values. Thus, the S and R register values can be written into memory, unconditionally, or condtioned by the Enable, or the S and R regsiters can be recirculated into the M register, or memory can be read into the M register.

The **input multiplexer** is controlled by the following set of 4 tokens, each of which slects the A and B inputs to the function generators in the ALU core, as follows:

token	A input	B input
MT	M register	tree result
MR	M register	R register
TR	tree result	R register
(M)R	M register (conditional)	R register
MR-	M register	R register of lower-numbered PE
MR+	M register	R register of higher-numbered PE
(M)R-	M register (conditional)	R register of lower-numbered PE
(M)R+	M register (conditional)	R register of higher-numbered PE

The **(M)R*** tokens are meant to be used in a multiply loop. The A input is locally controlled by the LSB of the S register. If the LSB is 1, then the A input is the contents of the M register, otherwise it is zero.

Each **function generator** can produce any of the 16 possible logical functions of the two ALU core inputs A and B; however, only a subset of the 256 possibilities are available, since many are not useful. The function generators are controlled by the tokens **L:logical_function** and **F:arithmetic_function**. The following tokens are defined:

Token	P function	G Function
L:ZERO	all zeroes (the byte 0x00) (default)	zeroes (default)
L:ONE	all ones (the byte 0xff)	zeroes
L:A	the A input	zeroes
L:B	the B input	zeroes
L:Abar	~A (the ones-complement of the A input)	zeroes
L:Bbar	~B (the ones-complement of the B input)	zeroes
L:AandB	A * B (bitwise logical-and of A and B)	zeroes
L:AnandB	~ (A * B)	zeroes
L:AandBbar	A * ~B	zeroes
L:AbarandB	~A * B	zeroes
L:AorB	A + B (bitwise logical-or of A and B)	zeroes
L:AnorB	~ (A + B)	zeroes
L:AorBbar	A + ~B	zeroes
L:AbarorB	~A + B	zeroes
L:AxorB	A ^ B (bitwise logical-exclusive-or of A and B)	zeroes
L:AxnorB	~ (A ^ B)	zeroes

F:2A	zeroes	A
F:2B	zeroes	B
F:A-1	~A	A
F:B-1	~B	B
F:A+B	A ^ B	A * B
F:A-B-1	~ (A ^ B)	A * ~B
F:B-A-1	~ (A ^ B)	~A * B

The P and G functions feed a standard carry chain. Note that the combinations defined above insure that the P and G functions are bit-wise mutually exclusive.

The "carry in" to the carry chain is specified using the following tokens:

token	**carry in**
c=0	logical zero (default)
c=1	logical one
c=cry	the Carry register value
c=crybar	the logical-inverse of the Carry register value

The ALU core produces the following outputs:

Sum: the byte-wide output generated by a standard exclsuive-or sum generator from the "propagate" bit and the value of carry-in at that bit

Cout: the carry-out value of the carry-chain

Nflg: status flag equal to the MSB of Out

Zflg: status flag asserted if and only if Out is all-zeroes

Vflg: status flag asserted if result of a 2's complement operation overflows

Lflg: status flag asserted if result of 2's complement operation is negative

These ALU core outputs can then be latched.

The four flags are simply loaded into a 4-bit register if and only if the token

cc	load ALU status flag register

is asserted. The values in this register can be used to modify the Enable register, as described below.

The Enable register is normally used to qualify conditional writes into pixel-memory. It is

sometimes used for temporary storage as well. There is also an Enable-Hold registers, a one-level stack for the Enable register; this stack is not meant to be visible to the programmer. The Enable and Enable-Hold Register function is controlled by the following tokens:

token	enable register action	enable-hold register
<default>	<save>	<save>
setenab	enable = 1	<save>
enabinv	enable = ! enable	<save>
andcry	enable &&= carry	<save>
andcrybar	enable &&= !carry	<save>
andmsb	enable &&= Nflg	<save>
andmsbbar	enable &&= !Nflg	<save>
andzero	enable &&= Zflg	<save>
andnonz	enable &&= !Zflg	<save>
andovf	enable &&= Vflg	<save>
andovfbar	enable &&= !Vflg	<save>
andlt	enable &&= Lflg	<save>
andge	enable &&= !Lflg	<save>
andpush	<save>	enable-hold = enable
pop	enable = enable-hold	<save>
popand	enable &&= enable-hold	<save>

The treatment of the Sum and Cout outpus is far more complicated. These values go into the shifter, which feeds the R and S registers, and the Carry register. The shifter and these registers are controlled by the following mutually exclusive tokens:

token	R register	S register	Carry register
<default>	<saved>	<saved>	<saved>
ldR	Sum	<saved>	<saved>
ldRC	Sum	<saved>	loaded w/ Cout (if Enable=1)
ldS	<saved>	Sum	<saved>
ldSC	<saved>	Sum	loaded w/ Cout (if Enable=1)
ror	Sum>>1, Carry into MSB	<saved>	loaded w/ LSB of Sum (if Enable=1)
urRS	Sum>>1, Cout into MSB	S>>1, Sum0->MSB	<saved>
srRS	Sum>>1, Lflg into MSB	S>>1, Sum0->MSB	loaded w/ Cout (if Enable=1)

The token **ror** (*rotate right*) performs a single-bit right shift on the Sum output and loads it into the R regsiter, with a rotate through carry; this means that the LSB of Sum is shifted into the Carry register, and the old contents of the Carry register is loaded into the MSB of the R register. As with all operations, the Carry register is upadted only if the pixel is Enabled. There is no **rol** token; a left-shift can be accomplished within the ALU core using the token **L:2A** or **L:2B**.

The tokens **urRS** and **srRS** treat the R and S registers as a single 16-bit register (R is upper-byte, S is lower-byte). The ALU core **Sum** output is right-shifted and loaded into the R register, and the LSB of **Sum** is shifted into the MSB of the S register, with the remaining bits of the S register right-shifted (the LSB of the S register is discarded). The token **urRS** puts the ALU core carry-out, **COut**, into the MSB of the R register, and so is useful for unsigned multiplies. The token **srRS** puts the ALU core *less-than* flag, **Lflg**, into the MSB of the R register, and so is useful for signed multiplies. Note that **srRS** can also be used to sign-extend the Sum value: provided one of the ALU core inputs is zero and the carry-in is zero, then **Lflg** will be equal to the MSB of the *other* ALU core input, so the value loaded into the R register will be equal to the value of *other* ALU core input arithmetically shifted right (with sign-extension).

The tokens **ext[0:7]** are for strobing the external operation outputs. Each token strobes the respective external operatin output of the IGC (**ExtOp[i]H**) for one cycle. For example, asserting the token 'ext4' causes **ExtOp4H** to go high for cycle. It must be remembered that there is a severak cycle latency between when the microcode sequencer executes the microcode word containing the 'ext[i]' token and when the IGC output actually is asserted.

IV.3.4 — 3 General programming considerations.

Some important considerations when writing microcode for the IGC:

Loop counters. When implementing a loop using the token **bnzX:N**, where N is zero or negative, some microcode word in the loop must contain the token **cntX**; otherwise, the sequencer will hang in the loop (unless the loop counter was at 0 upon entering the loop).

It makes no sense to specify one of the count tokens (**cntX**) with the **done** token, because the **done** token ends the instruction and causes the loop counters to be loaded with the initial count values for the next instruction. If **done** and a **cntX** token are specified in the same microcode word, *asmEMC* prints a warning and.....

The loop counts may also be used as flags, to control conditional execution of portions of the microcode sequence. That is, the same microcode section may be used to implement more than one instruction, by specifying zero or non-zero values for LpCnt1 or LpCnt2 in the opcode specification, and using *bnz1*, *bnz2*, and *br* branches. For example:

```
%INSTRUCTION_A()    LpCnt1:1
%INSTRUCTION_B()    LpCnt1:0
bnz1:2 <other microcode tokens>     ; executed for both instructions
br:2 <other microcode tokens>       ; exceuted only for instruction B
<microcode tokens>                  ; executed only for instruction A
<microcode tokens>                  ; executed for both instructions
......
```

Pixel-memory address counters. It makes no sense to specify any of the post-decrement or post-increment pixel-memory address tokens in a 'done' microcode word, since the executing the 'done' word ends the instruction and causes the address counters to be loaded with the pixel-memory addresses for the next instruction. If any post-increment of post-decrement of pixel-memory address is specified in the same microcode word as *done*, *asmEMC* prints a warning and ignores the post-increment or post-decrement.

Control of pixel-ALUs. When none of the pixel-ALU control tokens are specified, the pixel-ALU's are effectively in a "no-op" state, since the Carry, Condition code, and Enable registers are saved, and the M, R, and S registers are re-cycled.

Programming conventions. By convention, only the Carry register, the Enable register, and the S Register are to be used to convey information between separate IGC instructions. By contrast, the M and R Registers, the ALU Status Flag register, and the Enable-Hold Register are used only *within* an IGC instruction. Their values are assumed to be undefined at the beginning of each new instruction.

This convention is adopted because: (1) the rasterizer documentation specifies the effect of each instruction on only the Enable, Carry, and S Registers, and (2) the EMC_ALUSave instruction backs up only these registers (during a T FIFO interrupt sequence). This convention could be modified, but only for compelling reasons.