REX3 Specification

Mark Goudy
Rick Jeng
Eric Linstadt
Mukesh Patel
Adrian Sfarti
Bob Sherburne

Silicon Graphics, Inc.

Revision 1.0

1 Introduction

This document describes the REX3, part of the Newport ("the least graphics you'll ever need") graphics subsystem.

1.1 Part Name and Number

Part Name: REX3
SGI Part Number: 099-9005-001
Vendor: LSI Logic Corporation
Vendor Part Number: L1A9040
Technology: LC300K (0.6 micron CMOS gate array)
Base Wafer: L300415P
Package: 304 MQUAD
Gate Count:149,000 equivalent gates, including 5.7K bits dual port RAM.

1.2 General Description

REX3 is the raster engine for Newport graphics. The basic operation of the raster engine is to draw lines and spans. Various packed formats of host DMA are also supported. It is based on some of the concepts of REX1, i.e there is no dedicated geometry engine for graphics. Instead, the hosts floating point unit is used as the geometry engine. Like REX1, Z buffering is done by the host in system memory. REX3's register interface has been optimized for minimum host writes to execute primitives. REX3 has various pixel formats to accommodate a low cost 8 bits/pixel system as well as a 24 bits/pixel system. Besides the pixel planes REX3 supports CID, PUP and Overlay planes. Also, in order to achieve high frame buffer writing bandwidth, the frame buffer is architected as an 8 way interleave combined with a Y axis interleave. There are two sets of RGBA iterators so 2 shaded pixels/clock are generated. For flat filled spans, four pixels/clock are generated. In order to bound the package size to less than 304 pins, the frame buffer data is byte serialized for each of the eight interleaves. This data is deserialized by RB2s' before writing to the frame buffer. In order to limit the number of gates in REX3, the read/write formaters and the logicop functions have been incorporated into RB2s.

1.3 Features

- 33 MHz GIO64 Bus Interface
- 66 MHz Isotropic 8 way interleaved frame Buffer Interface
- 33MHz Display Bus Interface with synchronous / asynchronous / burst mode slave support
- Bresenham line iterators
- RGB and CI anti-aliased Bresenham lines
- Bi-endian support
- Software Z buffer
- Blend function
- 1280 x 1024 resolution
- Upto 76Hz screen refresh
- Upgradable from 8 pixel + 2PUP + 2CID planes to 24 pixel + 8 Overlay(or 4+4) + 2PUP + 2CID planes
- Optional Express Video ready

- GenLock capability

1.4 Newport Architecture

Newport graphics is made of the following major components:

1. REX3
2. RB2
3. Frame buffer
4. RO1
5. XMAP9
6. CMAP
7. VC2
8. RAMDAC
9. Static Ram

The graphics pipeline begins with the host writing into the REX3 registers to execute primitives. REX3 transforms these primitives into screen coordinates and writes the data via RB2 into the frame buffer. The frame buffer is made of Vrams (2MBit) in an 8 way interleave configuration. The serial ports of the frame buffer are read into RO1 and passed into XMAP9 which manipulates the data for multi mode screen. XMAP9 passes the data onto the CMAP which consists of high speed static ram for Color Index modes. When in RGB mode, the data goes through other static ram within CMAP that is normally linearly mapped, although for image processing applications it does not have to be linearly mapped. The output of CMAP is fed into the RAMDAC for display to the screen. The gamma correction tables reside in the RAMDAC. The output of the CMAPs is also fed back to XMAP9 and output onto the Video port. Video data can also be accepted from the video port and output to the CMAP to display on the graphics monitor. VC2 provides all the relevant timing for the graphics sub system.
A block diagram of the Newport graphics sub-system is shown in Figure 1.

1.5 REX3 Architecture

Figure 2 shows the top level block diagram of REX3. REX3 could be viewed as three logical blocks. The first block, which interface to the host bus (GIO64) is the GIO block. REX3 supports both GIO64 and GIO32 protocol, the default being GIO64. The GIO64 bus may be either 64 or 32 bits wide. This block receives commands for all the primitives that REX3 draws as well as provide host access to other devices in the display and video (optional) subsystems. REX3 is implemented as a GIO64 bus slave which decodes addresses on the GIO64 bus to detect accesses to its own registers, or those within the Video subsystem. Commands and data to and from the Display subsystem are sent over the Display Control Bus. The REX3 is the master of the Display Control Bus. The second block is the iterator block. This block generates the frame buffer addresses, interpolates the colors and provides masking and various patterning capabilities. The pixel address generation for lines is done by Bresenham iterators. This block also handles the coverage values for anti - aliased lines and does the swizzle for the frame buffer interleaving. The third section is the memory controller and pixel pipe. There are four instances of the memory controller and pixel pipe. This block has the frame buffer controller as well as the CID checking, color compare, dither and Blend functions. The GIO and Iterator sections operate at 33MHz and the memory controller and the pixel pipe operates at 66MHz. The GIO interface with the host is via a fifo which is 64 wide and 32 deep. The high water mark on the GIO fifo is programmable. The Iterator section communicates to the memory controller and pixel pipe via 4 bank fifos. Each bank fifo consists of one write and two read fifos. For screen to screen copy operations the Iterator section generates a read into the read bank fifos and swizzles the data before writing it into the write bank fifos. The memory controller operates each of the 4 banks independent of each other. The memory is cycled in 4 clocks (60nS) for page mode operations. Figure 3 shows the internal data path of REX3.

1.6 Performance

Operation	Performance
Shaded spans	50M pixels/sec
Flat-filled spans	100M pixels/sec
Fastclear	400M pixels/sec
DMA	50M pixels/sec
Screen to Screen copy	40M pixels/sec
Depthcued or constant color linedraw rate	20M pixels/sec
10 pixel RGB Anti-aliased lines	200K lines/sec
Random points	6M points/sec

Table 1: REX3 Performance

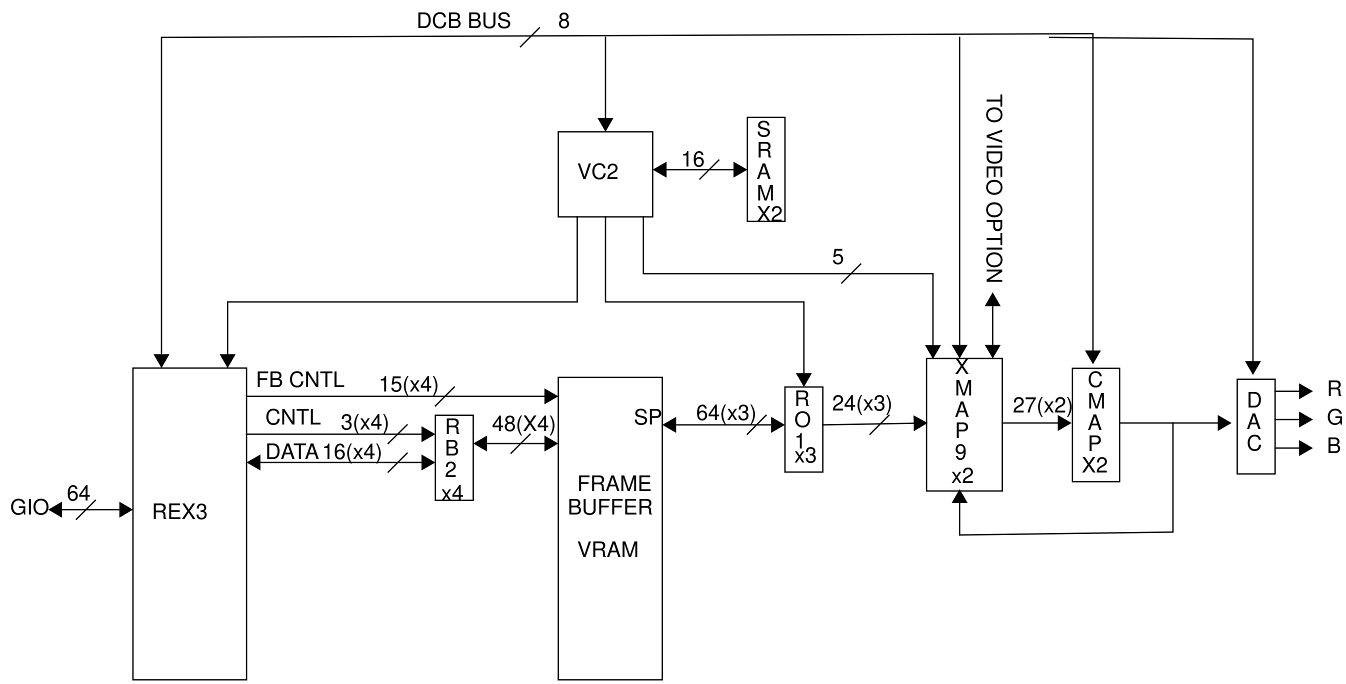


FIGURE 1. Newport Graphics Sub-system

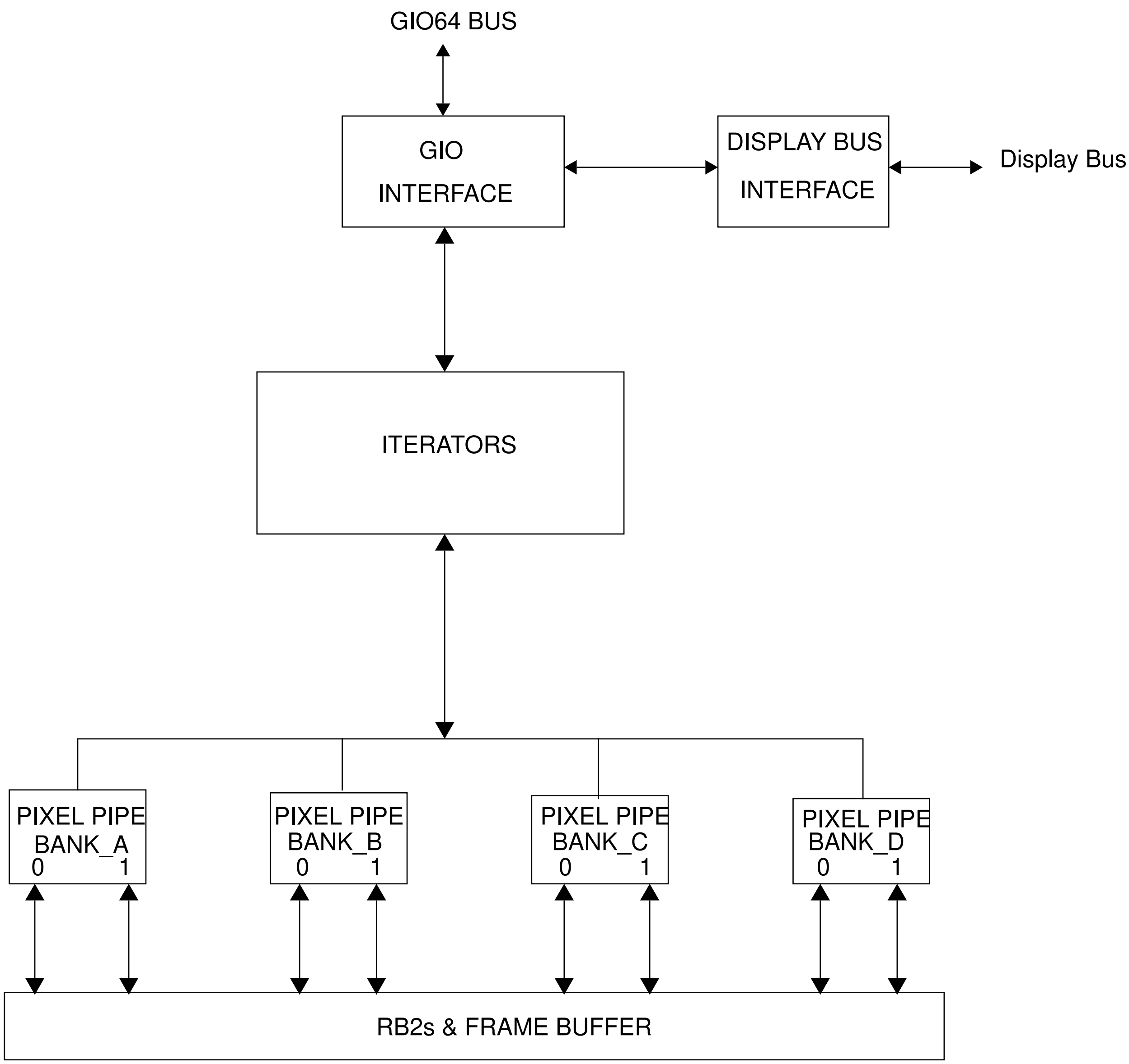
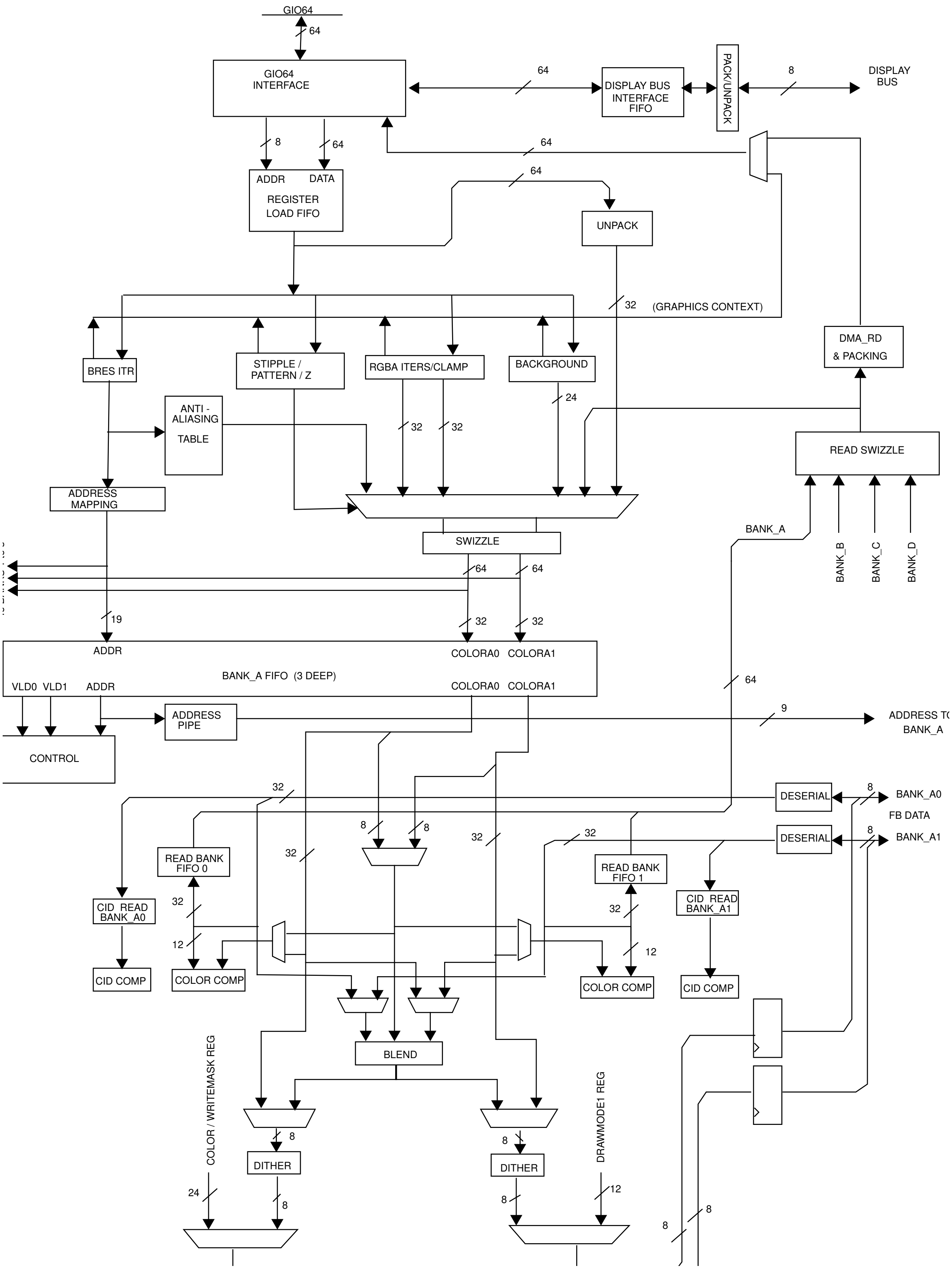


FIGURE 2. REX3 top-level block diagram

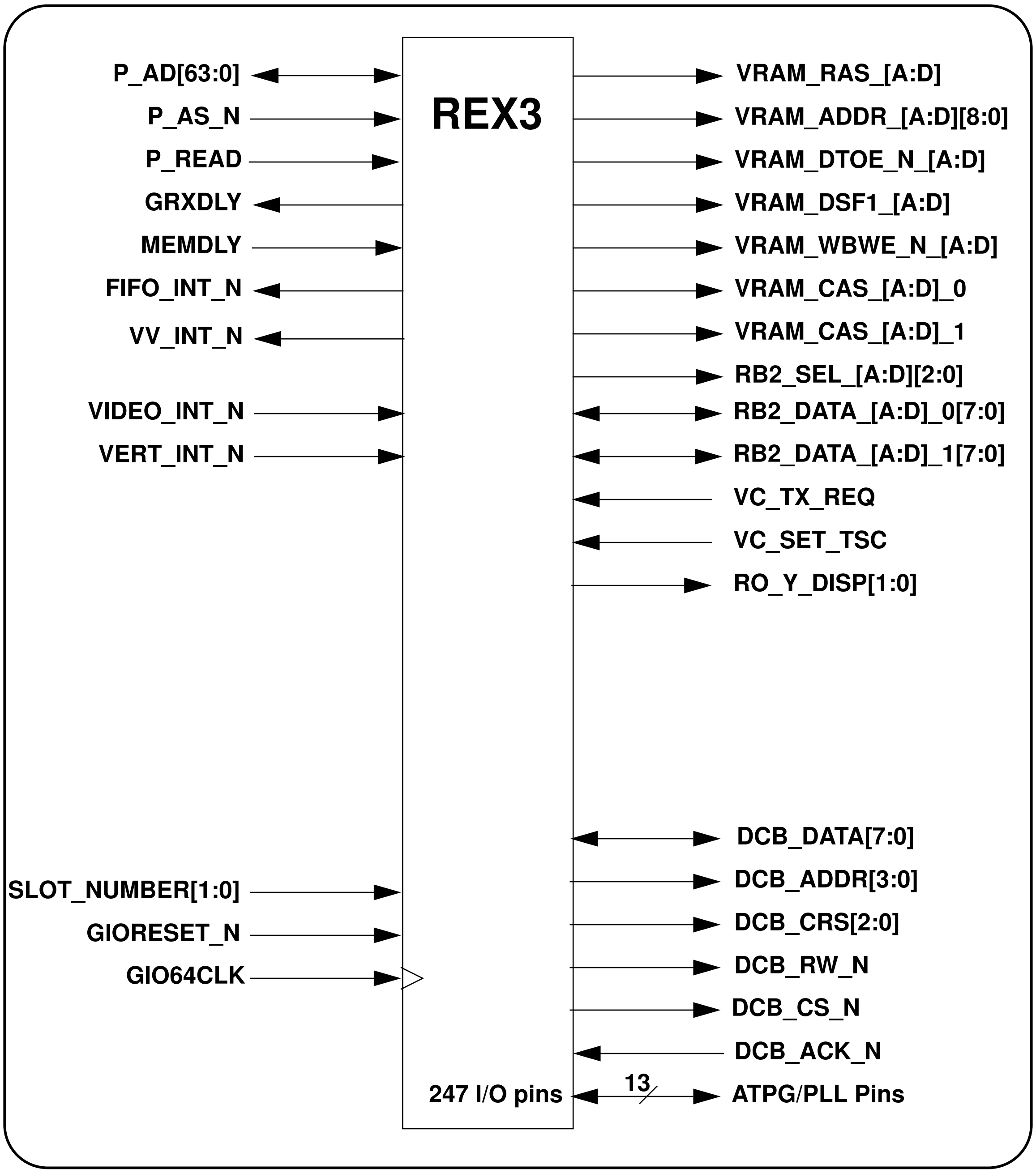
FIGURE 3. REX3 Internal Data Path



2 Device Interface

2.1 Pin Diagram

FIGURE 4. REX3 Pin Diagram



2.2 Pin Descriptions

The following tables list for each REX3 pin the assertion level, direction (I, O, I/O), LSI IO cell type, followed by a brief functional description.

TABLE 2. GIO64 Bus Interface

Pin Name	Level	Type	Function
P_AD[63:0]	NA	I/O(BD8TRPU)	64-bit pipelined Address/data bus
P_AS_N	L	I(TLCHT)	Asserted during an Address cycle on the GIO bus.
P_READ	NA	I(TLCHT)	Indicates the direction of the data transfer during Address cycles. After the Address cycle, P_READ is driven low to indicate that an active bus cycle is taking place. The GIO64 bus master preempts a transaction by asserting P_READ.
GRXDLY	H	O(BT8RP)	When asserted, this signal indicates that for read data cycles, the REX3 is not returning valid data on the P_AD bus. For write cycles, the REX3 asserts GRXDLY when the next transfer on the on the non-pipelined side of the GIO64 bus must be stalled (one more word will be accepted by the REX3).
MEMDLY	H	I(TLCHT)	When deasserted during write data cycles, this signal indicates that the host is presenting valid data on the GIO64 bus. When asserted during read data cycles, this signal indicates that the host cannot accept data from the REX3 during the next cycle.
FIFO_INT_N	L	O(BT4OD)	REX3 GFIFO/BFIFO above/below interrupt (Open Drain).
VV_INT_N	L	O(BT4OD)	VC2 Vertical retrace or Kaleidoscope Video Option interrupt (Open Drain).
SLOT_NUMBER[1:0]	NA	I(TLCHT)	Address bits [23:22] of the Newport graphics board. Address bits [31:24] = “0001_1111”.
GIORESET_N	L	I(TLCHT)	Synchronous reset.
GIO64CLK	NA	I(CMOS)	Positive GIO64 bus clock. All GIO64 bus signals are clocked on the rising edge of this signal.

TABLE 3. VRAM/RB2/REORG Interface

Pin Name	Type	Function
VRAM_RAS_[A:D]	O(B4)	VRAM RAS, for the four memory banks[A:D]
VRAM_ADDR_[A:D][8:0]	O(BT4RP)	VRAM Address bus
VRAM_DTOE_N_[A:D]	O(BT4RP)	VRAM Transfer Enable / Output Enable.
VRAM_DSF1_[A:D]	O(BT4RP)	VRAM special function control pin.
VRAM_WBWE_N_[A:D]	O(B4)	VRAM bank write enable (active low).
VRAM_CAS_[A:D]_0	O(BT4RP)	VRAM CAS for the even halves of the four memory banks
VRAM_CAS_[A:D]_1	O(BT4RP)	VRAM CAS for the odd halves of the four memory banks
RB2_SEL_[A:D][2:0]	O(B4)	Operation selects for the four memory banks. Encoded as follows: 000 NOOP 001 Write (4 components), lower pixel into OLY planes 010 Write higher pixel into OLY planes 011 Load write mask and partial DRAWMODE1 Regs 100 Read (4 components), lower pixel of OLY planes 101 Read higher pixel of OLY planes 110 Read lower pixel CID bits (for CID checking) 111 Read higher CID bits (for CID checking)
RB2_DATA_[A:D]_0[7:0]	I/O(BD8TRPU)	RB2 data for the even halves of the four memory banks
RB2_DATA_[A:D]_1[7:0]	I/O(BD8TRPU)	RB2 data for the odd halves of the four memory banks
VC_TX_REQ	I(TLCHT)	Transfer request
VC_SET_TSC	I(TLCHT)	Set top of scan.
RO_Y_DISP[1:0]	O(BT4RP)	Scanline (modulo-4) for staggering the frame buffer

TABLE 4. Display Control Bus Interface

Pin Name	Level	Type	Function
DCB_DATA[7:0]	NA	I/O(BD8TRPU)	Data read from (DCB_RW_N = 1) or written (DCB_RW_N = 0) to the Display Control Bus slave devices.
DCB_ADDR[3:0]	NA	O(BT4RP)	Display Control Bus slave device Address.
DCB_CRS[2:0]	NA	O(BT8RP)	Display Control Bus slave device command or register select field.
DCB_RW_N	NA	O(BT8RP)	Read/Write direction signal.
DCB_CS_N	NA	O(BT4RP)	Display Control Bus command strobe, indicating that valid DCB_ADDR, DCB_CRS, DCB_RW_N and, for write transfers, DCB_DATA are on the bus.
DCB_ACK_N	L	I(IBUFN)	Acknowledge signal for Display Control Bus slaves to handshake transfers with the REX3. When asserted during write cycles, DCB_ACK_N indicates that the slave device has accepted the DCB_DATA, and that the next Display Control Bus cycle may begin. During read cycles, the Display Control Bus slave asserts DCB_ACK to indicate that it has placed valid data on the DCB_DATA lines.

TABLE 5. Miscellaneous Back-End Pins

Pin Name	Level	Type	Function
VERT_INT_N	L	I(IBUFN)	Vertical retrace/sync interrupt from VC2
VIDEO_INT_N	L	I(IBUFN)	Interrupt from Express Video option

TABLE 6. ASIC Mandatory PLL and Test Pins

Pin Name	Level	Type	Function
JTAG_TDI	NA	I(TLCHTU)	Scan Test Data In
JTAG_TMS	NA	I(TLCHTU)	Scan Test Mode Select. Selects the scan input of all flip-flops when driven low. Driven high for normal operation.
JTAG_TCK	NA	I(TLCHTU)	Scan Test Clock
JTAG_TDO	NA	O(B2)	Scan Test Data/Parametric NAND tree/PLL Test Clock Out
TEI	NA	I(TLCHN)	I/O pin tristate enable. When driven low, all bidirectional pins and tri-state unidirectional pins are forced into high impedance state. Driven high for normal operation.
TP[1:0]	NA	I(TLCHT)	PLL/Scan Test Mode. Encoded as follows: 00 Normal Operation. VCO ripple counter output -> JTAG_TDO 01 PLL bypass mode. Scan chain output -> JTAG_TDO 10 PLL bypass mode. Parametric NAND tree -> JTAG_TDO 11 Scan mode. JTAG_TCK drives all flops. Scan chain output -> JTAG_TDO. VCO is disabled for IDD test
PLL_RESET_N	L	I(TLCHT)	PLL Reset. The loop filter output is grounded when asserted
LP1	NA	O(DDRVO)	PLL Charge Pump Output / Loop Filter Input
LP2	NA	I/O(RDDRVPD)	PLL VCO input / Loop Filter Output
AVDD	NA	I(RDDRV)	PLL Analog VDD
AVSS	NA	I(RDDRV)	PLL Analog VSS
AGND	NA	O(RDDRVO)	PLL Analog Ground

2.3 VHDL Description

This section describes the device level interface to the REX3 as a VHDL entity.

```
entitiy REX3 is
  port(
--GIO64 Bus interface (74 pins)
    P_AD : inout mvl7w_vector (63 downto 0);
    P_AS_N : in mvl7w;
    P_READ : in mvl7w;
    GRXDLY : out mvl7w;
    MEMDLY : in mvl7w;
    FIFO_INT_N : out mvl7w;
    VV_INT_N : out mvl7w;
    SLOT_NUMBER : in mvl7w_vector (1 downto 0);
    GIORESET_N : in mvl7w;
    GIO64CLK : in mvl7w;
--VRAM/RB2/REORG Interface (140 pins)
    VRAM_RAS_A: out mvl7w;
    VRAM_ADDR_A : out mvl7w_vector (8 downto 0);
    VRAM_DTOE_N_A : out mvl7w;
    VRAM_DSF1_A : out mvl7w;
    VRAM_WBWE_N_A : out mvl7w;
    VRAM_CAS_A_0 : out mvl7w;
    VRAM_CAS_A_1 : out mvl7w;
    RB2_SEL_A : out mvl7w_vector (2 downto 0);
    RB2_DATA_A_0 : inout mvl7w_vector (7 downto 0);
    RB2_DATA_A_1 : inout mvl7w_vector (7 downto 0);
    VRAM_RAS_B : out mvl7w;
    VRAM_ADDR_B : out mvl7w_vector (8 downto 0);
    VRAM_DTOE_N_B : out mvl7w;
    VRAM_DSF1_B : out mvl7w;
    VRAM_WBWE_N_B : out mvl7w;
    VRAM_CAS_B_0 : out mvl7w;
    VRAM_CAS_B_1 : out mvl7w;
    RB2_SEL_B : out mvl7w_vector (2 downto 0);
    RB2_DATA_B_0 : inout mvl7w_vector (7 downto 0);
    RB2_DATA_B_1: inout mvl7w_vector (7 downto 0);
    VRAM_RAS_C : out mvl7w;
    VRAM_ADDR_C : out mvl7w_vector (8 downto 0);
    VRAM_DTOE_N_C : out mvl7w;
    VRAM_DSF1_C : out mvl7w;
    VRAM_WBWE_N_C : out mvl7w;
    VRAM_CAS_C_0 : out mvl7w;
    VRAM_CAS_C_1 : out mvl7w;
    RB2_SEL_C : out mvl7w_vector (2 downto 0);
    RB2_DATA_C_0 : inout mvl7w_vector (7 downto 0);
    RB2_DATA_C_1 : inout mvl7w_vector (7 downto 0);
    VRAM_RAS_D : out mvl7w;
    VRAM_ADDR_D : out mvl7w_vector (8 downto 0);
    VRAM_DTOE_N_D : out mvl7w;
    VRAM_DSF1_D : out mvl7w;
    VRAM_WBWE_N_D : out mvl7w;
    VRAM_CAS _D_0 : out mvl7w;
    VRAM_CAS _D_1 : out mvl7w;
    RB2_SEL_D : out mvl7w_vector (2 downto 0);
    RB2_DATA_D_0 : inout mvl7w_vector (7 downto 0);
    RB2_DATA_D_1 : inout mvl7w_vector (7 downto 0);
    VC_TX_REQ : in mvl7w;
    VC_SET_TSC : in mvl7w;
```

```
     RO_Y_DISP : out mvl7w_vector (1 downto 0);
--Display Control Bus Interface (18 pins)
     DCB_ADDR : out mvl7w_vector (3 downto 0);
     DCB_DATA : inout mvl7w_vector (7 downto 0);
     DCB_CRS : out mvl7w_vector (2 downto 0);
     DCB_CS_N : out mvl7w;
     DCB_RW_N : out mvl7w;
     DCB_ACK_N : in mvl7w;
--Miscellaneous Back End pins (2 pins)
     VERT_INT_N : in mvl7w;
     VIDEO_INT_N : in mvl7w;
--ASIC Mandatory pins (13 pins)
     TEI : in mvl7w; --External tri-state control
     JTAG_TDI : in mvl7w;
     JTAG_TMS : in mvl7w;
     JTAG_TCK : in mvl7w;
     JTAG_TDO : out mvl7w;
     TP : in mvl7w_vector (1 downto 0);
     PLL_RESET_N : in mvl7w;
     LP1 : out mvl7w;
     LP2 : in mvl7w;
     AGND : out mvl7w;
     AVSS : in mvl7w;
     AVDD : in mvl7w
  );
end REX3;
```

2.4 Package Pin Assignment

The following list of package pin assignments is from the LSI Logic LBOND program. The REX3 is mounted in a 304 MQUAD cavity down package, and pins are numbered by LSI in a counter-clockwise manner when viewing the die. When mounted (cavity down) on the PC board, pins are also numbered in a counter-clockwise fashion. Therefore, the printed circuit board pin number is equal to (305-LSI pin number).

Pin Number	Signal Name
1	vdd
2	p_ad_10
3	p_ad_11
4	p_ad_12
5	p_ad_13
6	p_ad_14
7	vss
8	p_ad_15
9	p_ad_16
10	p_ad_17
11	p_ad_18
12	p_ad_19
13	vdd
14	p_ad_20
15	p_ad_21
16	p_ad_22
17	p_ad_23
18	p_ad_24
19	vss
20	gio64clkx
21	pll_reset_n
22	lp1
23	lp2
24	agnd
25	avdd
26	avss
27	p_ad_25
28	p_ad_26
29	p_ad_27
30	p_ad_28
31	p_ad_29
32	vdd
33	p_ad_30
34	p_ad_31

35	p_as_n
36	p_read
37	p_memdly
38	p_grxdly
39	p_ad_32
40	p_ad_33
41	vdd
42	vss
43	vss
44	p_ad_34
45	p_ad_35
46	p_ad_36
47	p_ad_37
48	p_ad_38
49	vdd
50	p_ad_39
51	p_ad_40
52	p_ad_41
53	p_ad_42
54	p_ad_43
55	vss
56	p_ad_44
57	p_ad_45
58	p_ad_46
59	p_ad_47
60	p_ad_48
61	vdd
62	p_ad_49
63	p_ad_50
64	p_ad_51
65	p_ad_52
66	p_ad_53
67	vss
68	p_ad_54
69	p_ad_55
70	p_ad_56
71	p_ad_57
72	p_ad_58
73	vdd
74	p_ad_59
75	p_ad_60
76	vss
77	vdd
78	p_ad_61
79	p_ad_62
80	p_ad_63
81	slot_number_0
82	slot_number_1
83	tp_0
84	tp_1
85	video_int_n
86	vert_int_n
87	jtag_tms
88	jtag_tdi
89	jtag_tck
90	tei
91	jtag_tdo
92	ro_y_disp_0
93	ro_y_disp_1
94	vss
95	vc_tx_req
96	vc_set_tsc
97	rb2_data_a_0_0
98	rb2_data_a_0_1
99	rb2_data_a_0_2
100	rb2_data_a_0_3
101	rb2_data_a_0_4
102	vdd
103	rb2_data_a_0_5
104	rb2_data_a_0_6
105	vss
106	rb2_data_a_0_7
107	vram_wbwe_n_a
108	vram_dtoe_n_a
109	vram_dsf1_a
110	rb2_sel_a_0
111	rb2_sel_a_1
112	vdd
113	rb2_sel_a_2
114	vram_addr_a_0
115	vram_addr_a_1
116	vram_addr_a_2

117	vram_addr_a_3
118	vss
119	vdd
120	vram_addr_a_4
121	vram_addr_a_5
122	vram_addr_a_6
123	vram_addr_a_7
124	vram_addr_a_8
125	vss
126	vram_ras_a
127	vram_cas_a_0
128	vram_cas_a_1
129	rb2_data_a_1_0
130	rb2_data_a_1_1
131	vdd
132	rb2_data_a_1_2
133	rb2_data_a_1_3
134	rb2_data_a_1_4
135	rb2_data_a_1_5
136	rb2_data_a_1_6
137	vss
138	rb2_data_a_1_7
139	rb2_data_b_0_0
140	vdd
141	rb2_data_b_0_1
142	rb2_data_b_0_2
143	rb2_data_b_0_3
144	rb2_data_b_0_4
145	rb2_data_b_0_5
146	vss
147	rb2_data_b_0_6
148	rb2_data_b_0_7
149	vram_wbwe_n_b
150	vram_dtoe_n_b
151	vram_dsf1_b
152	vdd
153	vss
154	rb2_sel_b_0
155	rb2_sel_b_1
156	rb2_sel_b_2
157	vram_addr_b_0
158	vdd
159	vram_addr_b_1
160	vram_addr_b_2
161	vram_addr_b_3
162	vram_addr_b_4
163	vram_addr_b_5
164	vss
165	vram_addr_b_6
166	vram_addr_b_7
167	vram_addr_b_8
168	vram_ras_b
169	vram_cas_b_0
170	vdd
171	vram_cas_b_1
172	rb2_data_b_1_0
173	rb2_data_b_1_1
174	rb2_data_b_1_2
175	rb2_data_b_1_3
176	vss
177	rb2_data_b_1_4
178	rb2_data_b_1_5
179	rb2_data_b_1_6
180	rb2_data_b_1_7
181	rb2_data_c_0_0
182	vdd
183	rb2_data_c_0_1
184	rb2_data_c_0_2
185	rb2_data_c_0_3
186	rb2_data_c_0_4
187	rb2_data_c_0_5
188	vss
189	rb2_data_c_0_6
190	rb2_data_c_0_7
191	vram_wbwe_n_c
192	vram_dtoe_n_c
193	vram_dsf1_c
194	vdd
195	vss
196	rb2_sel_c_0
197	rb2_sel_c_1

198	rb2_sel_c_2
199	vram_addr_c_0
200	vram_addr_c_1
201	vdd
202	vram_addr_c_2
203	vram_addr_c_3
204	vram_addr_c_4
205	vram_addr_c_5
206	vram_addr_c_6
207	vss
208	vram_addr_c_7
209	vram_addr_c_8
210	vram_ras_c
211	vram_cas_c_0
212	vram_cas_c_1
213	vdd
214	rb2_data_c_1_0
215	rb2_data_c_1_1
216	rb2_data_c_1_2
217	rb2_data_c_1_3
218	rb2_data_c_1_4
219	vss
220	rb2_data_c_1_5
221	rb2_data_c_1_6
222	rb2_data_c_1_7
223	rb2_data_d_0_0
224	rb2_data_d_0_1
225	vdd
226	rb2_data_d_0_2
227	rb2_data_d_0_3
228	vss
229	vdd
230	rb2_data_d_0_4
231	rb2_data_d_0_5
232	rb2_data_d_0_6
233	rb2_data_d_0_7
234	vram_wbwe_n_d
235	vss
236	vram_dtoe_n_d
237	vram_dsf1_d
238	rb2_sel_d_0
239	rb2_sel_d_1
240	rb2_sel_d_2
241	vdd
242	vram_addr_d_0
243	vram_addr_d_1
244	vram_addr_d_2
245	vram_addr_d_3
246	vss
247	vram_addr_d_4
248	vram_addr_d_5
249	vram_addr_d_6
250	vram_addr_d_7
251	vram_addr_d_8
252	vdd
253	vram_ras_d
254	vram_cas_d_0
255	vram_cas_d_1
256	rb2_data_d_1_0
257	rb2_data_d_1_1
258	rb2_data_d_1_2
259	vss
260	rb2_data_d_1_3
261	rb2_data_d_1_4
262	rb2_data_d_1_5
263	rb2_data_d_1_6
264	rb2_data_d_1_7
265	vdd
266	vss
267	dcb_data_0
268	dcb_data_1
269	dcb_data_2
270	dcb_data_3
271	dcb_data_4
272	vdd
273	dcb_data_5
274	dcb_data_6
275	dcb_data_7
276	dcb_crs_0
277	dcb_crs_1
278	vss

279	dcb_crs_2
280	dcb_rw_n
281	vdd
282	dcb_cs_n
283	dcb_addr_0
284	dcb_addr_1
285	dcb_addr_2
286	dcb_addr_3
287	vss
288	dcb_ack_n
289	vv_int_n
290	fifo_int_n
291	vss
292	gioreset_n
293	p_ad_0
294	p_ad_1
295	p_ad_2
296	p_ad_3
297	p_ad_4
298	vdd
299	p_ad_5
300	p_ad_6
301	p_ad_7
302	p_ad_8
303	p_ad_9
304	vss

3 Programmer Interface

3.1 Registers

Table 7 lists the host accessible registers in REX3.

Addresses shown are an offset from the base GIO address of 0x1FnF0000, where n=(0,4,8,C), depending upon the strapping of the GIO64 SLOT_NUMBER(1:0) pins. Address offsets beginning with 0x1nnn are intended to map corresponding registers into a separate "protected" page.

Access to address + 0x0800 issues primitive GO command.

Type "⊗" registers are not passed through either BFIFO or GFIFO, and force an immediate action when written to.

Type "◊" registers are associated with the Display Control Bus and go through BFIFO.

Registers other than type "⊗" and "◊" are associated with the graphics context and go through GFIFO.

Writes to type "•" registers will stall at the output of GFIFO until the graphics pipeline is idle.

Type "2c" indicates twos-complement value.

Type "sm" indicates signed magnitude value.

Write/Read format bit grouping is shown with location of binary point, (for COLOR registers, 24-bit mode binary point shown). "s" refers to sign bit and "o" refers to overflow bit. Parenthesis are used to indicate a place holder for unused bits.

Write format "#" denotes write-only command address.

Unused bits return 0 when read.

Address	Name	Type	Write	Read	Description
0x0000	DRAWMODE1	•	32	32	Draw mode bits.
0x0004	DRAWMODE0		24	24	Draw instruction and mode bits.
0x0008	LSMODE		28	28	Line stipple mode register.
0x000C	LSPATTERN		32	32	Line stipple pattern, (msb = first pixel).
0x0010	LSPATSAVE		32	32	Copy of LSPATTERN for pattern restore (LSRESTORE).
0x0014	ZPATTERN		32	32	Pattern register, (msb = first pixel).
0x0018	COLORBACK	•	32	32	AGBR/CI opaque patterning color or blendfunction destination color.
0x001C	COLORVRAM	•	32	32	VRAM FASTCLEAR color, (set DRAWDEPTH and RGBMODE first).
0x0020	ALPHAREF	•	8	8	AFUNCTION reference alpha value.
0x0024	STALL0	•	#		Forces stall at the output of GFIFO until graphics pipeline is idle.
0x0028	SMASK0X	2c	16,16	16,16	Screenmask 0: min, max boundaries, (window relative GL smask).
0x002C	SMASK0Y	2c	16,16	16,16	Screenmask 0: min, max boundaries, (window relative GL smask).
0x0030	SETUP		#		Performs line/span setup without iteration (ignore DOSETUP).
0x0034	STEPZ		#		Enables ZPATTERN (Z test fail) for one iteration, (current pixel).
0x0038	LSRESTORE		#		Updates LSPATTERN/LSRCOUNT with LSPATSAVE/LSRCNTSAVE.
0x003C	LSSAVE		#		Updates LSPATSAVE/LSRCNTSAVE with LSPATTERN/LSRCOUNT.
0x0100	XSTART	2c	16.4(7)	16.4(7)	Iterator X start-point (current), full state for context switch.
0x0104	YSTART	2c	16.4(7)	16.4(7)	Iterator Y start-point (current), full state for context switch.
0x0108	XEND	2c	16.4(7)	16.4(7)	Iterator X endpoint, full state for context switch.
0x010C	YEND	2c	16.4(7)	16.4(7)	Iterator Y endpoint, full state for context switch.
0x0110	XSAVE	2c	16	16	Copy of XSTART integer value for BLOCK addressing MODE.
0x0114	XYMOVE	2c •	16,16	16,16	X,Y offset from XSTART,YSTART for relative operations (Scr2Scr).
0x0118	BRESD	2c	19.8	19.8	Bresenham “d” error term, for context switch.
0x011C	BRESS1	2c	2.15	2.15	Antialiased Bresenham “s1” coverage term, for context switch.
0x0120	BRESOCTINC1		3(4),17.3	3(4),17.3	Bresenham octant & “incr1” error term increment value, for cntx switch.
0x0124	BRESRNDINC2	2c	8(3),18.3	8(3),18.3	Bresenham 8-bit octant rounding mode (msb == octant 1, lsb == octant 8) & Bresenham “incr2” error term increment value, for context switch.
0x0128	BRESE1		1.15	1.15	Bresenham “e1” constant (minor slope) for antialiased line draw.
0x012C	BRESS2	2c	18.8	18.8	Antialiased Bresenham “s2” coverage term, for context switch.
0x0130	AWEIGHT0		8 x 4	8 x 4	First half of 16x4-bit antialiased RGB/CI line weight table.
0x0134	AWEIGHT1		8 x 4	8 x 4	Second half of 16x4-bit antialiased RGB/CI line weight table.
0x0138	XSTARTF		12.4(7)		GL version of XSTART, (zeros 4 msbs).
0x013C	YSTARTF		12.4(7)		GL version of YSTART, (zeros 4 msbs).
0x0140	XENDF		12.4(7)		GL version of XEND, (zeros 4 msbs).
0x0144	YENDF		12.4(7)		GL version of YEND, (zeros 4 msbs).
0x0148	XSTARTI	2c	16		Integer format for XSTART.
0x014C	XENDF1		12.4(7)		Same as XENDF.
0x0150	XYSTARTI	2c	16,16		Packed integer format for XSTART & YSTART.
0x0154	XYENDI	2c	16,16		Packed integer format for XEND & YEND.
0x0158	XSTARTENDI	2c	16,16		Packed integer format for XSTART & XEND.

Table 7: REX3 host visible registers.

Address	Name	Type	Write	Read	Description
0x0200	COLORRED		o12.11	o12.11	Red/CI shade full state (CI modes = o8.11, o4.11; RGB red = o8.15, etc.).
			o12.9		12-bit CI mode shade. (Must first init DRAWMODE1 RGBMODE and DRAWDEPTH fields to set this register write mode; not for ctxt restore.)
0x0204	COLORALPHA		o8.11	o8.11	Full state of alpha shade.
0x0208	COLORGRN		o8.11	o8.11	Full state of green shade.
0x020C	COLORBLUE		o8.11	o8.11	Full state of blue shade.
0x0210	SLOPERED	sm 2c	s(7)12.11	 13.11	Red/CI DDA slope: "s" =1 on write denotes sm to 2c conversion, in which case 12.11 result is computed; always "s" is placed into msb of 13.1 field.
0x0214	SLOPEALPHA	sm 2c	s(11)8.11	 9.11	Alpha DDA slope: "s" =1 on write denotes sm to 2c conversion, in which case 8.11 result is computed; always "s" is placed into msb of 9.1 field.
0x0218	SLOPEGRN	sm 2c	s(11)8.11	 9.11	Green DDA slope: "s" =1 on write denotes sm to 2c conversion, in which case 8.11 result is computed; always "s" is placed into msb of 9.1 field.
0x021C	SLOPEBLUE	sm 2c	s(11)8.11	 9.11	Blue DDA slope: "s" =1 on write denotes sm to 2c conversion, in which case 8.11 result is computed; always "s" is placed into msb of 9.1 field.
0x0220	WRMASK	•	24	24	Write mask for pixel, OLAY, or PUP/CID planes, (lsbs for 8-bit system).
0x0224	COLORI		24		Packed BGR or CI color registers -- zeros fractions. (Must program DRAWMODE1 RGBMODE bit first to set color register write mode.)
0x0228	COLORX		12.11		Color index shade, zeros overflow bit.
0x022C	SLOPERED1	sm	s(7)13.11		Same as SLOPERED.
0x0230	HOSTRW0		32	32	Host PIO/DMA data port, most significant word.
0x0234	HOSTRW1		32	32	Host PIO/DMA data port, least significant word.
0x0238	DCBMODE	◊	29	29	Display control bus mode register.
0x0240	DCBDATA0	◊	32	32	Display control bus data port, most significant word.
0x0244	DCBDATA1	◊	32	32	Display control bus data port, least significant word.
0x1300	SMASK1X	2c •	16,16	16,16	Screenmask 1: min, max boundary (screen absolute: X11 directionality).
0x1304	SMASK1Y	2c •	16,16	16,16	Screenmask 1: min, max boundary.
0x1308	SMASK2X	2c •	16,16	16,16	Screenmask 2: min, max boundary (screen absolute: X11 directionality).
0x130C	SMASK2Y	2c •	16,16	16,16	Screenmask 2: min, max boundary.
0x1310	SMASK3X	2c •	16,16	16,16	Screenmask 3: min, max boundary (screen absolute: X11 directionality).
0x1314	SMASK3Y	2c •	16,16	16,16	Screenmask 3: min, max boundary.
0x1318	SMASK4X	2c •	16,16	16,16	Screenmask 4: min, max boundary (screen absolute: X11 directionality).
0x131C	SMASK4Y	2c •	16,16	16,16	Screenmask 4: min, max boundary.
0x1320	TOPSCAN		10	10	Y address for top of screen scan line, (0,1023=top,bottom of framebuffer).
0x1324	XYWIN	2c	16,16	16,16	Screen X,Y offset for window relative addressing and coordinate biasing.
0x1328	CLIPMODE	•	13	13	CID, screenmask mode and enable bits.
0x132C	STALL1	•	#		Forces stall at the output of GFIFO until graphics pipeline is idle.
0x1330	CONFIG	⊗	21	21	Miscellaneous configuration bits.
0x1338	STATUS	⊗		20	Chip busy and FIFO status register. Reading clears interrupt status bits.
0x133C	USER_STATUS	⊗		20	Chip busy and FIFO status register for User code. Non-destructive reads.
0x1340	DCBRESET	⊗	#		Resets the DCB bus state machine and flushes BFIFO.

Table 7: REX3 host visible registers.

3.1.1 Control Register Bit Definitions

The following tables outline the definition of REX3 control register bits. Refer to related sections in Chapter 3 for discussion of the REX3 drawing, masking, and pixel I/O programming interface.

3.1.1.1 DRAWMODE0 Register

Bits	Name	Access	Init	Active	Description
1:0	OPCODE(1:0)	R/W	0x0		Primitive function command.
4:2	ADRMODE(2:0)	R/W	0x0		Primitive function addressing mode.
5	DOSETUP	R/W	0x0	H	Enables SPAN/BLOCK/I_LINE/F_LINE/A_LINE iterator setup.
6	COLORHOST	R/W	0x0	H	RGB/CI draw source: 0=DDAs; 1=HOSTRW register.
7	ALPHAHOST	R/W	0x0	H	Alpha draw source: 0=DDA; 1=HOSTRW register.
8	STOPONX	R/W	0x0	H	Specifies execution tests for X coordinate endpoint reached.
9	STOPONY	R/W	0x0	H	Specifies execution tests for Y coordinate endpoint reached.
10	SKIPFIRST	R/W	0x0	H	Disable start-point draw (lines only).
11	SKIPLAST	R/W	0x0	H	Disable endpoint draw, freeze iterators at endpoint (lines only).
12	ENZPATTERN	R/W	0x0	H	Patterning enable.
13	ENLSPATTERN	R/W	0x0	H	Line stipple pattern enable.
14	LSADVLAST	R/W	0x0	H	Enables stipple advance at end of line.
15	LENGTH32	R/W	0x0	H	Limits draw primitive to 32 pixels.
16	ZPOPAQUE	R/W	0x0	H	Enables opaque (vs. transparent) stipple mode for ZPATTERN.
17	LSOPAQUE	R/W	0x0	H	Enables opaque (vs. transparent) stipple mode for LSPATTERN.
18	SHADE	R/W	0x0	H	Enables linear shader R,G,B,A/CI DDAs.
19	LRONLY	R/W	0x0	H	Aborts primitive if initial XSTARTI > XENDI.
20	XYOFFSET	R/W	0x0	H	Add XYMOVE to XSTART,YSTART for draw relative operations.
21	CICLAMP	R/W	0x0	H	Enables CI shader DDA over/underflow clamping for CI pixels.
22	ENDPTFILTER	R/W	0x0	H	Enables hardware endpoint filtering (A_LINE only).
23	YSTRIDE	R/W	0x0	H	Enables Y axis increment/decrement by 2

Table 8: DRAWMODE0 register

Value	Name	Description
00	NOOP	Do nothing.
01	READ	Host read from framebuffer using ADRMODE.
10	DRAW	Draw into framebuffer using ADRMODE.
11	SCR2SCR	Framebuffer to framebuffer copy, (valid with ADRMODE=SPAN/BLOCK).

Table 9: DRAWMODE0 OPCODE(1:0) definition.

Value	Name	Description
000	SPAN	Span (or point) addressing mode.
001	BLOCK	Block addressing mode, advance Y and restore XSTART at end of span.
010	I_LINE	Bresenham line addressing mode, integer endpoints.
011	F_LINE	Bresenham line addressing mode, fractional endpoints.
100	A_LINE	Antialiased Bresenham line addressing mode .

Table 10: DRAWMODE0 ADRMODE(2:0) definition.

3.1.1.2 DRAWMODE1 Register

Bits	Name	Access	Init	Active	Description
2:0	PLANES(2:0)	R/W	0x1		Specifies which framebuffer planes enabled for R/W access: 000 none 001 R/W RGB/CI planes 010 R/W RGBA planes 100 R/W OLAY planes 101 R/W PUP planes 110 R/W CID planes
4:3	DRAWDEPTH(1:0)	R/W	0x0		Drawn depth of framebuffer PLANES, *not including alpha*: 00 Depth = 4 bits 01 Depth = 8 bits 10 Depth = 12 bits 11 Depth = 24 bits
5	DBLSRC	R/W	0x0		Double-buffer mode pixel read source buffer, (0= buffer0).
6	YFLIP	R/W	0x0	H	Enable GL Y coord mapping: 0=origin at upper left; 1=origin at lower left.
7	RWPACKED	R/W	0x0	H	Enables pixel packing for HOSTRW access.
9:8	HOSTDEPTH(1:0)	R/W	0x0		HOSTRW pixel packing/unpacking: 00 Pixel depth = 4 bits (1-2-1 BGR or 4 CI) 01 Pixel depth = 8 bits (3-3-2 BGR or 8 CI) 10 Pixel depth = 12 bits (4-4-4 BGR or 12 CI) 11 Pixel depth = 32 bits (8-8-8-8 ABGR)
10	RWDOUBLE	R/W	0x0	H	Enables double word (64-bit) host transfers (vs. 32-bit single word). HOSTRW(0,1) format for host framebuffer DMA/PIO only.
11	SWAPENDIAN	R/W	0x0	H	OpenGL SWAP_ENDIAN pixel storage attribute. When true, HOSTRW short and long packed pixel data have their byte ordering swapped.
14:12	COMPARE(2:0)	R/W	0x7		Color compare and AFUNCTION condition specifier, (conditions OR'ed).
	COMPARE(2)	R/W		H	Enable compare condition: src > dest.
	COMPARE(1)	R/W		H	Enable compare condition: src = dest.
	COMPARE(0)	R/W		H	Enable compare condition: src < dest.
15	RGBMODE	R/W	0x1	H	Selects RGB (vs. CI) shade, round, dither, compare, and clamp modes.
16	DITHER	R/W	0x0	H	Enables dithering.
17	FASTCLEAR	R/W	0x1	H	Enables fast-clear write mode when CID checking disabled (CLIPMODE CIDMATCH = 0xF). Valid with DRAW SPAN/BLOCK only.
18	BLEND	R/W	0x0	H	Enable blendfunction.
21:19	SFACTOR(2:0)	R/W	0x0	H	Blendfunction source blending factor, (see Table 13).
24:22	DFACTOR(2:0)	R/W	0x0	H	Blendfunction destination blending factor, (see Table 14).
25	BACKBLEND	R/W	0x0	H	Enable COLORBACK to be used for blendfunction destination color.
26	PREFETCH	R/W	0x0	H	Enables host framebuffer pixel prefetch mechanism for PIO reads.
27	BLENDALPHA	R/W	0x0	H	Selects SFACTOR BF_SA source alpha: '1' = source alpha, '0' = 1.0.
31:28	LOGICOP(3:0)	R/W	0x3		Logical operation type, (see Table 12).

Table 11: DRAWMODE1 register.

Value	Symbol	Operation
0000	LO_ZERO	0
0001	LO_AND	src AND dst
0010	LO_ANDR	src AND (NOT dst)
0011	LO_SRC	src
0100	LO_ANDI	(NOT src) AND dst
0101	LO_DST	dst
0110	LO_XOR	src XOR dst
0111	LO_OR	src OR dst
1000	LO_NOR	NOT (src OR dst)
1001	LO_XNOR	NOT (src XOR dst)
1010	LO_NDST	NOT dst
1011	LO_ORR	src OR (NOT dst)
1100	LO_NSRC	NOT src
1101	LO_ORI	(NOT src) OR dst
1110	LO_NAND	NOT (src AND dst)
1111	LO_ONE	1

Table 12: DRAWMODE1 LOGICOP(3:0) definition.

Value	Symbol	Source Blending Factor
000	BF_ZERO	0
001	BF_ONE	1
010	BF_DC	normalized[destination color (or COLORBACK)]
011	BF_MDC	1 - normalized[destination color (or COLORBACK)]
100	BF_SA	normalized[source alpha]
101	BF_MSA	1 - normalized[source alpha]

Table 13: DRAWMODE1 SFACTOR(2:0) definition.

Value	Symbol	Destination Blending Factor
000	BF_ZERO	0
001	BF_ONE	1
010	BF_SC	normalized[source color]
011	BF_MSC	1 - normalized[source color]
100	BF_SA	normalized[source alpha]
101	BF_MSA	1 - normalized[source alpha]

Table 14: DRAWMODE1 DFACTOR(2:0) definition.

3.1.1.3 LSMODE Register

Bits	Name	Access	Init	Active	Description
7:0	LSRCOUNT(7:0)	R/W			Current value of LSREPEAT down counter, (advance LS pattern when 0).
15:8	LSREPEAT(7:0)	R/W			Line stipple pattern (bit expansion) repeat factor, (1 ≤ LSREPEAT ≤ 255).
23:16	LSRCNTSAVE(7:0)	R/W			Copy of LSRCOUNT, (updated with write to LSSAVE register address).
27:24	LSLENGTH(3:0)	R/W			Length of LSPATTERN, from 17 to 32, starting with msb, (0000=17).

Table 15: LSMODE register.

3.1.1.4 CLIPMODE Register

Bits	Name	Access	Init	Active	Description
4:0	ENSMASK(4:0)	R/W	0x0	H	Individual enables for SMASK4:0.
8:5	<reserved>	R/W	0x0		
12:9	CIDMATCH(3:0) CIDMATCH(3) CIDMATCH(2) CIDMATCH(1) CIDMATCH(0)	R/W	0x0	 H H H H	CID codes to compare, results OR'ed: selects CID code 11 for CID check selects CID code 10 for CID check selects CID code 01 for CID check selects CID code 00 for CID check

Table 16: CLIPMODE register.

3.1.1.5 STATUS Register/USER_STATUS Register

Bits	Name	Access	Init	Active	Description
2:0	VERSION(2:0)	R			Revision code, (001 = 1st revesion).
3	GFXBUSY	R	0x0	H	Indicates graphics pipeline not idle or GFIFO not empty.
4	BACKBUSY	R	0x0	H	Indicates backend pipeline not idle or BFIFO not empty.
5	VRINT	R		H	Video controller vertical retrace interrupt. VR_INT_N falling-edge detected, generating VV_INT interrupt. Cleared by the read of STATUS, not cleared by the read of USER_STATUS.
6	VIDEOINT	R		H	Video option interrupt VIDEO_INT_N status, generating VV_INT interrupt.
12:7	GFIFOLEVEL(5:0)	R	0x00		Current GIO graphics FIFO level, (0 = empty FIFO).
17:13	BFIFOLEVEL(4:0)	R	0x00		Current display bus FIFO level, (0 = empty FIFO).
18	BFIFO_INT	R	0x0	H	BFIFOLEVEL above BFIFODEPTH interrupt was generated. Cleared by the read of STATUS, not cleared by the read of USER_STATUS. Provides sticky status of BFIFO above FIFO_INT_N interrupts.
19	GFIFO_INT	R	0x0	H	GFIFOLEVEL above GFIFODEPTH interrupt was generated. Cleared by the read of STATUS, not cleared by the read of USER_STATUS. Provides sticky status of GFIFO above FIFO_INT_N interrupts.

Table 17: STATUS register.

3.1.1.6 CONFIG Register

Bits	Name	Access	Init	Active	Description
0	GIO32MODE	R/W	0x0	H	When set, the REX3 will assume that the information sent by the host during the byte count cycle of a GIO bus transfer is in GIO32 bus format. When cleared, GIO64 byte count cycles are assumed. When GIO32 mode is selected, EXTREGXCVR should also be set, and BUSWIDTH should be cleared.
1	BUSWIDTH	R/W	0x0		Denotes the physical width of the GIO64 bus. 1=64 bits, 0=32 bits
2	EXTREGXCVR	R/W	0x1		Denotes the presence of external registered transceivers separating the pipelined from the non-pipelined GIO64 bus.
6:3	BFIFODEPTH(3:0)	R/W	0x8		Display bus FIFO high/low trigger depth: stalls GIO bus and enables GIO timeout counter when BFIFOLEVEL≥BFIFODEPTH and BFIFABOVEINT is set. Host FIFO interrupt is generated when BFIFOLEVEL becomes less than BIFODEPTH and BFIFOABOVEINT is cleared.
7	BFIFOABOVEINT	R/W	0x1		Display bus FIFO interrupt select. When set, GIO bus stalls and GIO timeout counter is enabled when BFIFOLEVEL≥ BFIFODEPTH. When cleared and BFIFOLEVEL becomes less than BIFODEPTH, a host FIFO interrupt is generated.
12:8	GFIFODEPTH(4:0)	R/W	0x10		GIO graphics FIFO high/low trigger depth: stalls GIO bus and enables GIO timeout counter when GFIFOLEVEL≥GFIFODEPTH and GFIFOABOVEINT is set. Host FIFO interrupt is generated when GFIFOLEVEL becomes less than GIFODEPTH and GFIFOABOVEINT is cleared.
13	GFIFOABOVEINT	R/W	0x1		GFIFO interrupt select. When set, GIO bus stalls and GIO timeout counter is enabled when GFIFOLEVEL≥ GFIFODEPTH. When cleared and GFIFOLEVEL becomes less than GIFODEPTH, a host FIFO interrupt is generated.
16:14	TIMEOUT(2:0)	R/W	0x0		GIO bus timeout interval: 000=0.96µsec, 001=1.44µsec... 111=4.32µsec. Timeout generates host FIFOFULL interrupt and unstalls GIO bus.
19:17	VREFRESH(2:0)	R/W	0x1		Number of VRAM refresh cycles per transfer cycle, 000=refresh disabled.
20	FB_TYPE	R/W		H	Framebuffer fastclear column mask mode select: 0 TI mode: replicate 4-bit comumn mask 1 non-TI mode: zero-fill comumn mask 4 msbs

Table 18: CONFIG register.

3.1.1.7 DCBMODE Register

Bits	Name	Access	Init	Active	Description
1:0	DATAWIDTH(1:0)	R/W	0x0		Width of the data being transferred for each DCBDATA0 or DCBDATA1 word. Needed to support the OpenGL SWAP_ENDIAN construct, and to allow RGB triplets to be packed into words. 00 4 bytes 01 1 byte 10 2 bytes 11 3 bytes
2	ENDATAPACK	R/W	0x0	H	Determines the use of the DATAWIDTH field for packed/unpacked data. When set, all bytes addressed by DCBDATA will be transferred. When clear, only DATAWIDTH bytes in each addressed DCBDATA word will be transferred
3	ENCRSINC	R/W	0x0	H	Enables DCB_CRS(2:0) auto-increment following each DCB transfer.
6:4	DCBCRS(2:0)	R/W	0x0		Display bus control register select address.
10:7	DCBADDR(3:0)	R/W	0xF		Display bus slave address.
11	ENSYNCACK	R/W	0x0	H	Enables display control bus protocol with synchronous acknowledge of data transfer with DCB_ACK_N
12	ENASYNCACK	R/W	0x0	H	Enables display control bus protocol with asynchronous acknowledge of data transfer (four-edge handshake protocol with DCB_CS_N and DCB_ACK_N).
17:13	CSWIDTH(4:0)	R/W	0x0		# GIO_CLK cycles width for DCB_CS_N.
22:18	CSHOLD(4:0)	R/W	0x0		# GIO_CLK cycles hold time before DCB_CS_N de-asserted.
27:23	CSSETUP(4:0)	R/W	0x0		# GIO_CLK cycles setup before DCB_CS_N asserted.
28	SWAPENDIAN	R/W	0x0	H	OpenGL SWAP_ENDIAN pixel storage attribute. When true, DCBDATA short and long packed pixel data have their byte ordering swapped.

Table 19: DCBMODE register.

3.2 Coordinate System

There are several ways to describe the coordinate system in REX3. First, its framebuffer contains a region of 1280 x 1K pixels which can be displayed on a monitor. To the right of this area is an “off-screen” or non-displayed section of memory which is 64 pixels wide, adjacent to the right edge of displayable memory.

The physical coordinates for this displayable space are: 4K,4K for the upper left corner, and 4K+1279, 5K-1 for the lower right corner. The lower right corner becomes 5K+63 including the off-screen memory space.

The X11 window system normally considers the upper left region of displayable memory as being at 0,0; in order to achieve this with REX3, the window relative bias register XYWIN is loaded with a 4K,4K offset value. This allows the X11 coordinate system to be used directly with REX3, which supports the full 16b,16b addressability (-32K through +32k-1 along each axis), without the need for host clipping.

The GL implementation running on REX3 relies on float-to-fixed point coordinate transformation shortcuts which result in biased coordinates; this bias is hardwired within REX3 to a value of 4K,4K. Assuming that the GL makes use of exactly this bias value, applicatons which rely on transformed coordinates do not need to load XYWIN with the 4K,4K bias; instead, the XYWIN register is used for window-relative offset, from the displayed screen origin to the origin of the GL window of interest: xrel, yrel. If the GL uses a bias differing from 4K then XYWIN must be explicitly biased by the value (GL bias minus 4K) so as to yield values of: (xrel + GL bias - 4K), (yrel + GL bias -4K).

The GL relies on a subset of the X11 address space, limited to 8K x 8K (0 thru 8K-1 along each axis, where our origin is centered at or about 4K,4K, depending on the bias mentioned above) . When the Y axis is increasing in downward direction (X11 system, which some GL code has been modified to conform to), the DRAWMODE1 bit YFLIP is set to zero, and all window and screen origins are referenced to the upper left of respective area rectangles. When the Y axis increases upward (the usual GL convention) the DRAWMODE1 bit YFLIP is set to one; now all window and screen origins are referenced to the lower left of respecitve area rectangles. In this case, XYWIN must be set to (0+xrel, 9K-1-yrel), where xrel,yrel are signed distance from screen origin to window origin (all lower left here), assuming 4K,4K biasing of the X,YSTART and X,YEND coordinate values. This becomes a little more complex if biasing differs: (xrel + GL bias - 4K), (5K + GL bias -1 - yrel).

Subpixel positioning of XSTART, YSTART, XEND, YEND of 4 bits are supported for line drawing, including antialiasing and endpoint filtering.

An arbitrary signed offset may be applied to XSTART, YSTART via setting DRAWMODE0 bit XYOFFSET. The signed value in XYMOVE is then applied. (Note: XYOFFSET should never be set for screen-to-screen copy mode, which uses XYMOVE for its own offset between source and destination.)

3.3 Clipping and Masking

Framebuffer values are conditionally written as a function of sector clipping, screen masking, CID masking, afunction, and color compare. (Transparent patterning also conditions the writes, see 3.8.1, Patterning and Stippling.) Bits within each write are masked by the 24b WRITEMASK register (in this case, ‘0’ means don’t write).

Sector clipping is performed internally by REX3 so as to cull any writes which are outside the legal drawing area, defined by VRAM space. This space is described in Section 3.2, Coordinate System. Note that reads are not culled, so as to maintain simplified read behavior for DMA and host reads.

Screen masking is performed via the 5 sets of rectangles described by the SMASK registers. These are controlled by the CLIPMODE register, to define invocation of each mask. All screenmasks are selectively invoked by the ENSMASK field, and determine whether a given pixel is outside its area. SMASK0 is a GL mask, which clips drawing outside its region; it is window-relative (affected by XYWIN YFLIP) and conforms to GL coordinate behavior (ust be biased in the same way as X and Y coordinates: see previous chapter). Locations outside are masked. SMASKS1-4 are X11 general-purpose masks, not window-relative; coordinates are absolute, and unaffected by XYWIN or YFLIP, requiring the host to prebias them with the 4K,4K offset.

Overall, a screenmasked pixel may be written iff it is:

{(inside any of enabled screenmasks1-4) or (all screenmasks 1-4 disabled)} AND { inside screenmask0 or screenmask0 disabled}.

Reads are never screenmasked.

CID masking is invoked on writes to framebuffer whenever CLIPMODE register bits CIDMATCH are not '1111'. In that case, CID location corresponding to each framebuffer address is read and compared with CIDMATCH field. If there is a match, the framebuffer write is permitted. CID checking is never performed on framebuffer reads.

Afunction, or alpha function, is a GL feature which allows the user to inhibit framebuffer writes for specified compare relationship between source alpha (either from DDA or host, for bit ALPHAHOST=0,1 respectively) and a specified reference alpha stored in register ALPHAREF. The compare operator is given in the DRAWMODE1 register COMPARE field.

Color compare is a peculiar feature of old GL releases for aiding in antialiased color index line drawing. For RGBMODE=0, linedraw antialiased with DRAWMODE1 bits COMPARE not '111' will conditionally write based on source value, destination value comparison.

Writemasking is specified for 24b field and must match the bit positioning as described in Section 3.9, Framebuffer Formats (exception: writes to AUX planes only use lower 12b of the WRITEMASK). The WRITEMASK register is also used to specify double buffering, see Section 3.7, Double Buffering, for more details.

3.4 Iterator Overview

There are four types of hardware iterators in REX3: D,S1,S2; R,G,B,A/X; LSPATTERN,ZPATTERN; X,Y. First, the D term Bresenham error stepping iterator for controlling advance of X,Y major axis for Bresenham linedraw. Additional iterators are provided for antialiased linedraw, to control pixel coverage: S1 calculates the coverage value, in conjunction with S2 which determines secondary pixel direction along the minor axis. Second, DDA iterators for CI and R,G,B,A values for all planes. Third, recirculating iterators for line stipple pattern (LSPATTERN) and polygon or Z mask pattern (ZPATTERN). Fourth, integer increment/decrement iterators for X,Y of lines, spans and blocks.

The Bresenham stepper calculates one pixel address and coverage per clock. The Y iterator calculates one value per clock (+/-1). The shader DDA calculates one or two pixel values per clock (+1,+2 times slope). The pattern iterators calculate one, two, or four values per clock. The X iterator calculates one, two, four, or 32 values per clock (+/-1, +2, +/-4, +32).

Values per pipeline clock are determined as follows: aliased linedrawing, one/clk; antialiasaed linedrawing, two/3 clks; shaded DDA spans/blocks, two/clk; flat DDA spans/blocks, four/clk; screen-to-screen block copy, per read or write: four/clk; fast clear spans/blocks, 32/clk; host/DMA reads, one to four/clk, and writes, one or two/clk, depending on packed number of values per bus transfer specified in DRAWMODE1. For more information on these modes, see Section 3.5, Framebuffer Access Modes.

Each of these iterators can be loaded with new starting values at the start of each primitive; they compute successive values within that primitive, for multiple-pixel primitives. Normally each iterator will retain, after primitive completion, the state corresponding with the point after that last drawn. (H/W Note: for back-to-back primitives, the completion of the first overlaps with the start of the next, so that the iterator is never loaded with the final state of first primitive, should the following primitive load the same iterator.)

Special mode bits are provided so that connected lines, which cover vertex of intersection twice (once per iterated line), don't cause problems. For GL, stippled, connected lines would normally advance line stipple twice at intersection; to prevent this, DRAWMODE0 bit LSADVLAST is set to zero, to inhibit LSPATTERN advance at end of primitive. The pixel of intersection is, however, drawn twice. This is not desired for X11, where lines could be drawn with LOGICOP=xor: then drawing same location twice gives different value than drawing once! To handle this, bit SKIPLAST is set to inhibit drawing of endpoint of a line, and retain X,Y state of the endpoint. This has the additional advantage of eliminating the need to reset the X,Y starting values for successive connected lines (e.g., for integer endpoint case). A SKIPFIRST bit is provided to skip first pixel of antialiased line, should host prefer to do the endpoint filtering itself. When this bit is set, the X,Y iterators again retain the state of the endpoint. Note: “first” pixel is the first pixel per “GO” event; “last” pixel is (are) that corresponding to the major axis end value.

3.5 Framebuffer Access Modes

The framebuffer may be accessed as points, lines, spans, or blocks of data. Additionally, REX3 provides autoincrementing address features so that a line may be accessed as successive points (or patterned segments, for writes); a span as successive points or segments; and a block as successive points, segments, or spans. Here the term “segment” is loosely used to refer to a fixed length string of pixels, usually a subset of the primitive (line, span, or block row) being iterated. In the following subsections, “Segments I” refers to packed host data, using the HOSTRW registers with COLORHOST or ALPHAHOST set; “Segments II” refers to remaining cases which have DRAWMODE0 bit LENGTH32 set.

3.5.1 Lines: Overview

Line mode is indicated by DRAWMODE0 register field ADRMODE=I_LINE, F_LINE, A_LINE. The Bresenham setup is performed by REX3. This may include subpixel and antialiasing coverage calculations. For information on integer versus subpixel positioned cases, see Section 3.6, Linedrawing.

Line drawing is specified by DRAWMODE0 field OPCODE=draw; reading a line by OPCODE=read.

The endpoint of each line is not drawn if SKIPLAST=1; in this case the X,Y start state remains at the endpoint; the startpoint is not drawn if SKIPFIRST=1 (note: SKIPFIRST is used at start of each primitive, so it should be cleared for second and later segments or points for case of primitive decomposed as such).

3.5.1.1 Line Draw or Host Read: Points

A line can be read or written as sequential points by setting STOPONX=STOPONY=0. The state of XSTART, YSTART is post-iterated each access, in accordance with the Bresenham algorithm.

Prior to the first point, the host must write to address=SETUP to have REX calculate octant and Bresenham terms.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	0	0	0	0

3.5.1.2 Line Draw: Segments II

Here 32 pixels are drawn per primitive, until end condition reached. This is useful when patterning (LSPATTERN, ZPATTERN) using the 32b pattern/stipple/z masking registers.

Prior to drawing the first segment, a write to address=SETUP is required, to perform octant and error term initialization.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	1	0	1

3.5.1.3 Line Draw: Full Line

This primitive draws a line as one command.

.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	1	1	0

3.5.2 Point Draw or Read

A point is described by a XSTART, YSTART pair. This may be packed into a single word as a pair of integer values (XYSTARTI), or as two words.

Whether reading or writing points, the DRAWMODE0 register is initialized with ADRMODE=block, DOSETUP=0.

A point is written using OPCODE=draw. A collection of points as X,Y pairs per transfer may be written as a DMA to rapidly construct an arbitrary, monochrome shape, such as a circle. The DRAWMODE0 bit XYOFFSET may be used to add XYMOVE to these X,Y values. A point is read using OPCODE=read.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	--	YS	--	0	0	0	0

3.5.3 Spans: Overview

Unlike points, spans require an X endpoint; DRAWMODE field ADRMODE=span is set for spans, indicating that X stepping direction is to be implied by sign of {XEND minus XSTART}. Currently there are not plans to support Right-to-Left spans.

Spans may be culled by use of the DRAWMODE0 LRONLY bit: it aborts span primitives where {XEND < XSTART}, allowing Left-to-Right Only to draw.Spans are drawn using OPCODE=draw; they are read using OPCODE=read.

User beware: the graphics state at the end of span iteration is determined by granularity of X coordinate stepping.

3.5.3.1 Span Draw or Host Read: Segments I

This span drawing mode uses pixel values from host or DMA obtained through the HOSTRW registers. Depending on the packing format, this could be one to sixteen pixels per 64b word, or to eight per 32b word. DRAWMODE0 bit COLORHOST (ALPHAHOST) = 1 to indicate pixel source is not DDA.

This mode is also used for host reads of a span.

The host must, in advance, issue a write to address=SETUP in order to have REX calculate quadrant.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	--	0	0	0	0

3.5.3.2 Span Draw: Segments II

This span drawing mode unlike the above, uses the DDA to calculate pixel value. It stops after 32 pixels have been iterated, or the X endpoint is reached, whichever comes first. This is useful when using the LS-PATTERN or ZPATTERN features, for non-repeating pattern values, such as Z buffering or arbitrary X11 patterning. For spans of less than 33 pixels in length, the Full Span mode may be used instead.

The host must, in advance, issue a write to address=SETUP in order to have REX calculate quadrant.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	--	1	0	0	1

3.5.3.3 Span Draw or DMA Read: Full Span

This span mode draws a span as a single primitive.

A monochrome shape which is decomposed into a list of spans can be written using 64b writes as XYENDI#XYSTARTI. Shape would be redrawn at various locations via use of DRAWMODE0 bit XYOFFSET and XYMOVE.

This mode is used for DMA framebuffer reads of a span.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	--	1	0	1	0

3.5.4 Blocks: Overview

Block mode is specified by DRAWMODE0 field OPCODE=block. Drawing is performed on a span-by-span basis. At the end of each span, the XY DDA steps the YSTART value and resets the XSTART value to that from XSAVE; XSAVE is written whenever XSTART is updated by host. In addition to the coordinates needed for a span, the block mode also requires the YEND value. Stepping in the Y direction is implied by sign of {YEND minus YSTART}. As mentioned before, there is not support for Right-to-Left spans.

Block draw is performed with OPCODE=draw; reads via OPCODE=read.

Polygon filling may use block draw mode to automatically step Y per span; host then sets XSTART, XEND per span. YEND is set initially to an extreme so as to simply imply the direction of Y axis stepping per row. STOPONY=0 for this mode, which means the first three block draw cases below can support this.

3.5.4.1 Block Draw or Host Read: Segments I

Block draw in segments from host/memory is done as a sequence of span segment writes; a segment which exceeds the block width is truncated so that a segment is never covering two block rows (spans). Host must set COLORHOST or ALPHAHOST =1 for this mode. See Span Draw, Segments I for more information.

This mode is also used for host reads of framebuffer block.

The host must, in advance, issue a write to address=SETUP in order to have REX calculate quadrant.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	0	0	0	0

3.5.4.2 Block Draw: Segments II

Each primitive draws 32 pixels, maximum. Used in conjunction with LSPATTERN, ZPATTERN. The block mode makes this useful for large character or other bit expansion drawing. Again, a primitive (segment) is truncated at the end of each row, and never applied to two rows. See Span Draw, Segments II for more information.

The host must, in advance, issue a write to address=SETUP in order to have REX calculate quadrant.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	0	0	1

3.5.4.3 Block Draw or Stride DMA Read: Spans

The block is drawn as a span per primitive, with the XY DDA performing post-increment of Y and reset of X. Useful for characters of < 33 pixels width, using bit expansion of LSPATTERN, ZPATTERN.

Stride DMA reads use this mode.

The host must, in advance, issue a write to address=SETUP in order to have REX calculate quadrant.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	0	0	0

3.5.4.4 Block Draw or Linear DMA Read: Full Block

Draws an upright rectangular region as a single primitive.

Linear DMA read uses this mode for full block.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	1	1	0

3.5.5 Fast Clear

This drawing mode provides 4x rate, for fast area clear. No support for any per pixel operations, such as shade, stipple, dither, blend. Flat fill only, via value previously written by host into the COLORVRAM register. The loading of COLORVRAM must be performed after DRAWMODE1 fields RGBMODE and DRAWDEPTH have been set. In addition to the bits shown below, the DRAWMODE1 bit FASTCLEAR must be set. DRAW-

MODE0 register OPCODE=draw, ADRMODE=block or span must be used. CID checking is not allowed for this drawing mode. Spans must be Left to Right.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	1	1	0

3.5.6 Screen-to-Screen Move

Screen-to-screen copy is specified by DRAWMODE0 field OPCODE=Scr2Scr and ADRMODE=block or span. The command setup is similar to the Full Block or Full Span draw, with the addition of a signed offset to destination (**unlike REX1, which was offset to source**) specified by XYMOVE. This offset is with respect to the window origin, and is therefore interpreted with respect to YFLIP. Block move supports Right-to-Left spans. The host must order the X,Y start/end points (hence, quadrant) such that the copy does not destroy itself in the process, for source area overlapping destination. Using this mode with XYMOVE=0 will be slower than its obvious optimization. DRAWMODE0 bit XYOFFSET should be 0.

XS	XE	YS	YE	StoponX	StoponY	Dosetup	Length32
XS	XE	YS	YE	1	1	1	0

3.6 Line Draw Instructions

3.6.1 Bresenham Aliased Line Draw Instructions

Newport is the first system that uses exclusively Bresenham algorithms as opposed to DDA. The main reason is that Bresenham is has infinite precision whereas DDA cannot guarantee predictability at any number of bits of fraction.The second reason is that aliased lines have a much shorter setup since there is no division for slope computation. The third reason is that by using Bresenham we can unify the hardware and the algorithms for drawing both aliased and antialiased lines and polygons. The BRESROUND field of the DRAWMODE0 register decides how the comparison between d and 0 should be executed:

```
If BRESROUND=1 Then    // BRESROUND has 8 bits-one for each octant//
    If  d < 0 Then           // this branch is executed for d < 0 //
    Begin
      ...........
   End Else                  // this branch is executed for d >= 0 //


If BRESROUND=0 Then
    If  d =< 0 Then          // this branch is executed for d =< 0 //
    Begin
      ...........
   End Else                  // this branch is executed for d > 0 //
```

By appropriate programming of the BRESROUND bits we can produce hysteresis-free lines.

3.6.1.1 I_line(x1,y1,x2,y2,SKIPLAST,SKIPFIRST)

integer: x1,y1,x2,y2

This is an aliased line with integer endpoints The intent is to have maximum performance at the expense of line quality. Bresenham algorithm allows for very short setup (no multiplication/division) and for reproducibility of results (always touches the same pixels). REX3 computes the octant.

The performance is limited by :

-the time for passing the arguments from the CPU to REX3 over GIO bus

-the time for generating the setup values by REX3 : d=2dy-dx, etc (5 clocks)

-time for iterating a new coordinate (1 clock)

Since the coordinates are integer there are no precision requirements - Bresenham algorithm with integer endpoints is infinitely precise. If SKIPFIRST=TRUE the starting point (x1,y1) is not drawn by REX3.
If SKIPLAST=TRUE the endpoint (x2,y2) is not drawn by REX3.

Register-level description:

XYSTARTI=x1,y1 //only the listed registers must be saved at context switch because : //

XYENDI=x2,y2 //-all variables used by Bresenham are derived from the input variables//

DRAWMODE: OPCODE=I_line

Context-switched registers:

XSTART=x_current

YSTART=y_current

XEND=x2 // necessary for computing the pixel count //

YEND=y2 // necessary for computing the pixel count //

BRESD=d // current d value //

BRESEOCTINC1=octant,incr1 // octant + incr1 for d //

BRESINC2=incr2 // incr2 for d //

3.6.1.2 F_Line(x1,y1,x2,y2,SKIPLAST,SKIPFIRST)

fixed : x1,y1,x2,y2

This is an aliased line with fractional endpoints The intent is to have maximum performance at the expense of line quality. Bresenham algorithm allows for very short setup (two multiplications and no division) and for reproducibility of results (always touches the same pixels). REX3 or the CPU computes the octant . The performance is limited by :

-the time for passing the arguments from the CPU to REX3 over GIO bus

-the time for generating the setup values by REX3 : d=3dy-2dx+2(dx*y_fract-dy*x_fract) .All GL linedrawing primitives must use 3dy-2dx due to the way GL views the coordinate system as opposed to X. (12 clocks)

-time for iterating a new coordinate (1 clock)

-time for drawing the fractional coverage endpoints

A serial multiplier is necessary for computing d..Since the multiplicand involved (x_frac,y_frac) has very few bits a serial multiplier executes the required multiplication in few cycles. If SKIPFIRST=TRUE the starting point (x1,y1) is not drawn by REX3. If SKIPLAST=TRUE the endpoint (x2,y2) is not drawn by REX3.

Register-level description:

XSTART=x1 //fixed point number in 16.4 format//

YSTART=y1 //fixed point number in 16.4 format//

XEND=x2 //fixed point number//

YEND=y2 //fixed point number//

DRAWMODE: OPCODE=F_line

Context-switched registers:

XSTART=x_current

YSTART=y_current

XEND=x2

YEND=y2

BRESD=d

BRESOCTINC1=octant,incr1

BRESINC2=incr2

3.6.2 Bresenham Antialiased Line Draw Instructions

3.6.2.1 A_Line(x1,y1,x2,y2,e1,aa_table,SKIPFIRST,SKIPLAST)

fixed : x1,y1,x2,y2,e1

array : aa_table // angle-compensated table of pixel coverages indexed by s //

THIS PRIMITIVE WILL ALSO BE USED FOR GENERATING ANTIALIASING EDGES BY MASKING OUT THE TOP OR BOTTOM HALF WITH THE HELP OF THE ZPATTERN (BOTTOM HALF)), LSPATTERN (TOP HALF) REGISTERS.
This is an anti-aliased line with fractional endpoints and with angle compensation but without any endpoint filtering.It has INFINITE precision in terms of pixel positioning (exactly like I_LINE) since it doesn't rely on a DDA algorithm in terms of position determination. The intent is to generate high quality lines at 80-90% the speed of aliased blended lines. The width of the line is restricted to 1 - for wider lines (and for polygons) the Bresenham antialiasing edge (see 3.6.2.3) should be used. Two pixels (in the minor axis direction) are interpolated at each major axis iteration . This approach allows for line intensity independent of line angle (i.e. independent of pixel density). The performance is limited by :

-the time required for the CPU to compute the slope e1 and to find the aa_table (which is a function of e1) in memory

-the time for passing the arguments from the CPU to REX3 over GIO bus

-the time for generating the setup : d=3dy-2dx+2(dx*y_fract-dy*x_fract) ,s=y_fract+e1*(.5-x_fract)-.5

s*dx=s*2dx=dy-dx+2(dx*y_fract-dy*x_fract)

-REX3 time for iterating two new coordinates (closely related to each other) (3 clocks/pair)

A serial multiplier is necessary for computing d and s.If SKIPLAST=TRUE the endpoint (x2,y2) is not drawn by REX3. If SKIPFIRST=TRUE the first point (x1,y1) is not drawn by REX3. The algorithm draws two pixels (T and S) at each iteration.The coverages for these two pixels are derived by indexing into the AWEIGHT table with a function of s as described below.The AWEIGHT table needs to be reloaded for every change in the line slope e1. Note that here s_frac represents the absolute value of the fractional part of s.

If 0=<s=<1 Then

Begin

coverage_T=f(s)=f(s_frac) // s_frac=Fraction(s)//

coverage_S=f(1-s)=f(1-s_frac)=f(~s_frac) //~s_frac=1.0-s_frac //

End

If -1=<s<0 Then

Begin

coverage_T=f(1+s)=f(~s_frac)

coverage_S=f(-s)=f(s_frac)

End

The case for d>0 that includes the subcases 0<s<1 and 1<s<2 reduces to the above case if we manage to arrange for -1<s<1.This is done by adding e2=e1-1 prior to rendering the pixels.As it can be seen the AWEIGHT table is indexed with s_frac and ~s_frac (one's complement of the fractional portion of s).

Register-level description:

XSTART=x1 //fixed point number//

YSTART=y1

XEND=x2

YEND=y2

aa_table=AWEIGHT0,1=function(s,e1) // this table is calculated for each slope and is indexed by s //

DRAWMODE: OPCODE=A_line

BLEND=enabled

SFACTOR=BF_SA // SFACTOR=alpha//

DFACTOR=BF_MSA // DFACTOR=1-alpha//

Context-switched registers:

XSTART=x_current

YSTART=y_current

XEND=x2

YEND=y2

BRESE1=e1

BRESD=d

BRESS1=s

BRESS2=sdx //sdx=s*dx must be context switched//

BRESOCTINC1=octant,incr1

BRESINC2=incr2

AWEIGHT0,AWEIGHT1=aa_table

3.6.2.2 A_Edge_Top(x1,y1,x2,y2,e1,aa_table,SKIPFIRST,SKIPLAST,ENDPTFILTER)

fixed : x1,y1,x2,y2,e1

array : aa_table

THIS PRIMITIVE IS REMOVED FROM REX3 INSTRUCTION SET. THE REASON FOR NOT REMOVING IT FROMTHE SPEC IS TO ALLOW GL CODERS TO UNDERSTAND WHAT IS THAT THEY NEED TO DO IN ORDER TO COMPUTE THE MASKS (ZPATTERN, LSPATTERN) USED FOR 3D ANTIALIASED LINES. This is an anti-aliasing polygon edge with fractional endpoints.It differs from the antialiased Bresenham line because only one pixel is drawn at each iteration (the pixel external to the polygon).For clockwise polygons A_Edge_Top is invoked by the CPU for edges located in octants 1,3,5,7(for even octants the CPU must invoke A_Edge_Bottom).The reason for this is that in octants 1,3,5,7 it is the top pixel that lies outside the polygon whereas in octants 2,4,6,8 it is the bottom pixel that lies on the outside.For counterclockwise polygons the convention is reversed: CPU must invoke A_Edge_Top for edges in octants 2,4,6,8 and A_edge_Bottom in octants 1,3,5,7.The overhead for computing the octant is nill since the CPU must do it anyways in order to calculate the z-mask.It has INFINITE precision in terms of pixel positioning (exactly like I_LINE) since it doesn't rely on a DDA algorithm in terms of position determination.Since GL has a very precise notion of T-mesh edge it is possible to use this primitive to antialias only the contour of the mesh without touching the inner edges.Only the pixels above the infinitely precise line are being rendered ("above" is viewed as looking down the axis of maximum motion).The AWEIGHT table needs to be reloaded for every change in the line slope e1.Two types of antialiasing edges (top and bottom) have been invented in order to facilitate polygon antialiasing.If a top and bottom edges are drawn between the same pair of points (x1,y1) and (x2,y2) an antialiased line will result. The performance is limited by :

-the time required for the CPU to compute the slope e1

-the time for passing the arguments from the CPU to REX3 over GIO bus

-the time for generating the setup : d=3dy-2dx+2(dx*y_fract-dy*x_fract) , s=y_fract+e1*(.5-x_fract)-.5

s*dx=s*2dx=dy-dx+2(dx*y_fract-dy*x_fract)

-REX3 time for iterating one new coordinate

-REX3 time for drawing the fractional coverage endpoints

The endpoints may not be drawn in order to simplify the implementation and in order to generate the impression of sharp vertices.

Register-level description:

XSTART=x1 //fixed point number//

YSTART=y1

XEND=x2

YEND=y2

AWEIGHT0,AWEIGHT1=aa_table(s)

DRAWMODE: OPCODE=AA_Edge_Top

Context-switched registers:

XSTART=x_current

YSTART=y_current

XEND=x2

YEND=y2

BRESE1=e1

BRESD=d

BRESS1=s

BRESS2=sdx

BRESOCTINC1=octant,incr1

BRESINC2=incr2

AWEIGHT0,AWEIGHT1=aa_table

3.6.2.3 A_Edge_Bottom(x1,y1,x2,y2,e1,aa_table,SKIPFIRST,SKIPLAST,ENDPTFILTER)

fixed : x1,y1,x2,y2,e1

array : aa_table

THIS PRIMITIVE IS REMOVED FROM REX3 INSTRUCTION SET. THE REASON FOR NOT REMOVING IT FROMTHE SPEC IS TO ALLOW GL CODERS TO UNDERSTAND WHAT IS THAT THEY NEED TO DO IN ORDER TO COMPUTE THE MASKS (ZPATTERN, LSPATTERN) USED FOR 3D ANTIALIASED LINES. This is an anti-aliasing polygon edge with fractional endpoints.It differs from the antialiased Bresenham line because only one pixel is drawn at each iteration (the pixel external to the polygon).It has INFINITE precision in terms of pixel positioning (exactly like I_LINE) since it doesn't rely on a DDA algorithm in terms of position determination.Since GL has a very precise notion of T-mesh edge it is possible to use this primitive to antialias only the contour of the mesh without touching the inner edges.Only the pixels below the infinitely precise line are being rendered ("below" is viewed as looking down the axis of maximum motion).

The performance is limited by :

-the time required for the CPU to compute the slope e1

-the time for passing the arguments from the CPU to REX3 over GIO bus

-the time for generating the setup : d=3dy-2dx+2(dx*y_fract-dy*x_fract) , s=y_fract+e1*(.5-x_fract)-.5, s*dx=s*2dx=dy-dx+2(dx*y_fract-dy*x_fract)

-REX3 time for iterating one new coordinate

-REX3 time for drawing the fractional coverage endpoints

The endpoints may not be drawn in order to simplify the implementation and in order to generate the impression of sharp vertices.

Register-level description:

XSTART=x1 //fixed point number//

YSTART=y1

XEND=x2

YEND=y2

AWEIGHT0,AWEIGHT1=aa_table(s)

DRAWMODE: OPCODE=AA_Edge_Bottom

Context-switched registers:

XSTART=x_current

YSTART=y_current

XEND=x2

YEND=y2

BRESE1=e1

BRESD=d

BRESS1=s

BRESS2=sdx

BRESOCTINC1=octant,incr1

BRESINC2=incr2

AWEIGHT0,AWEIGHT1=aa_table

Code for X- line with integer endpoints

```
Procedure X_line(x1,y1,x2,y2,SKIPLAST,SKIPFIRST)
integer: x1,y1,x2,y2
Begin
//Compute the octant-independent values//
    x=x1, y=y1
    dx=ABS(x1-x2), dy=ABS(y1-y2)
   Coverage=1
If SKIPFIRST=FALSE Then Write_Pixel(x,y,Coverage)    //Starting pixel has coverage=1//
Case Octant of (x2-x1,y2-y1,dx-dy) :

1:    d=2dy-dx , incr1=2dy , incr2=2(dy-dx), Loop=dx  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=1
2:    d=2dx-dy , incr1=2dx , incr2=2(dx-dy), Loop=dy  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=1,incry2=1
3:    d=2dx-dy , incr1=2dx , incr2=2(dx-dy), Loop=dy  //compute the octant-dependent values//
       incrx1=0,incrx2=-1,incry1=1,incry2=1
4:    d=2dy-dx , incr1=2dy , incr2=2(dy-dx), Loop=dx  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=1
5:    d=2dy-dx , incr1=2dy , incr2=2(dy-dx), Loop=dx  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=-1
6:    d=2dx-dy , incr1=2dx , incr2=2(dx-dy), Loop=dy  //compute the octant-dependent values//
       incrx1=0,incrx2=-1,incry1=-1,incry2=-1
7:    d=2dx-dy , incr1=2dx , incr2=2(dx-dy), Loop=dy  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=-1,incry2=-1
8:    d=2dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=-1
For i=1 to Loop-1  Do
Begin
  If d<0 Then          // s<t , execute a horizontal step//
  Begin
    x=x+incrx1          //advance to next pixel//
    y=y+incry1
    d=d+incr1           //compute new values for d and s//
  End Else//s>t>0, execute a 45 degree step//
```

```
 Begin//45 degree move//
  x=x+incrx2
  y=y+incry2
  d=d+incr2
 End
 Write_Pixel(x,y,Coverage)
End
If  SKIPLAST=FALSE Then  Write_Pixel(x2,y2,Coverage)
End
```

Code for aliased line with fractional endpoints

```
Procedure GL_Bresenham(x1,y1,x2,y2,SKIPLAST,SKIPFIRST)
fixed : x1,y1,x2,y2     //x=x_int.x_fract where x_fract is 4  bits of precission//
fixed : dx,dy
integer: x10,y10,x20,y20,dx_i,dy_i
//Compute the octant-independent values//
x10=int(x1) , y10=int(y1)//REX3 computes the flixed->int and the d term//
x20=int(x2) , y20=int(y2)
x=x10, y=y10
dx=ABS(x1-x2), dy=ABS(y1-y2)
dx_i=ABS(x10-x20)-1, dy_i=ABS(y10-y20)-1
Case Octant  of (x2-x1,y2-y1,dx-dy) :
1:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
     incrx1=1,incrx2=1,incry1=0,incry2=1
2:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
     incrx1=0,incrx2=1,incry1=1,incry2=1
     temp=x1_fract            //swap x and y//
     x1_fract=y1_fract
     y1_fract=temp
     temp=dx
     dx=dy
     dy=temp
3:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
     incrx1=0,incrx2=-1,incry1=1,incry2=1
     temp=1-x1_fract  //use 1-x_fract left of y-axis.//
     x1_fract=y1_fract
     y1_fract=temp
     temp=dx
     dx=dy
     dy=temp
4:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
     incrx1=-1,incrx2=-1,incry1=0,incry2=1
     x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
5:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
     incrx1=-1,incrx2=-1,incry1=0,incry2=-1
```

```
     x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
     y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
6:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
     incrx1=0,incrx2=-1,incry1=-1,incry2=-1
     temp=1-x1_fract
     x1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
     y1_fract=temp
     temp=dx
     dx=dy
     dy=temp
7:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
     incrx1=0,incrx2=1,incry1=-1,incry2=-1
     temp=1-y1_fract//use 1-y_fract below  of x-axis//
     y1_fract=x1_fract
     x1_fract=temp
     temp=dx
     dx=dy
     dy=temp
8:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
     incrx1=1,incrx2=1,incry1=0,incry2=-1
     y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
d=d+2(dx*y1_fract-dy*x1_fract)          //adjust d due to fractional endpoints//
E=d-2dx                                 //variable used for adjusting the start point up one pixel//
If E>0 Then
Begin
  d=E
  x=x+incrx2*~x_major
  y=y+incry2*x_major
End
Coverage=1                        // or we can use the CPU-calculated coverage //
//* This section has been removed on 11/5/92 as a result of a discussion with BobS
If SKIPFIRST=FALSE Then
Begin
    Write_Pixel(x,y,Coverage)    //Starting pixel has coverage=1, can be drawn conditionally//
End
For i=1 to Loop-1  Do
```

```
Begin
 If d<0 Then          // s<t , execute a horizontal step//
 Begin
  x=x+incrx1          //advance to next pixel//
  y=y+incry1
  d=d+incr1           //compute new values for d and s//
 End Else//s>t>0, execute a 45 degree step//
 Begin//45 degree move//
  x=x+incrx2
  y=y+incry2
  d=d+incr2
 End
Write_Pixel(x,y,Coverage)
End
If SKIPLAST=FALSE Then //Draw the last pixel conditionally//
Begin
 If d<0 Then          // s<t , execute a horizontal step//
 Begin
  x=x+incrx1          //advance to next pixel//
  y=y+incry1
  d=d+incr1           //compute new values for d and s//
 End Else//s>t>0, execute a 45 degree step//
 Begin//45 degree move//
  x=x+incrx2
  y=y+incry2
  d=d+incr2
 End
Write_Pixel(x,y,Coverage)
End
End
```

Code for antialiased line with fractional endpoints and angle compensation, no endpoint filtering

```
Procedure Write_Pixel(x,y,alpha)          // 0=< alpha<=1  due to looking it up in aa_table //
global variable : new_color               //new_color is the current drawing color//
Begin
      Read_Framebuffer(x,y,bckg_color)  //read the background color at location (x,y)//
      color=alpha*new_color + (1-alpha)*bckg_color// alpha represents pixel coverage//
      Write_Framebuffer(x,y,color)        //write back the resultant of blending to location (x,y)//
  End



Procedure GL_AA_Bresenham(x1,y1,x2,y2,e1,c1)
fixed  : x1,y1,x2,y2,e1    //CPU computes the octant//
aray : aa_table0 (s,e1) ,aa_table1(1-s,e1)    // This array is a function of slope and is indexed with s_frac //
integer : Octant,x10,y10,x20,y20
integer : x_major//x_major=1 in octants 1,4,5,8 //
//e1=dy/dx for x-major .  e1=dx/dy for y-major where dx=ABS(x1-x2) and dy=ABS(y1-y2)//
//Compute the octant-independent values//
e2=e1-1.0
x10=int(x1) , y10=int(y1)//REX3 computes the fixed->int and the d term//
x20=int(x2) , y20=int(y2)
x=x10, y=y10
dx=ABS(x1-x2), dy=ABS(y1-y2)
dx_i=ABS(x10-x20)-1, dy_i=ABS(y10-y20)-1
Case Octant of (x2-x1,y2-y1,dx-dy) :

1:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=1 ,x_major=1
2:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=1,incry2=1 ,x_major=0
       temp=x1_fract
       x1_fract=y1_fract
       y1_fract=temp
       temp=x2_fract
       x2_fract=y2_fract
```

```
      y2_fract=temp
      temp=dx
      dx=dy
      dy=temp
3:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=1,incry2=1 ,x_major=1
      temp=1-x1_fract/use 1-x_fract left of y-axis//
      x1_fract=y1_fract
      y1_fract=temp
      temp=1-x2_fract/use 1-x_fract left of y-axis//
      x2_fract=y2_fract
      y2_fract=temp
      temp=dx
      dx=dy
      dy=temp
4:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=-1,incrx2=-1,incry1=0,incry2=1 ,x_major=0
      x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
      x2_fract=1-x2_fract
5:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=-1,incrx2=-1,incry1=0,incry2=-1 ,x_major=1
      x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
      y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
      x2_fract=1-x2_fract//use 1-x_fract left of y-axis//
      y2_fract=1-y2_fract//use 1-y_fract below  of x-axis//
6:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=-1,incry2=-1 ,x_major=0
      temp=1-x1_fract
      x1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
      y1_fract=temp
      temp=1-x2_fract
      x2_fract=1-y2_fract//use 1-y_fract below  of x-axis//
      y2_fract=temp
      temp=dx
      dx=dy
      dy=temp
```

```
7:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=-1,incry2=-1 ,x_major=0
       temp=1-y1_fract//use 1-y_fract below  of x-axis//
       y1_fract=x1_fract
       x1_fract=temp
       temp=1-y2_fract//use 1-y_fract below  of x-axis//
       y2_fract=x2_fract
       x2_fract=temp
       temp=dx
       dx=dy
       dy=temp
8:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=-1 ,x_major=1
       y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
       y2_fract=1-y2_fract

 s=y1_frac-0.5+e1(0.5-x1_frac)       //s for the first pixel//
 sdx=2[(y1_frac-0.5)dx+(0.5-x1_frac)dy]=dy-dx+2(dx*y1_fract-dy*x1_fract)
                                                  // sdx=s*2dx is an infinitely precise number //
If s<0 Then     // The correct , positive s, is in this case s=y1_frac+0.5+e1(0.5-x1_frac)  = s+1 //
Begin
       s=s+1
       sdx=sdx+2*dx   //when s=s+1 sdx=sdx+2*dx //
End
d=d+2(dx*y1_fract-dy*x1_fract)    //adjust d due to fractional endpoints,this is d for second pixel//
       // s=y1_fract+e1*(1.5-x1_fract)-0.5 this would have been s for second pixel , s=s+e1 //
E=d-2dx
If E>0 Then
Begin
  d=E
End

If SKIPFIRST=TRUE Then
Begin
       If sdx>0 Then     //Compute the coverage for the starting pixel//
       Begin      // THE BOLD CODE MAY BE  EXECUTED ON THE HOST IF SKIPFIRST=TRUE//
```

```
                // remember that for ymajor lines y1 and x1 have been swapped//
                Coverage_T=(y1_fr-1+c1/2)(1-0)+.5*e1(1-0)**2    //consider x1_fr=0//
                Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
                Coverage_S=(1-0)*c1-Coverage_T   //S is below the line and has the larger coverage//
                Write_Pixel(x,y,Coverage_S)
     End Else   //sdx<0//
     Begin
                Coverage_T=(y1_fr+c1/2)(1-0)+.5*e1(1-0)**2//T has the  larger coverage//
                Write_Pixel(x,y,Coverage_T)
                Coverage_S=(1-0)*c1-Coverage_T     //S is below the line and has the larger coverage//
                Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
     End
End  Else
Begin
     If sdx>0 Then     //Compute the coverage for the starting pixel//
     Begin      // THIS CODE  EXECUTED BY REX3 BECAUSE SKIPFIRST=FALSE//
          Coverage_T=aa_table(s_frac)    //T has the smaller coverage//
          Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
          Coverage_S=aa_table(~s_frac)   //S is below the line and has the larger coverage//
          Write_Pixel(x,y,Coverage_S)
     End Else  // sdx<0 //
     Begin
          Coverage_T=aa_table(~s_frac) //T has the  larger coverage//
          Write_Pixel(x,y,Coverage_T)
          Coverage_S=aa_table(s_frac)     //S is below the line and has the larger coverage//
          Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
     End
End // SKIPFIRST //
For i=1 to Loop-1  Do
Begin
 If d<0 Then         // s<t , execute a horizontal step//
 Begin
   x=x+incrx1          //advance to next pixel//
   y=y+incry1
   d=d+incr1           //compute new values for d and s//
   s=s+e1
```

```
    sdx=sdx+2dy=sdx+incr1         // s*dx=s*dx+e1*dx   i.e   sdx=sdx+dy //
     End Else// d>0 results into s>t>0, execute a 45 degree step//
  Begin//45 degree move//
    x=x+incrx2
    y=y+incry2
    d=d+incr2            //compute new values for d and s//
    s=s+e2               // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2   //s*dx=s*dx+e2*dx=s*dx+(dy/dx-1)*dx=s*dx+dy-dx   i.e. sdx=sdx+dy-dx //
   End
   If sdx>0 Then
   Begin
        Coverage_T=aa_table(s_frac)         // s_frac=Fraction(s) //
        Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
        Coverage_S=aa_table(~s_frac)      //~s_frac=0.f-s_frac //
        Write_Pixel(x,y,Coverage_S)
   End Else
   Begin
        Coverage_T=aa_table(~s_frac)
        Write_Pixel(x,y,Coverage_T)
        Coverage_S=aa_table(s_frac)
        Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
    End
End
```

If SKIPLAST=TRUE Then //THIS CODE MAY BE EXECUTED ON THE HOSTIF SKIPLAST=TRUE//

Begin

```
       If sdx>0 Then      //Compute the coverage for the ending pixel//
      Begin               //Correct the endpoint(s) if start point <> end point//
         Coverage_S=(1+c1/2-y2_fr)*1+.5*e1*1r**2//consider x2_fr=1//
         Write_Pixel(x,y,Coverage_S)
         Coverage_T=c1*1-Coverage_S//T is above  the line and has the smaller coverage//
         Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
     End Else
     Begin
         Coverage_S=(c1/2-y2fr)*1+.5*e1*1**2    //S is below  and has the smaller coverage//
         Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
         Coverage_T=c1*1-Coverage_S//T is above  the line and has the larger coverage//
```

```
        Write_Pixel(x,y,Coverage_T)
     End
End Else
Begin   //For SKIPLAST=FALSE REX3 fills the last pixel //
 If d<0 Then        // s<t , execute a horizontal step//
 Begin
   x=x+incrx1          //advance to next pixel//
   y=y+incry1
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else// d>0 results into s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   s=s+e2   // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2
   End
   If sdx>0 Then
   Begin
      Coverage_T=aa_table(s_frac)
      Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
      Coverage_S=aa_table(~s_frac)
      Write_Pixel(x,y,Coverage_S)
   End Else
   Begin
      Coverage_T=aa_table(~s_frac)
      Write_Pixel(x,y,Coverage_T)
      Coverage_S=aa_table(s_frac)
      Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
   End
 End
End //SKIPLAST//
End
```

Code for antialiased line with fractional endpoints and angle compensation, with endpoint filtering

```
Procedure Write_Pixel(x,y,alpha)          // 0=< alpha<=1  due to looking it up in aa_table //
global variable : new_color               //new_color is the current drawing color//
Begin
    Read_Framebuffer(x,y,bckg_color)  //read the background color at location (x,y)//
    color=alpha*new_color + (1-alpha)*bckg_color// alpha represents pixel coverage//
    Write_Framebuffer(x,y,color)       //write back the resultant of blending to location (x,y)//
 End

Procedure GL_AAE_Bresenham(x1,y1,x2,y2,e1,c1)
fixed  : x1,y1,x2,y2,e1    //CPU computes the octant//
aray : aa_table0 (s,e1) ,aa_table1(1-s,e1)    // This array is a function of slope and is indexed with s //
integer : Octant,x10,y10,x20,y20
integer : x_major//x_major=1 in octants 1,4,5,8 //
//e1=dy/dx for x-major .  e1=dx/dy for y-major where dx=ABS(x1-x2) and dy=ABS(y1-y2)//
//Compute the octant-independent values//
e2=e1-1.0
x10=int(x1) , y10=int(y1)//REX3 computes the fixed->int and the d term//
x20=int(x2) , y20=int(y2)
x=x10, y=y10
dx=ABS(x1-x2), dy=ABS(y1-y2)
dx_i=ABS(x10-x20)-1, dy_i=ABS(y10-y20)-1
Case Octant of (x2-x1,y2-y1,dx-dy) :

1:   d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
     incrx1=1,incrx2=1,incry1=0,incry2=1 ,x_major=1
2:   d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
     incrx1=0,incrx2=1,incry1=1,incry2=1 ,x_major=0
     temp=x1_fract
     x1_fract=y1_fract
     y1_fract=temp
     temp=x2_fract
     x2_fract=y2_fract
     y2_fract=temp
     temp=dx
```

```
      dx=dy
      dy=temp
3:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=1,incry2=1 ,x_major=1
      temp=1-x1_fract/use 1-x_fract left of y-axis//
      x1_fract=y1_fract
      y1_fract=temp
      temp=1-x2_fract/use 1-x_fract left of y-axis//
      x2_fract=y2_fract
      y2_fract=temp
      temp=dx
      dx=dy
      dy=temp
4:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=-1,incrx2=-1,incry1=0,incry2=1 ,x_major=0
      x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
      x2_fract=1-x2_fract
5:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=-1,incrx2=-1,incry1=0,incry2=-1 ,x_major=1
      x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
      y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
      x2_fract=1-x2_fract//use 1-x_fract left of y-axis//
      y2_fract=1-y2_fract//use 1-y_fract below  of x-axis//
6:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=-1,incry2=-1 ,x_major=0
      temp=1-x1_fract
      x1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
      y1_fract=temp
      temp=1-x2_fract
      x2_fract=1-y2_fract//use 1-y_fract below  of x-axis//
      y2_fract=temp
      temp=dx
      dx=dy
      dy=temp
7:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=1,incry1=-1,incry2=-1 ,x_major=0
```

```
        temp=1-y1_fract//use 1-y_fract below  of x-axis//
        y1_fract=x1_fract
        x1_fract=temp
        temp=1-y2_fract//use 1-y_fract below  of x-axis//
        y2_fract=x2_fract
        x2_fract=temp
        temp=dx
        dx=dy
        dy=temp
8:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
        incrx1=1,incrx2=1,incry1=0,incry2=-1 ,x_major=1
        y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
        y2_fract=1-y2_fract

        s=y1_frac-0.5+e1(0.5-x1_frac)        //s for the first pixel//
        sdx=2[(y1_frac-0.5)dx+(0.5-x1_frac)dy]=dy-dx+2(dx*y1_fract-dy*x1_fract)    // sdx=s*2dx is an
                                                                        infinitely precise number //
If s<0 Then      // The correct , positive s, is in this case s=y1_frac+0.5+e1(0.5-x1_frac)  = s+1 //
Begin
        s=s+1
        sdx=sdx+2*dx   //when s=s+1 sdx=sdx+2*dx //
End
        d=d+2(dx*y1_fract-dy*x1_fract) //adjust d due to fractional endpoints,this is d for second pixel//
        // s=y1_fract+e1*(1.5-x1_fract)-0.5 this would have been s for second pixel , s=s+e1 //
E=d-2dx
If E>0 Then
Begin
   d=E
End

If SKIPFIRST=TRUE Then
Begin
        If sdx>0 Then     //Compute the coverage for the starting pixel//
        Begin      // THE BOLD CODE MAY BE  EXECUTED ON THE HOST IF SKIPFIRST=TRUE//
          Coverage_T=(y1_fr-1+c1/2)(1-x1_fr)+.5*e1(1-x1_fr)**2    //T has the smaller coverage//
          Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
```

```
        Coverage_S=(1-x1_fr)*c1-Coverage_T   //S is below the line and has the larger coverage//
        Write_Pixel(x,y,Coverage_S)
      End  Else    // sdx<0 //
      Begin
         Coverage_T=(y1_fr+c1/2)(1-x1_fr)+.5*e1(1-x1_fr)**2//T has the  larger coverage//
         Write_Pixel(x,y,Coverage_T)
          Coverage_S=(1-x1_fr)*c1-Coverage_T     //S is below the line and has the larger coverage//
          Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
      End
End  Else
Begin
      If sdx>0 Then      //Compute the coverage for the starting pixel//
      Begin      // THIS CODE  EXECUTED BY REX3 BECAUSE SKIPFIRST=FALSE//
          Coverage_T=aa_table(s_frac*(1-x1_fr))    //T he coverages are inversely proportional with x1_fr//
          Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
          Coverage_S=aa_table(~s_frac*(1-x1_fr))   //S is below the line and has the larger coverage//
          Write_Pixel(x,y,Coverage_S)
      End Else  // sdx<0//
          Coverage_T=aa_table(~s_frac*(1-x1_fr)) //T has the  larger coverage//
          Write_Pixel(x,y,Coverage_T)
          Coverage_S=aa_table(s_frac*(1-x1_fr))     //S is below the line and has the larger coverage//
          Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
      End
End // SKIPFIRST //
For i=1 to Loop-1  Do
Begin
  If d<0 Then         // s<t , execute a horizontal step//
  Begin
    x=x+incrx1          //advance to next pixel//
    y=y+incry1
    d=d+incr1           //compute new values for d and s//
    s=s+e1
    sdx=sdx+2dy=sdx+incr1
     End Else// d>0 results into s>t>0, execute a 45 degree step//
  Begin//45 degree move//
    x=x+incrx2
```

```
   y=y+incry2
   d=d+incr2          //compute new values for d and s//
   s=s+e2             // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2
  End
  If sdx>0 Then
  Begin
       Coverage_T=aa_table(s_frac)        // s_frac=Fraction(s) //
       Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
       Coverage_S=aa_table(~s_frac)      //~s_frac=1-s_frac //
       Write_Pixel(x,y,Coverage_S)
  End Else
  Begin
       Coverage_T=aa_table(~s_frac)
       Write_Pixel(x,y,Coverage_T)
       Coverage_S=aa_table(s_frac)
       Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
   End
End
If  SKIPLAST=TRUE Then  //THIS CODE MAY BE  EXECUTED ON THE HOSTIF SKIPLAST=TRUE//
Begin
      If sdx>0 Then    //Compute the coverage for the ending pixel//
       Begin//Correct the endpoint(s) if start point <> end point//
        Coverage_S=(1+c1/2-y2_fr)x2_fr+.5*e1*x2_fr**2//s is below  and has the larger coverage//
        Write_Pixel(x,y,Coverage_S)
        Coverage_T=c1*x2fr-Coverage_S//T is above  the line and has the smaller coverage//
        Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
   End Else     //  sdx<0 //
   Begin
        Coverage_S=(c1/2-y2fr)x2fr+.5*e1*x2fr**2    //S is below  and has the smaller coverage//
        Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
        Coverage_T=c1*x2_fr-Coverage_S//T is above  the line and has the larger coverage//
         Write_Pixel(x,y,Coverage_T)
       End
End Else
Begin   //For SKIPLAST=FALSE REX3 fills the last pixel //
```

```
 If d<0 Then          // s<t , execute a horizontal step//
 Begin
   x=x+incrx1          //advance to next pixel//
   y=y+incry1
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else// d>0 results into s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   s=s+e2   // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2
   End
   If sdx>0 Then   //The coverages are directly proportional with x2_fr//
   Begin
       Coverage_T=aa_table(s_frac*x2_fr)
       Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
       Coverage_S=aa_table(~s_frac*x2_fr)
       Write_Pixel(x,y,Coverage_S)
   End Else
   Begin
       Coverage_T=aa_table(~s_frac*x2_fr)
       Write_Pixel(x,y,Coverage_T)
       Coverage_S=aa_table(s_frac*x2_fr)
       Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
   End
 End
End //SKIPLAST//
End
```

Code for polygon antialiasing top edge with fractional endpoints

```
Procedure Write_Pixel(x,y,alpha)// 0=< alpha<=1 //
Begin
global variable : new_color                    //new_color is the current drawing color//
Read_Framebuffer(x,y,bckg_color)   //read the background color at location (x,y)//
color=alpha*new_color + (1-alpha)*bckg_color  // alpha represents pixel coverage//
Write_Framebuffer(x,y,color)          //write back the resultant of blending to location (x,y)//
End

Procedure GL_ AA_Bresenham_Edge(x1,y1,x2,y2,e1)
fixed : x1,y1,x2,y2,e1    //e1=dy/dx for x-major .  e1=dx/dy for y-mjor //
array : aa_table0(s) // for antialiasing edges we may not need angle compensation //
integer : x_major//x_major=1  in octants 1,4,5,8 //
//Compute the octant-independent values//
e2=e1-1
x10=int(x1) , y10=int(y1)//REX3 computes the fixed->int and the d term//
x20=int(x2) , y20=int(y2)
x=x1, y=y1
dx=ABS(x1-x2), dy=ABS(y1-y2)
dx_i=ABS(x10-x20)-1, dy_i=ABS(y10-y20)-1

Case Octant of (x2-x1,y2-y1,dx-dy:

1:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=1,incrx2=1,incry1=0,incry2=1 ,x_major=1
2:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=1,incry1=1,incry2=1 ,x_major=0
      temp=x1_fract
      x1_fract=y1_fract
      y1_fract=temp
      temp=dx
      dx=dy
      dy=temp
3:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=1,incry2=1 ,x_major=0
```

```
      temp=x1_fract//use 1-x_fract left of y-axis//
      x1_fract=y1_fract
      y1_fract=temp
      temp=dx
      dx=dy
      dy=temp
4:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=1 ,x_major=1
       x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
5:    d=2dy-dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=-1
       x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
       y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
6:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=-1,incry1=-1,incry2=-1 ,x_major=0
       temp=1-x1_fract
       x1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
       y1_fract=temp
       temp=dx
       dx=dy
       dy=temp
7:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=-1,incry2=-1 ,x_major=0
      temp=1-y1_fract//use 1-y_fract below  of x-axis//
      y1_fract=x1_fract
      x1_fract=temp
      temp=dx
      dx=dy
      dy=temp
8:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=-1 ,x_major=1
       y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
```

d=d+2**(dx*y1_fract-dy*x1_fract)**   //adjust d due to fractional endpoints//

s=y1_fract-0.5+e1*(.5-x1_fract)

sdx=2[(y1_fract-0.5)dx+(.5-x1_fract)dy]=dy-dx+**2(dx*y1_fract-dy*x1_fract)**

```
If s<0 Then      // The correct , positive s, is in this case s=y1_frac+0.5+e1(0.5-x1_frac)  = s+1 //
Begin
        s=s+1
        sdx=sdx+2*dx   //when s=s+1 sdx=sdx+2*dx //
End
E=d-2dx
If E>0 Then
Begin
  d=E
End

If SKIPFIRST=TRUE Then
Begin
        If sdx>0 Then     //Compute the coverage for the starting pixel//
        Begin      // THE BOLD CODE MAY BE  EXECUTED ON THE HOST IF SKIPFIRST=TRUE//
           Coverage_T=(y1_fr-1+c1/2)(1-x1_fr)+.5*e1(1-x1_fr)**2    //T has the smaller coverage//
           Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
         End  Else    // sdx<0 //
         Begin
             Coverage_T=(y1_fr+c1/2)(1-x1_fr)+.5*e1(1-x1_fr)**2//T has the  larger coverage//
             Write_Pixel(x,y,Coverage_T)
         End
End  Else
Begin
        If sdx>0 Then      //Compute the coverage for the starting pixel//
        Begin      // THIS CODE  EXECUTED BY REX3 BECAUSE SKIPFIRST=FALSE//
              Coverage_T=aa_table(s_frac*(1-x1_fr))    //T he coverages are inversely proportional with x1_fr//
              Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
        End Else  // sdx<0//
              Coverage_T=aa_table(~s_frac*(1-x1_fr)) //T has the  larger coverage//
              Write_Pixel(x,y,Coverage_T)
        End
End // SKIPFIRST //
For i=1 to Loop-1  Do
Begin
  If d<0 Then         // s<t , execute a horizontal step//
```

```
 Begin
   x=x+incrx1          //advance to next pixel//
   y=y+incry1
   d=d+incr1           //compute new values for d and s//
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else//s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   d=d+incr2
   s=s+e2
   sdx=sdx+2(dy-dx)=sdx+incr2
 End
 If sdx>0 Then
 Begin
      Coverage_T=aa_table(s_frac)     // only the top pixel is antialiased //
      Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
 End Else
 Begin
      Coverage_T=aa_table(~s_frac)     // only the top pixel is antialiased //
      Write_Pixel(x,y,Coverage_T)
  End
 End // If//
End // For //
If  SKIPLAST=TRUE Then  //THIS CODE MAY BE  EXECUTED ON THE HOSTIF SKIPLAST=TRUE//
Begin
     If sdx>0 Then    //Compute the coverage for the ending pixel//
      Begin//Correct the endpoint(s) if start point <> end point//
      Coverage_T=c1*x2fr-Coverage_S//T is above  the line and has the smaller coverage//
      Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
    End Else     //  sdx<0 //
    Begin
       Coverage_T=c1*x2_fr-Coverage_S//T is above  the line and has the larger coverage//
       Write_Pixel(x,y,Coverage_T)
      End
```

```
End Else
Begin   //For SKIPLAST=FALSE REX3 fills the last pixel //
 If d<0 Then          // s<t , execute a horizontal step//
 Begin
   x=x+incrx1            //advance to next pixel//
   y=y+incry1
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else// d>0 results into s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   s=s+e2   // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2
   End
   If sdx>0 Then   //The coverages are directly proportional with x2_fr//
   Begin
       Coverage_T=aa_table(s_frac*x2_fr)
       Write_Pixel(x+incrx2*~x_major,y+incry2*x_major,Coverage_T)
   End Else
   Begin
       Coverage_T=aa_table(~s_frac*x2_fr)
       Write_Pixel(x,y,Coverage_T)
   End
 End
End //SKIPLAST//
End
```

Code for polygon antialiasing bottom edge with fractional endpoints

```
Procedure Write_Pixel(x,y,alpha)// 0=< alpha<=1 //
Begin
global variable : new_color                    //new_color is the current drawing color//
Read_Framebuffer(x,y,bckg_color)   //read the background color at location (x,y)//
color=alpha*new_color + (1-alpha)*bckg_color  // alpha represents pixel coverage//
Write_Framebuffer(x,y,color)          //write back the resultant of blending to location (x,y)//
End

Procedure GL_ AA_Bresenham_Edge(x1,y1,x2,y2,e1)
fixed : x1,y1,x2,y2,e1    //e1=dy/dx for x-major .  e1=dx/dy for y-mjor //
array : aa_table0(s) // for antialiasing edges we may not need angle compensation //
integer : x_major//x_major=1  in octants 1,4,5,8 //
//Compute the octant-independent values//
e2=e1-1
x10=int(x1) , y10=int(y1)//REX3 computes the fixed->int and the d term//
x20=int(x2) , y20=int(y2)
x=x1, y=y1
dx=ABS(x1-x2), dy=ABS(y1-y2)
dx_i=ABS(x10-x20)-1, dy_i=ABS(y10-y20)-1

Case Octant of (x2-x1,y2-y1,dx-dy:

1:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
      incrx1=1,incrx2=1,incry1=0,incry2=1 ,x_major=1
2:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=1,incry1=1,incry2=1 ,x_major=0
      temp=x1_fract
      x1_fract=y1_fract
      y1_fract=temp
      temp=dx
      dx=dy
      dy=temp
3:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
      incrx1=0,incrx2=-1,incry1=1,incry2=1 ,x_major=0
```

```
      temp=x1_fract//use 1-x_fract left of y-axis//
      x1_fract=y1_fract
      y1_fract=temp
      temp=dx
      dx=dy
      dy=temp
4:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=1 ,x_major=1
       x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
5:    d=2dy-dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=-1,incrx2=-1,incry1=0,incry2=-1
       x1_fract=1-x1_fract//use 1-x_fract left of y-axis//
       y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
6:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=-1,incry1=-1,incry2=-1 ,x_major=0
       temp=1-x1_fract
       x1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
       y1_fract=temp
       temp=dx
       dx=dy
       dy=temp
7:    d=3dx-2dy , incr1=2dx , incr2=2(dx-dy), Loop=dy_i  //compute the octant-dependent values//
       incrx1=0,incrx2=1,incry1=-1,incry2=-1 ,x_major=0
      temp=1-y1_fract//use 1-y_fract below  of x-axis//
      y1_fract=x1_fract
      x1_fract=temp
      temp=dx
      dx=dy
      dy=temp
8:    d=3dy-2dx , incr1=2dy , incr2=2(dy-dx), Loop=dx_i  //compute the octant-dependent values//
       incrx1=1,incrx2=1,incry1=0,incry2=-1 ,x_major=1
       y1_fract=1-y1_fract//use 1-y_fract below  of x-axis//
```

d=d+2**(dx*y1_fract-dy*x1_fract)**   //adjust d due to fractional endpoints//

s=y1_fract-0.5+e1*(.5-x1_fract)

sdx=2[(y1_fract-0.5)dx+(.5-x1_fract)dy]=dy-dx+**2(dx*y1_fract-dy*x1_fract)**

```
If s<0 Then      // The correct , positive s, is in this case s=y1_frac+0.5+e1(0.5-x1_frac)  = s+1 //
Begin
        s=s+1
        sdx=sdx+2*dx   //when s=s+1 sdx=sdx+2*dx //
End
E=d-2dx
If E>0 Then
Begin
  d=E
End

If SKIPFIRST=TRUE Then
Begin
        If sdx>0 Then     //Compute the coverage for the starting pixel//
        Begin      // THE BOLD CODE MAY BE  EXECUTED ON THE HOST IF SKIPFIRST=TRUE//
          Coverage_S=(1-x1_fr)*c1-Coverage_T   //S is below the line and has the larger coverage//
          Write_Pixel(x,y,Coverage_S)
        End  Else    // sdx<0 //
        Begin
            Coverage_S=(1-x1_fr)*c1-Coverage_T     //S is below the line and has the larger coverage//
            Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
        End
End  Else
Begin
        If sdx>0 Then      //Compute the coverage for the starting pixel//
        Begin      // THIS CODE  EXECUTED BY REX3 BECAUSE SKIPFIRST=FALSE//
            Coverage_S=aa_table(~s_frac*(1-x1_fr))   //S is below the line and has the larger coverage//
            Write_Pixel(x,y,Coverage_S)
        End Else  // sdx<0//
            Coverage_S=aa_table(s_frac*(1-x1_fr))     //S is below the line and has the larger coverage//
            Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
        End
End // SKIPFIRST //
For i=1 to Loop-1  Do
Begin
  If d<0 Then          // s<t , execute a horizontal step//
```

```
 Begin
   x=x+incrx1          //advance to next pixel//
   y=y+incry1
   d=d+incr1           //compute new values for d and s//
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else//s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   d=d+incr2
   s=s+e2
   sdx=sdx+2(dy-dx)=sdx+incr2
 End
 If sdx>0 Then
 Begin
      Coverage_S=aa_table(~s_frac)     // only the bottom  pixel is antialiased //
     Write_Pixel(x,y,Coverage_S)
 End Else
 Begin
      Coverage_S=aa_table(~s_frac)     // only the bottom  pixel is antialiased //
      Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
  End
 End // If//
End // For //
If  SKIPLAST=TRUE Then  //THIS CODE MAY BE  EXECUTED ON THE HOSTIF SKIPLAST=TRUE//
Begin
     If sdx>0 Then    //Compute the coverage for the ending pixel//
      Begin//Correct the endpoint(s) if start point <> end point//
      Coverage_S=(1+c1/2-y2_fr)x2_fr+.5*e1*x2_fr**2//s is below  and has the larger coverage//
      Write_Pixel(x,y,Coverage_S)
    End Else     //  sdx<0 //
   Begin
      Coverage_S=(c1/2-y2fr)x2fr+.5*e1*x2fr**2    //S is below  and has the smaller coverage//
      Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
      End
```

```
End Else
Begin   //For SKIPLAST=FALSE REX3 fills the last pixel //
 If d<0 Then          // s<t , execute a horizontal step//
 Begin
   x=x+incrx1           //advance to next pixel//
   y=y+incry1
   s=s+e1
   sdx=sdx+2dy=sdx+incr1
 End Else// d>0 results into s>t>0, execute a 45 degree step//
 Begin//45 degree move//
   x=x+incrx2
   y=y+incry2
   s=s+e2   // this brings s back into the interval [-1,1] //
   sdx=sdx+2(dy-dx)=sdx+incr2
   End
   If sdx>0 Then   //The coverages are directly proportional with x2_fr//
   Begin
       Coverage_S=aa_table(~s_frac*x2_fr)
       Write_Pixel(x,y,Coverage_S)
   End Else
   Begin
       Coverage_S=aa_table(s_frac*x2_fr)
       Write_Pixel(x-incrx2*~x_major,y-incry2*x_major,Coverage_S)
   End
 End
End //SKIPLAST//
End
```

3.7 Double Buffering

Double-buffered drawing is supported for pixels. Allowed formats are described in Section 3.9, Framebuffer Formats.

Double-buffering for writes is specified by the pixel depth and format in DRAWMODE1, and implicitly via the WRITEMASK, which must be set to match the Table in Section 3.9. Writes to both buffers use replicated source data.

Double-buffered reads are explicitly specified by DRAWMODE1 bit DBLSRC. Buffer0 (or BufferA) is the lower significant pixel within the framebuffer data value: see Section 3.9 for details. Pixel format is again specified as above, via DRAWMODE1. This handles cases of R-M-W drawing, and host/DMA reads of double-buffered framebuffer.

Double buffering brings about a peculiarity with LOGICOP function: while the LO_DST normally can be viewed as a NOOP (write result is simply the original, destination value), the case of double buffer source not equal to double buffer destination actually must perform a copy from one buffer to the other. Therefore the REX3 hardware will treat LO_DST as a copy, not a NOOP.

3.8 Framebuffer Data Values

Framebuffer data includes pixel, overlay, and CID types; one is specified for each read or write operation, using the PLANES field of DRAWMODE1 register.

There are two main sources for drawn data: the DDA, and the host data register, RWHOST1,0. Data source is specified by DRAWMODE0 register COLORHOST, ALPHAHOST. For host data, COLORHOST, ALPHAHOST=1 and the data is interpreted using DRAWMODE1 as specified by fields RWPACKED, RWDOUBLE, HOSTDEPTH. The data is assumed within legal range, no clamping necessary. COLORHOST, ALPHAHOST=0 directs the graphics pipeline to make use of the DDA values; in this case, SHADE=1 specifies linear shading is performed for successive, iterated values. The bit RGBMODE specifies whether color index or RGB values are to be calculated. DDA values of R,G,B,A are clamped each iteration before sending down the pipeline. As each of these components has an additional, overflow bit at the DDA, a normalized range of [-.5 to +1.5) is handled prior to clamping. Color index DDA values can be clamped to desired range by setting the DRAWMODE0 bit ENCICLAMP.

Normally either the DDA or the host value is used, but there is an exception for blend function where both are taken: ALPHAHOST=1 with COLORHOST=0 specifies the HOSTRW1,0 alpha fields are to be used to blend the DDA R,G,B components. For more information on the Blend Functions, see Section 3.8.4.

The framebuffer pixel depth to be drawn is specified by DRAWMODE1 field DRAWDEPTH. In conjunction with the rest of the modes mentioned, framebuffer format can be controlled as shown in Section 3.9.

Other options or modes which affect pixel value include dither, round, antialias, blend, pattern, and logicop. These are covered in the following sections.

3.8.1 Patterning and Stippling

There are two 32b pattern registers in REX3: LSPATTERN and ZPATTERN. They are enabled via DRAWMODE0 bits ENLSPATTERN, ENZPATTERN. This determines whether each are used in the pixel path, and whether the pattern iterates during drawing. Each of these patterns can be specified as transparent (mask out pixels corresponding to pattern=0), or opaque (substitute a background color for pixels corresponding to pattern=0), via bits LSOPAQUE, ZOPAQUE. Opaque patterning relies the background color stored in the COLORBACK register.

The LSPATTERN is used mainly for lines by the GL, or more generally by X11. The LSMODE register contains a length specifier LSLENGTH (17-32) for pattern recirculation, and a repeat per bit specifier (1-255) LSREPEAT to describe iterations of each pattern bit. Context switching is aided by the LSRCOUNT field, which contains the iteration state of LSREPEAT counter. The LSREPEAT function is for linedraw only, and must be set to ‘1’ by host otherwise. Similarly, the LSADVLAST function is cleared for connected vectors case only, and must be set by host otherwise.

Wide lines require the line stipple pattern to be reset identically for each wide line segment; this is accomplished via the state in registers LSPATSAVE and LSMODE field LSRCNTSAVE. At the start of drawing a wide line, these registers are initialized to the same values as LSPATTERN, LSRCOUNT respectively. For all but the first line of a wide line segment, the saved versions are copied into the working registers, using command with GO “LSRESTORE”. Upon completion of the last line of a wide line segment, command with GO “LSSAVE” is issued to copy iterated state into the saved registers.

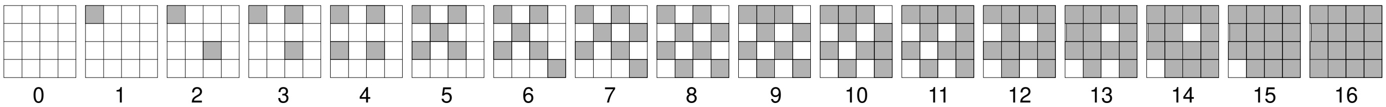
The ZPATTERN is used for patterning and as a Z write enable mask (soft Z). It is always 32b long and repeats.

When both pattern are enabled, the background color is substituted into the pixel path iff not both pattern bits are asserted (e.g., LSPATTERN & ZPATTERN bitwise false). The pixel location can be written iff {(LSOPAQUE+LSPATTERN) & (ZOPAQUE+ZPATTERN)} is bitwise true.

Z buffering of antialiased lines makes use of both patterns, with ZPATTERN used for the primary pixel mask, and the LSPATTERN used for the secondary pixel mask.

3.8.2 Dither

REX3 uses the 4 x 4 Bayer dither matrix. The seventeen intensities created by this dither matrix are illustrated below.



y \ x	00	01	10	11
00	0	8	2	10
01	12	4	14	6
10	3	11	1	9
11	15	7	13	5

4 x 4 Bayer Dither Matrix.

The least significant two bits of the *window* X and Y addresses are used to select a value from the dither matrix. The matrix value is then compared against the 4 msbs of the target color fraction. The pixel at X,Y is intensified if the desired value is *greater* than the matrix value, otherwise it is not intensified.

Because this operation would create an overall brightening of the image (and clamping at the high end), the pre-dithered pixel values are scaled prior to matrix comparison.

DIthering is enabled by setting the DRAWMODE1 register DITHER bit.

3.8.2.1 RGB Dithering

If enabled, REX3 dithers 1, 2, 3, and 4-bit stored RGB pixel components. No dithering is performed on 24-bit RGB. The following illustrates REX3 scaled dithering for 1 through 4-bit RGB components, given an 8-bit target pixel value, P[7:0]:

<u>1-bit (1-2-1):</u>

Scale P[7:0] by $^{128}/_{255}$ ($\approx$ $^{1}/_{2}$):

1. S = P[7:3] x $^{1}/_{2}$ = P[7:3] - P[7:4]
2. if (S[3:0] > DitherMatrix[x,y]) then D = S[4] + 1
   else D = S[4]

<u>2-bit (1-2-1 and 3-3-2):</u>

Scale P[7:0] by $^{192}/_{255}$ ($\approx$ $^{3}/_{4}$):

1. S = P[7:2] - P[7:2]/$_{4}$ = P[7:2] - P[7:4]
2. if (S[3:0] > DitherMatrix[x,y]) then D = S[5:4] + 1
   else D = S[5:4]

3-bit (3-3-2):

Scale P[7:0] by $^{224}/_{255}$ ($\approx {}^{7}/_{8}$):

1. S = P[7:1] - P[7:4]
2. if (S[3:0] > DitherMatrix[x,y]) then D = S[6:4] + 1
   else D = S[6:4]

4-bit (4-4-4):

Scale P[7:0] by $^{240}/_{255}$ ($\approx {}^{15}/_{16}$):

1. S = P[7:0] - P[7:4]
2. if (S[3:0] > DitherMatrix[x,y]) then D = S[7:4] + 1
   else D = S[7:4]

3.8.2.2 Color Index Dithering

No scaling is performed for CI pixels. In REX3, the CI fraction is clamped before the dither stage so that no overflow will occur due to the dither increment operation.

For antialiased CI, the integer 4 lsbs are replaced by a 4-bit AWEIGHT (intensity). REX3 then dithers by incrementing CI(4). The DDA-section muxes the original integer 4 lsbs to the CI fraction, so that dithering logic always uses the same 4-bit field for matrix comparison. (Dithering has no effect on 4-bit antialiased pixels).

The following illustrates CI dithering given a 12-bit integer and 4-bit fraction, I[11:0]).F[3:0]:

CI 4, 8 and 12-bit, non-antialiased:

if (F[3:0] > DitherMatrix[x,y]) then D = I[11:0] + 1

else D =I[3:0]

CI 8 and 12-bit, antialias enabled:

if (F[3:0] > DitherMatrix[x,y]) then D = I[11:0] + 0x10

else D =I[7:0]

3.8.3 Color rounding

GL requires the color be rounded to the nearest color. The dithering algorithm takes care of color rounding when dithering is enabled. When dithering is turned off, the intensity P of the color is rounded to the nearest color according to the algorithm described as follows.

Non-antialiased Color index: Increment the color if the MSB of the color fraction is 1.
Antialiased Color index : Increment the color by 16 if bit 3 of the iterated color integer is 1.
Antialiased 4 bit color index is not rounded.

RGB 1 bit : The final color D[0] = P[7] the MSB bit of the color .

RGB 2 bits : S[5:0] = P[7:2] - P[7:2]/4 = P[7:2] - P[7:4]
The final color D[1:0] = S[5:4] + S[3]

RGB 3 bits : S[6:0] = P[7:1] - P[7:1]/8 = P[7:1] - P[7:4]
The final color D[2:0] = S[6:4] + S[3]

RGB 4 bits : S[7:0] = P[7:0] - P[7:0]/16 = P[7:0] - P[7:4]
The final color D[3:0] = S[7:4] + S[3]

RGB 8 bits : The final color D[7:0] = P[7:0] no rounding is performed.

The rounding of color is performed in the dithering block, which is before the logicop block, therefore the source color of the logicop is rounded but the destination color and the logicop result are not rounded.

3.8.4 Logic OP

The LOGICOP field of DRAWMODE1 register defines the logicop operation used to combine the pixels being iterated (source pixels) with the pixels already written (destination pixels). Logical operations can be performed on any planes. Logical operations are disabled when LOGICOP=3. The logical operation is implemented in RB2 chip.

3.8.5 Blend

In RGB mode, the system draws pixels using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (destination) or the background color register COLORBACK (if BACKBLEND in DRAWMODE1 register is set to 1). The SFACTOR and DFACTOR fields of the DRAWMODE1 register defines the source color multiplier (Fs) and destination color multiplier (Fd) used for blending. The blending function is : Cb = Cs*Fs + Cd*Fd, where Cb is blended color , Cs is source color and Cd is destination color. The normalization of the alpha and color components in source and des-

SFACTOR	Source Multiplier (Fs)
0	zero
1	one
2	normalized destination color
3	one minus normalized destination color
4	normalized source alpha
5	one minus normalized source alpha

Table 20: SFACTOR Definition

DFACTOR	Destination Multiplier (Fd)
0	zero
1	one
2	normalized source color
3	one minus normalized source color
4	normalized source alpha
5	one minus normalized source alpha

Table 21: DFACTOR Definition

tination multipliers are converted from 8-bit integers to numbers between 0 and 1 by adding the MSB to the number and dividing by 256. Thus FF becomes 1.0 and 0 remains 0.

When source multiplier is set to source alpha (SFACTOR=4), alpha component can be blended in two different ways depending on how BLENDALPHA bit in the DRAWMODE1 register is set. When BLENDALPHA is set to 0, the source multiplier for blending alpha is one instead of source alpha and destination multiplier is defined by DFACTOR. When BLENDALPHA is set to 1, alpha is blended the way defined by the SFACTOR and DFACTOR.

Blending is enabled by setting BLEND in the DRAWMODE1 register to 1. Enable blender will slow down pixel process, therefore blender should not be enabled if it is not used. Blending and logical operation are mutually exclusive.

3.9 Framebuffer Formats

Table 22: Frame Buffer Pixel Formats

BIT PLANES	PIXEL TYPE	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
24	RGB-SB 24BIT	B0	R0	G0	B1	R1	G1	B2	R2	G2	B3	R3	G3	B4	R4	G4	B5	R5	G5	B6	R6	G6	B7	R7	G7
24	RGB-DB 444+444	B4	R4	G4	B5	R5	G5	B6	R6	G6	B7	R7	G7	B4	R4	G4	B5	R5	G5	B6	R6	G6	B7	R7	G7
24	CI-SB 12BIT	-	-	-	-	-	-	-	-	-	-	-	-	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0
24	CI-DB 12+12	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0
8/24	RGB-SB 8BIT 332	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R5	G5	B6	R6	G6	B7	R7	G7
8/24	RGB-DB 8BIT 121+121	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	G6	B7	R7	G7	G6	B7	R7	G7
8/24	CI-SB 8BIT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	I7	I6	I5	I4	I3	I2	I1	I0
8/24	CI-DB 4+4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	I3	I2	I1	I0	I3	I2	I1	I0
24	RGBα - DB 3324 + 3324	α4	α5	α6	α7	R5	G5	B6	R6	G6	B7	R7	G7	α4	α5	α6	α7	R5	G5	B6	R6	G6	B7	R7	G7
24	RGBα - SB 444 8	-	-	-	-	α0	α1	α2	α3	α4	α5	α6	α7	B4	R4	G4	B5	R5	G5	B6	R6	G6	B7	R7	G7
24	CID/AUX 2BITS-CID 2BITS-PUP 8BIT AUX	A87	A86	A85	A84	A07	A06	A05	A04	A83	A82	A81	A80	A03	A02	A01	A00	P81	P80	C81	C80	P01	P00	C01	C00
8	2BITS-CID 2BITS-PUP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	P81	P80	C81	C80	P01	P00	C01	C00

NOTES: R - Red, G - Green, B - Blue, In - Color index, $C_{p,n}$ - CIDpixel, bit field, $P_{p,n}$ - PUPpixel, bit field, $A_{p,(b),n}$ - OLAYpixel, (buffer), bit field, α_n - Alpha

Programing of the Planes(2:0), Drawdepth(1:0), and RGBmode bits will allow writing the frame buffer formats shown in Table 24.

BIT PLANES	PIXEL TYPE	Planes(2:0)	Drawdepth(1:0)	RGBmode
24	RGB-SB 24BIT	001	11	1
24	RGB-DB 444+444	001	10	1
24	CI-SB 12BIT	001	10	0
24	CI-DB 12+12	001	10	0
8/24	RGB-SB 8BIT 332	001	01	1
8/24	RGB-DB 121+121	001	00	1
8/24	CI-SB 8BIT	001	01	0
8/24	CI-DB 4+4	001	00	0
24	RGBα - DB 3324 + 3324	010	01	1
24	RGBα - SB 444 8	010	10	1
8/24	CID	110	xx	0
8/24	PUP	101	xx	0
24	OLY -SB 8 Bit	100	01	0

Table 23: Frame buffer formats programmed by Planes(2:0), Drawdepth(1:0) and RGBmode

Note: For all the modes shown as double buffered, software will have to set the writemask for the appropriate buffer. REX3 allows writing to any one plane at a time. When writing to one of the Auxiliary planes (CID, PUP or OVERLAY), writemask has to be set to disable writing to the other planes, e.g. the PUP and OVERLAY planes have to masked when writing to the cid planes. The writemask would have to match the pixel formats shown in Table 23.

3.10 Framebuffer PIO and DMA

REX3 supports programmed I/O and DMA reads and writes, from and to all bitplane types. All spanmode read and write addressing must step X from left to right. Packing, unpacking, and word size are specified via the following DRAWMODE1 bits: RWPACKED, HOSTDEPTH<1:0>, RWDOUBLE. All reads and writes are made through the HOSTRW1,0 register pair; for 32b access, only the HOSTRW0 is used. All writes to framebuffer rely on COLORHOST and/or ALPHAHOST=1 to indicate HOSTRW values, not DDA, are used.

The data formats for the HOSTRW1,0 register are illustrated in the accompanying table. Each data value resides in a field of 8, 16, or 32 bits as programmed by HOSTDEPTH; the leftmost field is the first one to be used; each value is right-aligned within the field, and zero-filled where approriate. RGB data have components ordered such that red is least significant, and blue (or alpha) is most significant, subfield. Reads of framebuffer values via HOSTRW registers return undefined values for start-byte masked locations and for unused, trailing fields.

Pixel programmed I/O refers to host reads and writes of either pixels, overlays, popups, or CID planes. REX3 is set up by host to the desired mode via DRAWMODE0 and DRAWMODE1. The bits STOPONX, STOPONY should be zero, indicating one GIO word per primitive “GO”. For pixel reads, the DRAWMODE1 PREFETCH bit must be set to 1, with the DRAWMODE0 OPCODE=read. The set up of the DRAWMODE registers should be performed with a write to the GO (address+800H) command, prefetching the data, reducing the I/O latency of subsequent transfers. Pixel data may then be read from the HOSTRW register, again with the “GO” command. All PIO is context switchable. Reads and writes of the HOSTRW register when saving or restoring context should be made without the “GO” command (address 800H bit) being set.

To the REX3, DMA’s are indistinguishable from burst activity, in that the GIO activity is identical. Any distinction between these two modes are purely at the level of the kernel and, to some lesser extent, the write buffer (or MC). We will hereafter refer to any burst access as DMA; the term “word” refers to the width of data transferred in a bus cycle (for REX3, 4 or 8 bytes, depending on state of CONFIG register BUSWIDTH bit: see Section 4.1). To insure correct operation with the MC, all pixel DMA read transfers must be performed with DRAWMODE1 PREFETCH = 0. In addition, a pixel DMA read transfer may not begin until the graphics pipe is idle (STATUS GFXBUSY = 0). Pixel DMA transfers must be made with the “GO” command bit set.

DMA is supported for span and block addressing modes. In either case, each burst is restricted to left-to-right stepping per scanline; there is no support for right-to-left or mirroring. Span DMA is supported for arbitrary byte count and start byte values. Block DMA may have certain restrictions, as noted below.

There are two main categories of block DMA: linear and stride. A linear block DMA sends data across the bus in a single string, so that a block of data in framebuffer is packed into consecutive locations within this string and written or read as such to/from main memory. The DMA stride register in the write buffer is set to zero for this mode (linewidth = total transfer, linecount = one). There are REX3 restrictions on this type of block DMA: the start byte (SB) must be zero, and the width per scanline must be an integer number of bus words (64b words for GIO64 64b transfers, for instance). This width constraint applies to main memory storage, and not the actual width in framebuffer: the REX3 block addressing coordinates are programmed to exactly the desired size, and unused bytes contained in the last word per block row are ignored on writes. In effect, the end pixel of a block row must never be packed into the same GIO word as the start pixel of the next row: this is a REX3 restriction; to overcome this limitation, stride block DMA is used.

Stride block DMA consists of bursts of contiguous data which are each separated by a constant number of bytes, essentially an address gap. It supports a “virtual framebuffer” in main memory with which any block subset can be read or written, therefore not necessarily as a single contiguous, linearly addressed string. By definition, the linecount register in the write buffer is set to the number of framebuffer rows, and the linewidth is set to width of one row. The DMA stride register is usually set to a nonzero value, equal to the (byte) distance between bursts in a physically mapped main memory. (However, this value can be zero to handle the case of packed, linear data where last pixel of a block row is packed in the same main memory word as first pixel of next row: this memory word is then accessed at least twice, once per scanline.)

A single kernel call initiates the stride block DMA, which is decomposed by the write buffer into a GIO burst per scanline. Each of these scanline DMA's have identical BC value, but SB may vary per scanline and is calculated incrementally by the write buffer. (For example, say first SB is SB(0), calculated as Start_Address%8; then subsequent SB is calculated as SB(i) = {SB (i-1) + BC + Stride}%8, for 64b DMA).

All DMA's can be pre-empted and resumed, with the restriction that during the preemption period of a read DMA, no other access is made to the subsystem of REX3 pertaining to the pre-empted DMA. In short, REX3 contains two main subsystems: one for graphics, the other for the display control bus. DMA with the graphics subsection can be pre-empted by host in order to perform accesses across the display control bus, but not to the graphics. The converse is also true. In any event, no other DMA may be performed with REX3 during a read DMA preemption. In addition, rev. 0 and rev. 1 REX3 chips do not allow the reading of pixels (from HOSTRW) when a read DMA from DCBDATA is preempted. Similarly, rev. 0 and rev. 1 REX3 chips do not allow display control bus data (from DCBDATA) to be read while a read DMA from HOSTRW is preempted. Access to the STATUS, CONFIG, and DCBRESET registers are a third category, or subsection, for which this rule applies. Therefore, STATUS may be read by host during any DMA preemption.

Notes: Unlike REX1, the REX3 supports nonuniform SB for block DMA: therefore the logic will subtract the number of pixels represented by SB from XSTART at start of each burst. (Programmers note: you will not have to subtract SB from XSTART, which you did for REX1.)

RWDOUBLE	RWPACKED	RWDEPTH(1:0)	Host Pixel Packing GIO_DATA(63:0)
0	0	00	
0	0	01	
0	0	10	
0	0	11	
0	1	00	
0	1	01	
0	1	10	
0	1	11	
1	0	00	
1	0	01	
1	0	10	
1	0	11	
1	1	00	
1	1	01	
1	1	10	
1	1	11	

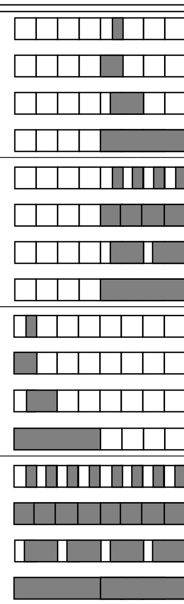


Table 0.2.2.2. HOSTRW pixel packing modes, (big endian format illustrated).

The following table summarizes the above cases:

Table 24: Summary of PIO and DMA Cases

Case	Context Switchable	Bursts	PREFETCH	SB	BC	STOPONX,Y
PIO Write	yes	--	--	0/4	8/4	0,0
PIO Read	yes	--	yes	0/4	8/4	0,0
Linear DMA Span Write	yes	single	--	0-7	length	0,0
Linear DMA Span Read	no	single	no	0-7	length	1,0
Linear DMA Block Write	yes	single	--	0	length (length%8 =0)	0,0
Linear DMA Block Read	no	single	no	0	length (length%8 =0)	1,1
Stride DMA Block Write	yes	per row	--	0-7 per row	width	0,0
Stride DMA Block Read	no	per row	no	0-7 per row	width	1,0

(note 1: STOPONX,STOPONY are from DRAWMODE0 register; PREFETCH is from DRAWMODE1.)

(note 2: above is for 64b GIO transfers; for 32b case, SB 0-7 is then 0-3; length%8 is then length%4.)

(note 3: when a span crosses a page boundary, an additional burst is done; BC is decomposed per burst.)

3.11 FIFO Management

A bus timeout counter (CONFIG register TIMEOUT field) is provided (1-4.3 usec) to generate a FIFO_INT_N interrupt during continuous GRXDLY stalling for host I/O; graphics FIFO is enlarged to 32 deep, doubles. (note: GRXDLY is asserted whenever FIFO level meets or exceeds CONFIG GFIFODEPTH and CONFIG GFIFABOVEINT is set; at timeout, this FIFOMAX GRXDLY stall is disabled, while FIFO_INT_N remains asserted until the FIFO drains below the GFIFODEPTH level or GFIFOABOVEINT is cleared, enabling a “below level” interrupt). DMA uses this mechanism also, though it should never result in a timeout because the FIFO drain rate is sufficient to reset the timer-counter frequently.

In the event that the FIFOMAX level is set too high so that FIFO overflow would occur, the following would happen: first, the FIFO would not push overflow data; second, this data would then be lost; third, this is ensured by the h/w disallowing push when (fifolevel=32 & !pop). Occurrence of this condition, for system prototyping and debug, will be visible via assertion of GRXDLY during any “overflow push” clocks. Logic analyzer trigger on (FIFO_INT_N & GRXDLY) will capture it; should never trigger in a correctly configured/ operating system.

Host should check STATUS or USER_STATUS for graphics idle (GFXBUSY=0) and GFIFO empty (GFIFOLEVEL = 0x00) before beginning a read or a series of reads, to avoid bus timeout. User programs should check USER_STATUS so that interrupt information contained in the STATUS register is not cleared. Additionally, there are several cases which yield worst-case latency for framebuffer reads which the system should accomodate without generating a bus timeout: (a) any PIO read may be delayed by VRAM transfer and refresh cycles; this latency increases for slower video rate and interlaced display; (b) maximum memory cycles required for a read is for a double-word of packed 8b or 4b data, unaligned in VRAM; (c) the transition between block rows is stalled by the graphics pipeline so that current scanline is finished through the read packer before the next scanline read is initiated (this is to accomodate the Y swizzle in memory). The combined effect of all these cases will yield a worst-case latency for reads which the system must accept. (**we should have numbers for this before tapeout!**).

3.12 Context Switching

REX3 supports context switching except for during preempted read dma. There are two main contexts which may be switched: graphics context, and display bus context. Graphics context includes the X,Y values, colors, and all other modifiers or modes which affect writing into, or reading from, the framebuffer. Display bus controller context, by definition, includes those registers specifying target device, interface protocol and timing, and data registers used for display bus transactions. Before any context switching can take place, the host must poll the STATUS register and wait for the appropriate BUSY bit to be cleared (GFXBUSY for graphics, BACKBUSY for display bus backend). At such time, the context registers are considered to be stable.

A complete graphics context save is performed by first checking for GFXBUSY=0 and GFIFOLEVEL = 0x00, then reading all registers except display control bus registers. Only those registers which have a read format listed need be saved. In many cases, however, it is likely that only a subset of these registers need be saved; it is up to the application to decide this. The read of the HOSTRW or any other register must not be issued with a “GO” (address+800H) command.

A complete graphics context restore is done by writing back all the registers which were saved. There is one complication the host must handle during context save/restore of the SLOPERED register: the saved value must be converted from a 2’s complement (s12.11) to a signed magnitude (s(8)12.11) format before writing back to REX3. Restoral of COLORRED must be done with DRAWMODE bit RGBMODE=1 in order to circumvent the 12b CI formatting process. Also note that XSAVE should be restored after XSTART. The write to the HOSTRW or any other register must not be issued with a “GO” (address+800H) command. The restored process may be started immediately.

Display bus context switching is done simply by reading or writing the appropriate registers, after waiting for BACKBUSY=0. If non-atomic transfers are performed to or from the devices on the display bus, their context will also need to be saved/restored; however, this should no longer be necessary, now that REX3 supports packing and unpacking of multiple-byte data onto the bus.

3.13 Display Control Bus

The host communicates with devices on the Display Control Bus (DCB) by first writing to the DCBMODE register, and then writing to or reading from the DCBDATA register. Data written to both the DCBMODE and DCBDATA registers pass through the BFIFO (backend fifo) prior to their being used by the DCB state machine or transferred on the DCB. The DCB state machine will empty the BFIFO prior to starting a read operation on the DCB.

Slave device selection is made by the DCBADDR(3 downto 0) field of the DCBMODE register. No physical device attached to the DCB will ever be allowed to respond to the reserved DCBADDR = X"F". DCBADDR decoding for the Newport Graphics subsystem is as follows:

DCBADDR	Device
0000	VC2
0001	CMAP0 and CMAP1 (write only)
0010	CMAP0
0011	CMAP1
0100	XMAP0 and XMAP1 (write only)
0101	XMAP0
0110	XMAP1
0111	RAMDAC
1000	Video CC1
1001	Video AB1
1010 to 1110	undefined
1111	reserved

Table 25: Newport Graphics DCBADDR Decoding

The register to be accessed within the device selected by DCBADDR is determined by the DCBMODE DCBCRS(2 downto 0) field. If the DCBMODE ENCRSINC bit is set, then DCBCRS will increment following the transfer of each byte on the DCB.

The protocol used to transfer data on the DCB is described by the ENASYNCACK and ENSYNCACK fields in DCBMODE.

If ENASYNCACK is set, an asynchronous handshake protocol will be used to transfer data across the DCB. The asynchronous handshake protocol runs as follows: The REX3 will assert DCB_CS_N when it is presenting valid data (for write cycles) or ready to accept data (for read cycles). The slave device asserts DCB_ACK_N when it has accepted data (write cycles) or is returning the requested data (read cycles). When the REX3 detects that DCB_ACK_N, synchronized to the 33 MHz. GIOCLK, has been asserted, it will de-assert DCB_CS_N. The slave device will signal then de-assert DCB_ACK_N. The REX3 will not begin another transfer on the bus until a synchronized de-asserted DCB_ACK_N has been detected. Data can be transferred at a peak rate of 1 byte/ 4 cycles.

If ENSYNCACK is set, a synchronous handshake protocol will be used to transfer data across the DCB. When the REX3 is presenting valid data (write cycles), or is ready to accept data (read cycles), it will assert DCB_CS_N. When the slave device is accepting the data, or is returning the requested data, it will assert DCB_ACK_N in the current GIOCLK cycle. This protocol will allow the transfer of data at a peak rate of 1 byte/cycle.

If neither ENASYNCACK nor ENSYNCACK is set, then data will be transferred at a rate determined solely by the DCBMODE cycle timing parameters CSSETUP(4 downto 0), CSWIDTH(4 downto 0), and CSHOLD(4 downto 0). DCB_RW_N, DCB_ADDR, DCB_CRS, and (for write cycles) DCB_DATA will be valid for CSSETUP cycles prior to asserting DCB_CS_N. DCB_CS_N will then be asserted for (CSWIDTH + 1) cycles. DCB_CS_N will then be de-asserted for CSHOLD cycles prior to changing any of the DCB control signals. For read transfers, data will be sampled by the REX3 at the end of last cycle in which DCB_CS_N is asserted.

The DATAWIDTH (1 downto 0) field describes the number of bytes to transfer from each word written to or read from DCBDATA0 or DCBDATA1 when ENDATAPACK is cleared. It is used to simplify the transfer across the GIO64 bus of 3-byte (RGB triplet) quantities packed into words. When ENDATAPACK is set, all bytes written to or read from DCBDATA will be transferred across the DCB

The SWAPENDIAN bit, in conjunction with the DATAWIDTH field, is used to support the OpenGL SWAP_ENDIAN pixel packing attribute. When set, the ordering of bytes within short and long width data is reversed.

Once the DCBMODE register has been written to, subsequent reads to and writes from the DCBDATA register will result in data transfers on the DCB, using the specified timing and protocol.

3.14 Chip Reset and Initialization

Following reset, the REX3 assumes that it is attached to GIO64 bus that is physically 32 bits wide, and that the registered transceivers that isolate the pipelined GIO64 bus from the non-pipelined GIO64 bus are physically present. If the registered transceivers are not present (as in the Sapphire system), the host must clear the EXTREGXCVR bit in the CONFIG register prior to performing *any* reads from REX3 registers. If the REX3 is attached to a GIO64 bus which is physically 64 bits wide, the BUSWIDTH bit in the CONFIG register should also be set at this time. If the REX3 is installed in a system with a GIO32 bus master, the GIO32MODE bit in the CONFIG register must be set.

4 System Interface

4.1 GIO64 Bus Interface

The REX3 is a pipelined GIO64 slave device. The REX3 does not check or generate parity, so the GIO64 bus parity signals, P_ADP(7 downto 0) and P_VLD_PARITY_N, are ignored. Only the two least significant SLOT_NUMBER pins from the GIO64 bus are brought into the REX3 for address comparison. The two most significant SLOT_NUMBER pins are assumed to be B”11”. This implies that the base address of the REX3 and associated Newport Graphics subsystem is at X”1F000000”, X”1F400000”, X”1F800000”, or X”1FC00000”. Consequently, REX3/Newport Graphics subsystems may only be placed in GIO64 slots C, D, E or F. Multiple head operation (up to four displays) is achieved by populating slots C, D, E, and F with REX3/Newport Graphics subsystems.

Two interrupts are returned from the Newport Graphics subsystem by REX3. VV_INT_N is the sum of the VERT_INT_N (vertical retrace) signal (from the VC2), and the VIDEO_INT_N signal (from the Express Video Option). The REX3 will latch the occurrence of a falling edge on the VERT_INT_N input and assert the VV_INT_N interrupt. A ’low’ level on the VIDEO_INT_N input will also result in VV_INT_N being asserted. The host determines the source of VV_INT_N by reading the STATUS register. When the STATUS register is read, VRINT, the latch associated with VERT_INT_N, is cleared, removing that contribution to VV_INT_N. User code which is not willing to service VERT_INT_N interrupts should read the USER_STATUS register, which does not clear the VRINT latch.

FIFO_INT_N is generated whenever the number of entries in either the graphics fifo (GFIFO) or the display control bus fifo (BFIFO) has exceeded a programmed level for a programmed amount of time, or when the number of entries in either the GFIFO or BFIFO has fallen below a programmed level. REX3 fifo interrupt behavior is therefore determined bythe CONFIG register BFIFODEPTH, GFIFODEPTH, BFIFOABOVEINT, GFIFOABOVEINT, and TIMEOUT fields. Whenever a ’fifo above’ interrupt is generated, this occurrence is latched in either the STATUS register BFIFO_INT or GFIFO_INT field. Reading the STATUS register will reset these bits, but FIFO_INT_N will remain asserted as long as the interrupting condition exists. The latching ’fifo above’ status is intended to provide the host with a means of identifying the source of spurious interrupts. User code should only read status from USER_STATUS, to prevent the uncontrolled clearing of the BFIFOABOVEINT, GFIFOABOVEINT, and VRINT interrupt status bits.

The REX3 will operate as a pipelined GIO64 slave with or without the presence of external registered transceivers. The REX3 assumes that the external registered transceivers that define the pipelined GIO64 bus are present. The absence of the external registered transceivers is communicated to the REX3 by programming CONFIG register EXTREGXCVR bit to B”0”. When REX3 is installed in a system without external registered transceivers, this bit must be programmed to B”1” prior to any read operation.

The REX3 will respond to both 64-bit and 32-bit wide GIO64 bus masters, as determined by the P_GSIZE64 signal. The REX3 will operate with a GIO64 bus that is physically either 64 bits wide or 32 bits wide, as determined by the CONFIG register BUSWIDTH bit.

The REX3 will follow the GIO32 bus protocol when the CONFIG register GIO32MODE bit is set. In this mode, data transferred during the GIO bus byte count cycle will be interpreted according to GIO32 protocol convention.

The REX3 supports both little-endian and big-endian addressing conventions in GIO64 mode.

Please consult the GIO64 Bus Specification, and the Graphics IO (GIO) Bus Specification for precise descriptions of the GIO64 and GIO32 protocols.

4.2 Display Control Bus Interface

The Display Control Bus (DCB) is an 8-bit, 33 MHz bus controlled by the REX3, interfacing the REX3 to the XMAP5s, VC2, CMAPs, RAMDAC, and Video Option in the Newport Graphics subsystem. In addition to the 8 bidirectional data lines (DCB_DATA(7 downto 0)), the bus includes 4 device address lines (DCB_ADDR(3 downto 0)), driven by the REX3, which are externally decoded to produce 15 device chip select signals. The bus also includes 3 command/register select lines (DCB_CRS(2 downto 0)), allowing eight registers to be accessed within each device. A data transfer direction line (DCB_RW_N), a command strobe line (DCB_CS_N), and an acknowledge signal (DCB_ACK_N) complete the set of bus signals.

All signals on the DCB sourced by the REX3 (DCB_DATA, DCB_ADDR, DCB_CRS, DCB_RW_N, and DCB_CS_N) change on the rising edge of the 33 MHz GIO_CLK. All inputs (DCB_DATA and DCB_ACK_N) are sampled with the rising edge of GIO_CLK.

DCB_ADDR(3 downto 0) = X"F" is reserved as a null-device chip select. No physical device is allowed to respond to transactions to this reserved address.

The DCB supports different slave device timing requirements, synchronous and asynchronous operation, and data transfer protocols with or without acknowledgement. These different modes of operation are programmed through fields in the DCBMODE register.

Driver conflict and bus contention are avoided by having the REX3 insert at least two idle cycles (DCB_DATA tri-stated, DCB_CS_N = B"1", DCB_ADDR = X"F") between transactions of different directions (read followed by write, or write followed by read), and between transactions to or from different slave devices. As the DCB_ACK_N signal may be shared by multiple deveics, a slave device must return this signal to the inactive ("1") state prior to tri-stating its driver.

Please consult the Display Control Bus Specification for precise descriptions and definitions of the Display Control Bus protocol.

4.3 VRAM Interface

The memory controller runs at 66MHz. and is made of 5 state machines. Following are the four state machines and their respective functions:

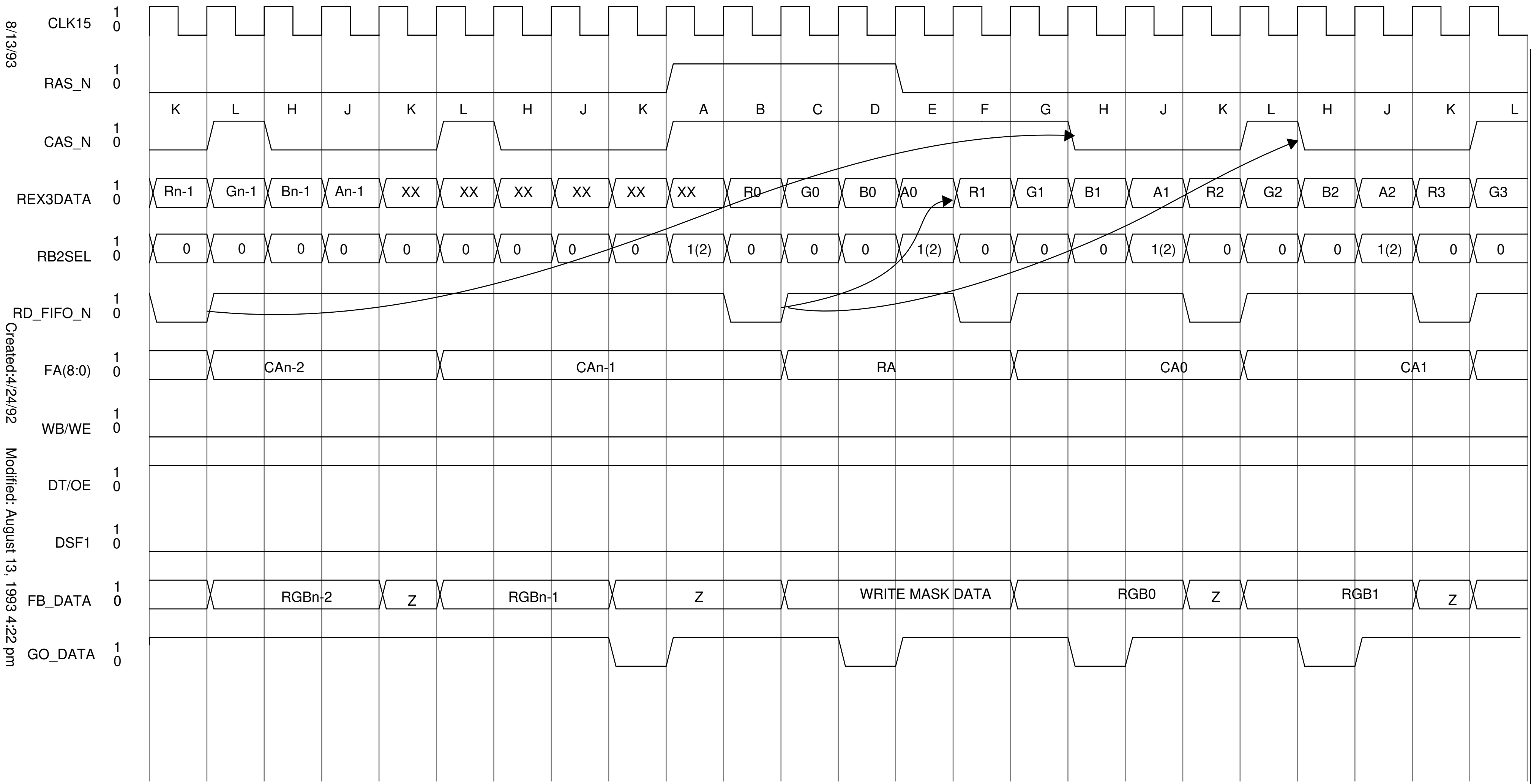
1.	CONTROL MODULE	Controls the other state machines
2.	TR_FSM	Performs screen refresh and memory refresh
3.	LD_REG_FSM	Loads write mask and color regs. in RB2 and Vrams respectively
4.	WRITE_FSM	Performs write only operations to Vram (including block writes)
5.	RMW_FSM	Performs read, read modify write and read/read modify write cycles.

The WRITE_FSM and RMW_FSM state machines keep the Vrams in page mode unless a page miss or a request to transfer to another state machine is requested. LD_REG_FSM is invoked when the host reads the chip is not busy and wants to load a new write mask or new color register value in the Vrams.

When a screen refresh request is made, the state machines make sure there are no pixels in the data pipe before honoring the request. The memory controller in all four banks operate independently from each other.

A page mode cycle takes 4 (66MHz) clocks. A full ras cycle is 11 clocks.

Various timing for the frame buffer is shown in the next few pages.

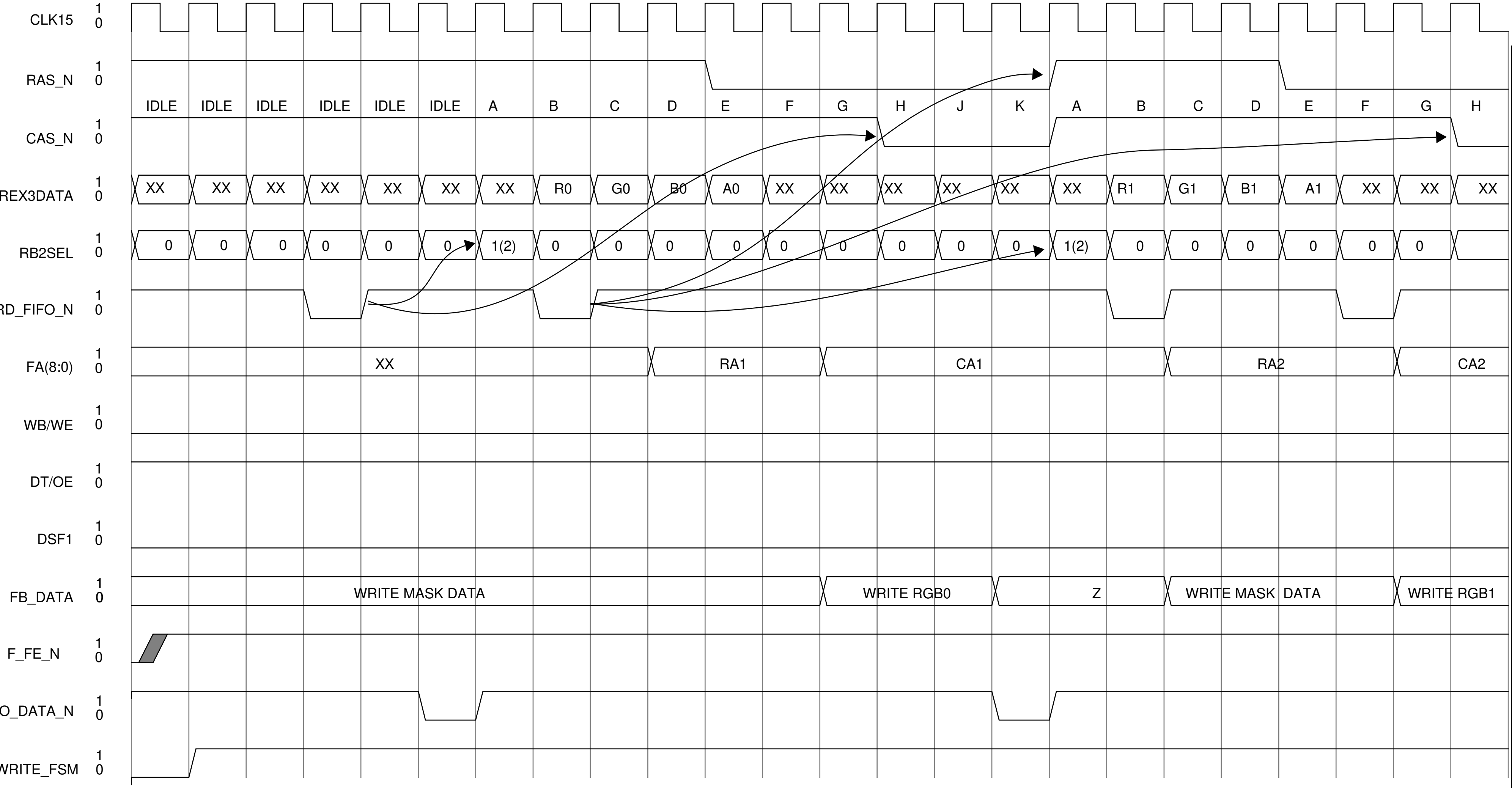


FRAME BUFFER PAGE MODE WRITES 8 OR 24 BITS (4 components) PIXEL PLANES ONLY

Created:4/24/92

Modified: August 13, 1993 4:22 pm

SILICON GRAPHICS PROPRIETARY and CONFIDENTIAL

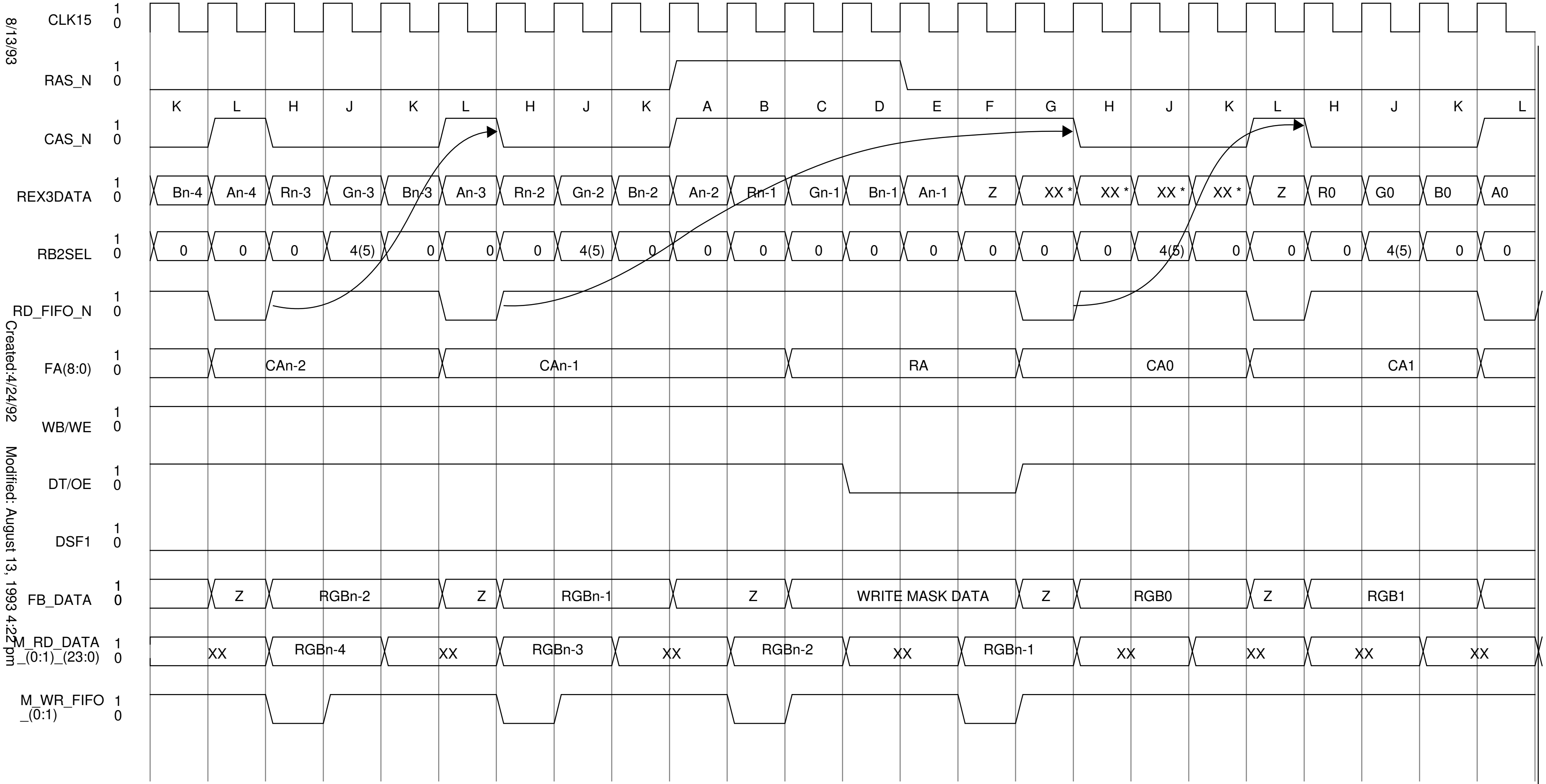


FRAME BUFFER PAGE MODE WRITES 8 OR 24 BITS (4 components) PIXEL PLANES ONLY

STARTING WITH NO PIXELS IN THE PIPE + CONSECUTIVE PAGE MISSES

Created:4/24/92

Modified: August 13, 1993 4:22 pm



FRAME BUFFER PAGE MODE READS 8 OR 24 BITS ANY PLANES

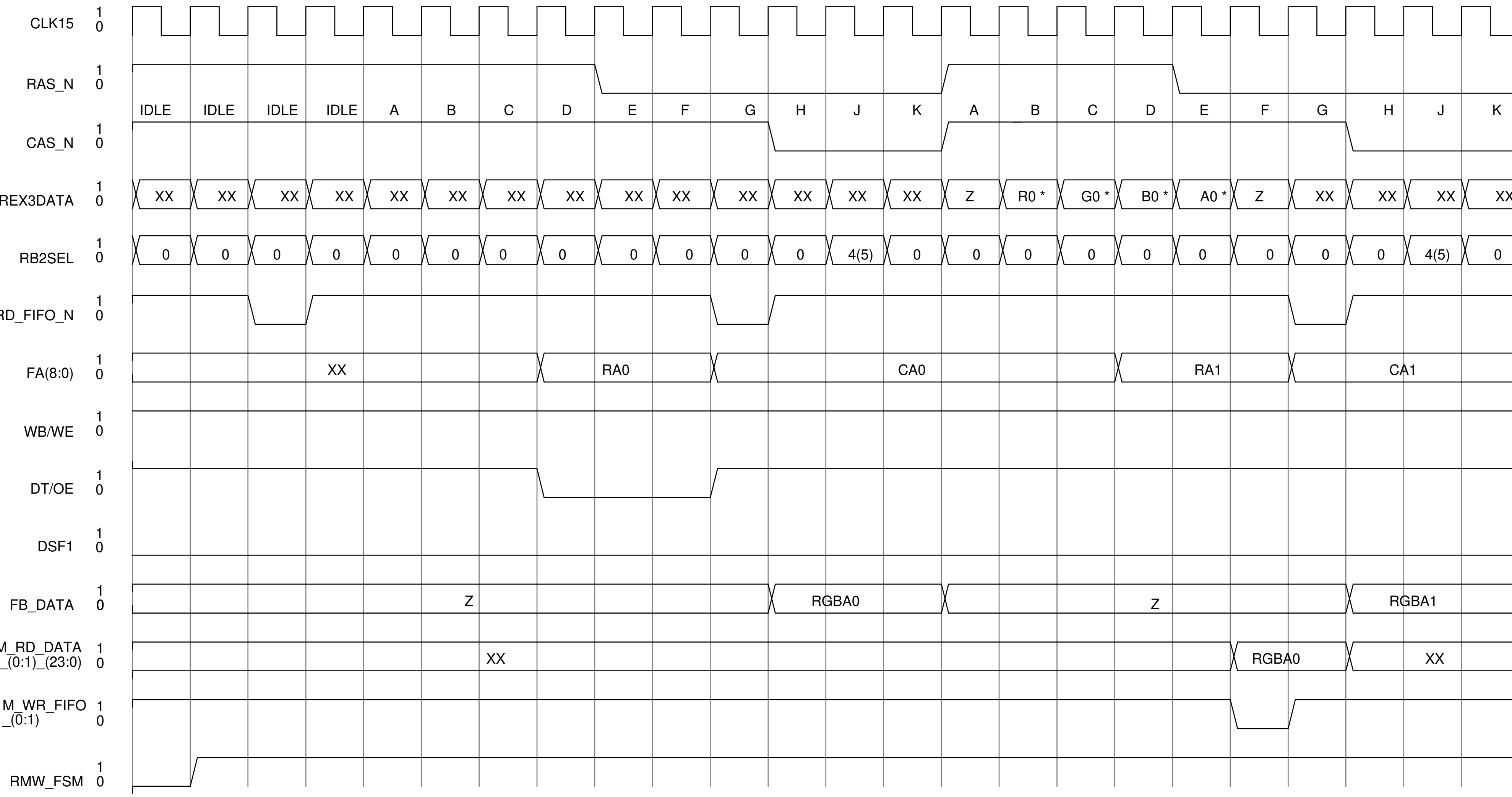
* Dirven by REX3

Created:4/24/92

Modified: August 13, 1993 4:22 pm

Created:4/24/92

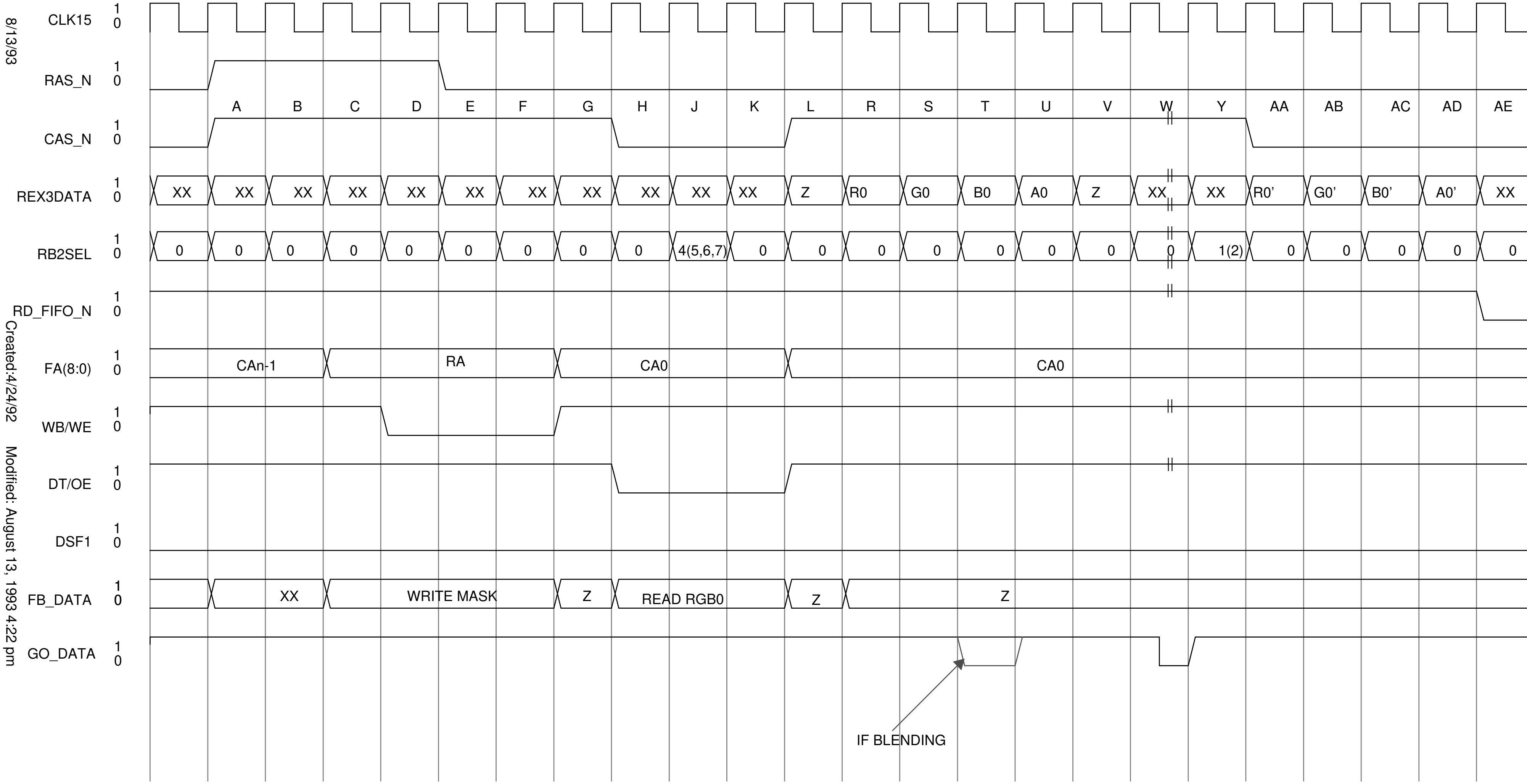
Modified: August 13, 1993 4:22 pm



FRAME BUFFER READS 8 OR 24 BITS ANY PLANES

STARTING WITH NO PIXELS IN THE PIPE AND CONSECUTIVE PAGE MISSES

* Dirven by RB2



FRAME BUFFER READ MODIFY WRITE 8 AND 24 BITS

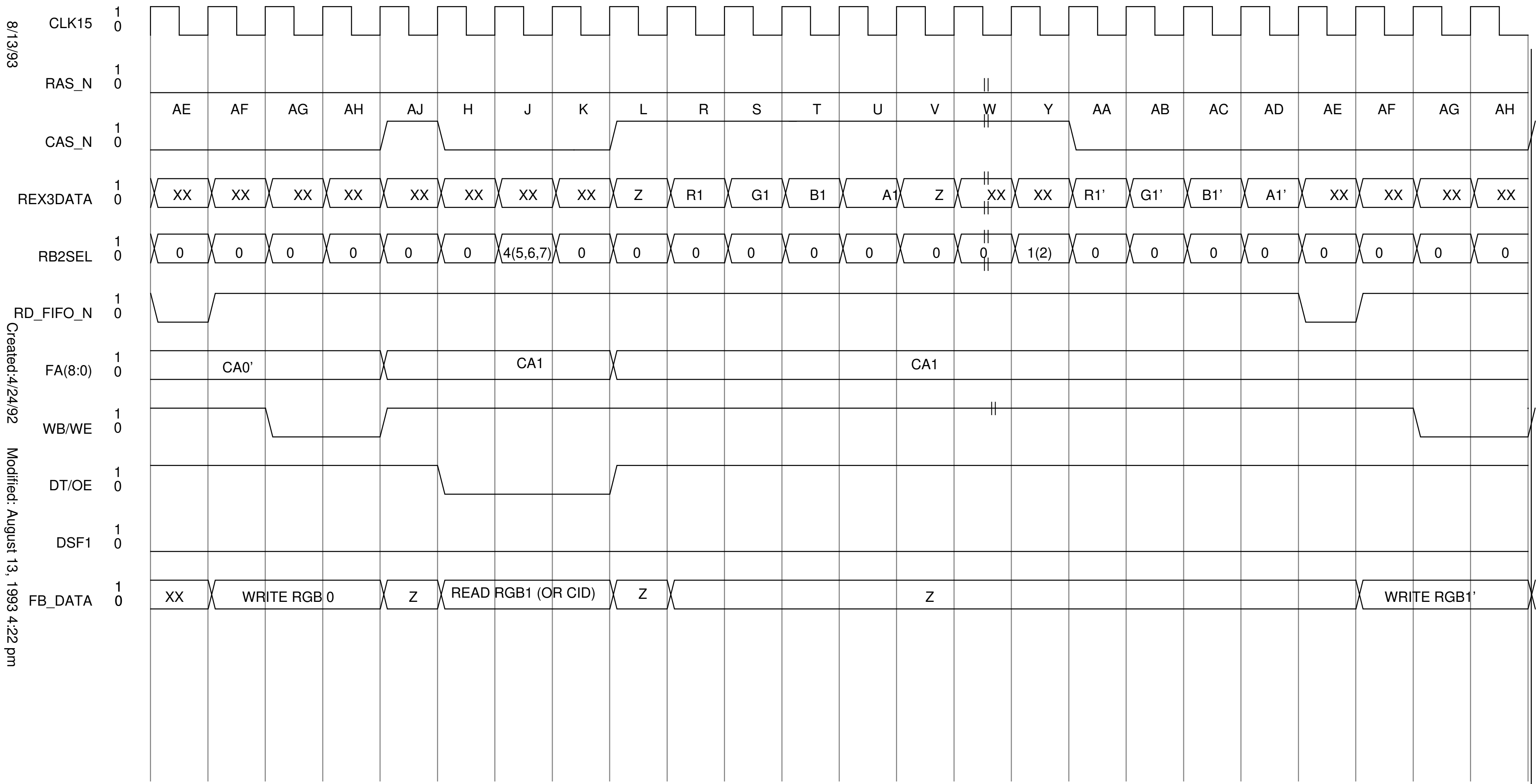
CONTINUED ON NEXT PAGE.

PIXEL OPS.

Non cid chkd blend in pixel planes
Non cid chkd ccomp in any planes
Cid chkd writes in pixel planes
Non cid chkd writes in aux planes

AUX Planes = cid or pup or olay

Created:4/24/92 Modified: August 13, 1993 4:22 pm



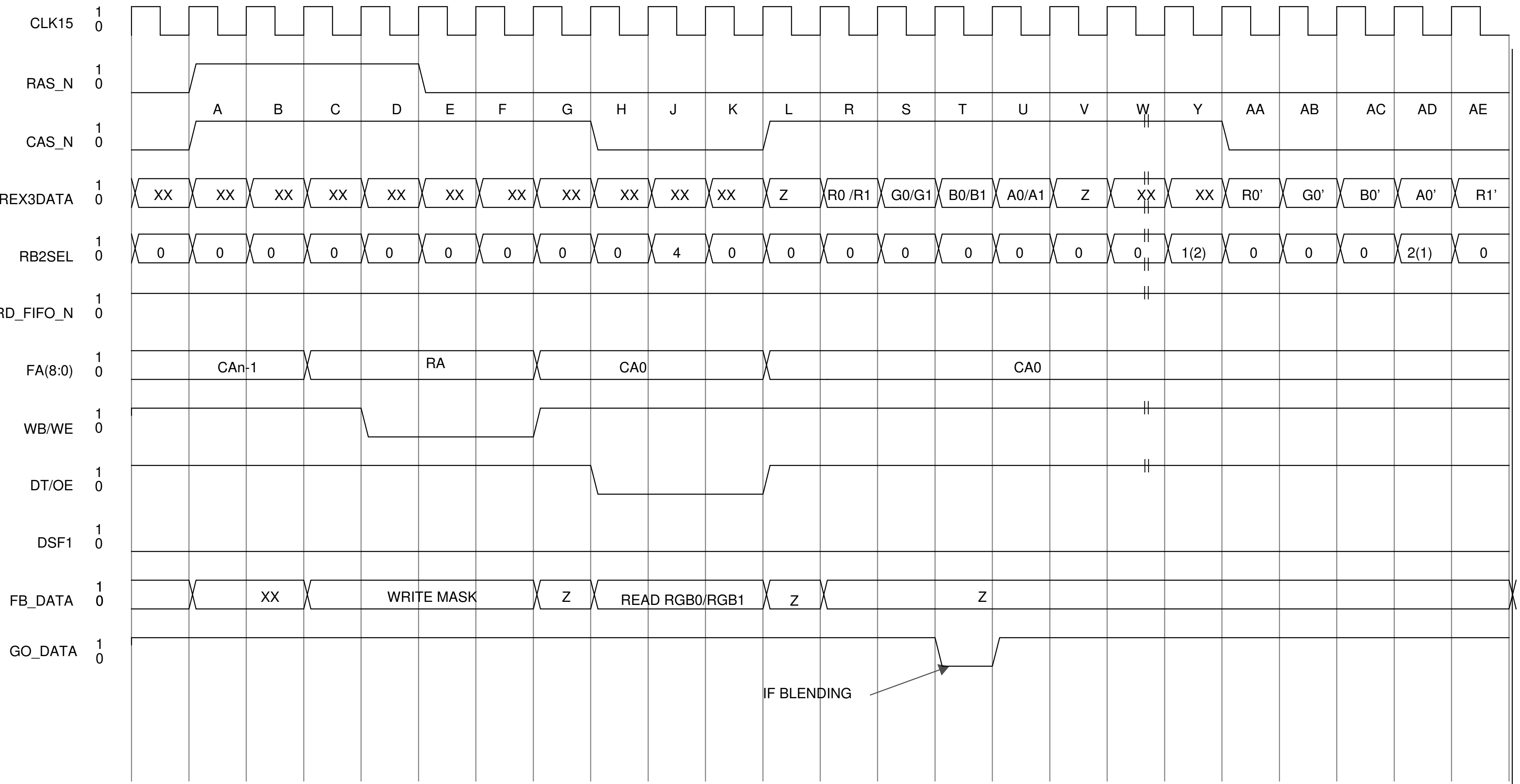
PAGE MODE READ MODIFY WRITE 8 AND 24 BITS.

(CONTINUED)

PIXEL OPS.

Non cid chkd blend in pixel planes
Non cid chkd ccomp in any planes
Cid chkd writes in pixel planes
Non cid chkd writes in aux planes

Created:4/24/92 Modified: August 13, 1993 4:22 pm



Created:4/24/92

Modified: August 13, 1993 4:22 pm

SILICON GRAPHICS PROPRIETARY and CONFIDENTIAL

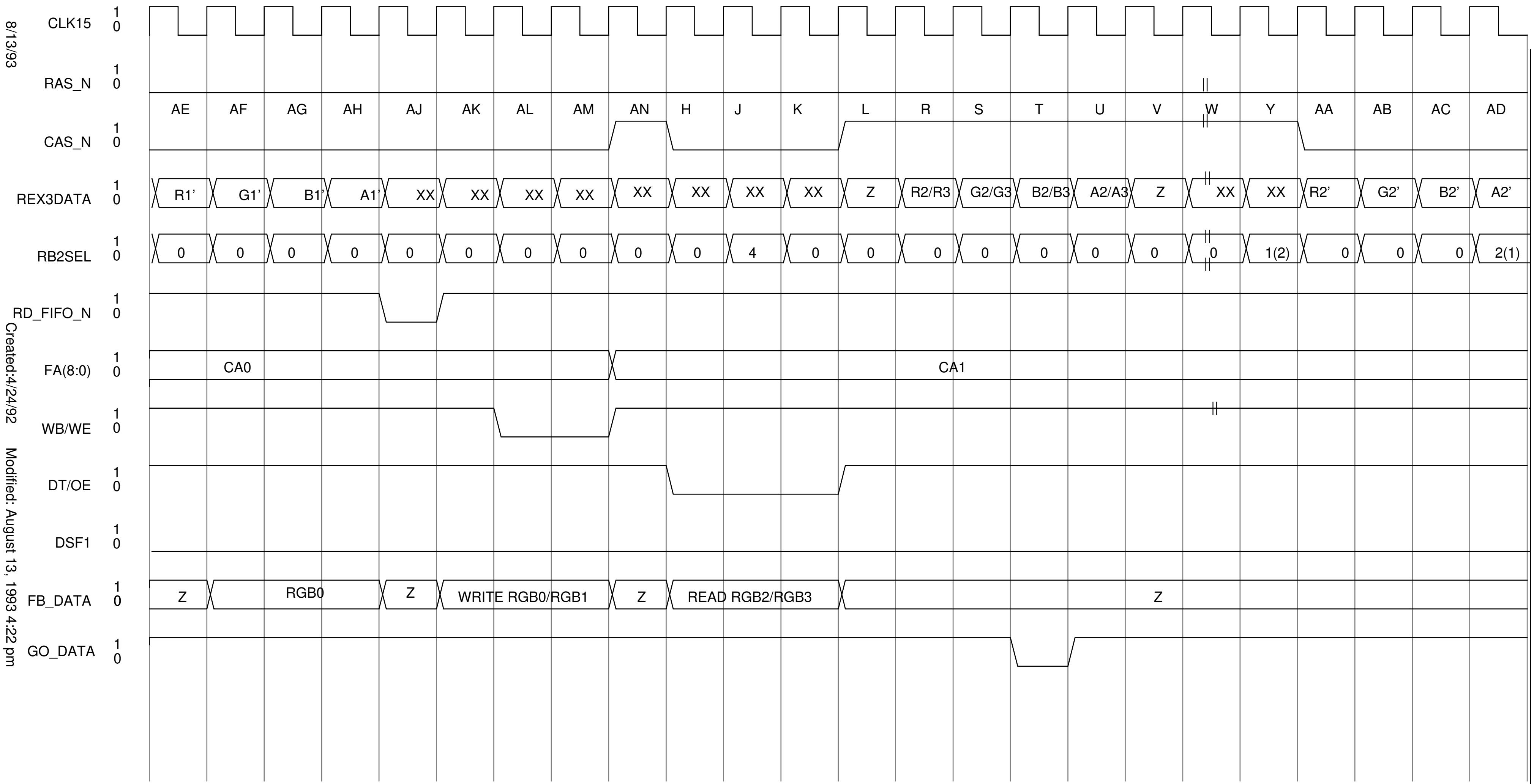
FRAME BUFFER READ MODIFY WRITE 8 AND 24 BITS. WITH VALID PIXELS IN BOTH SUBBANKS TO BE BLENDED

CONTINUED ON NEXT PAGE.

PIXEL OPS.

Non cid chkd blend in pixel planes

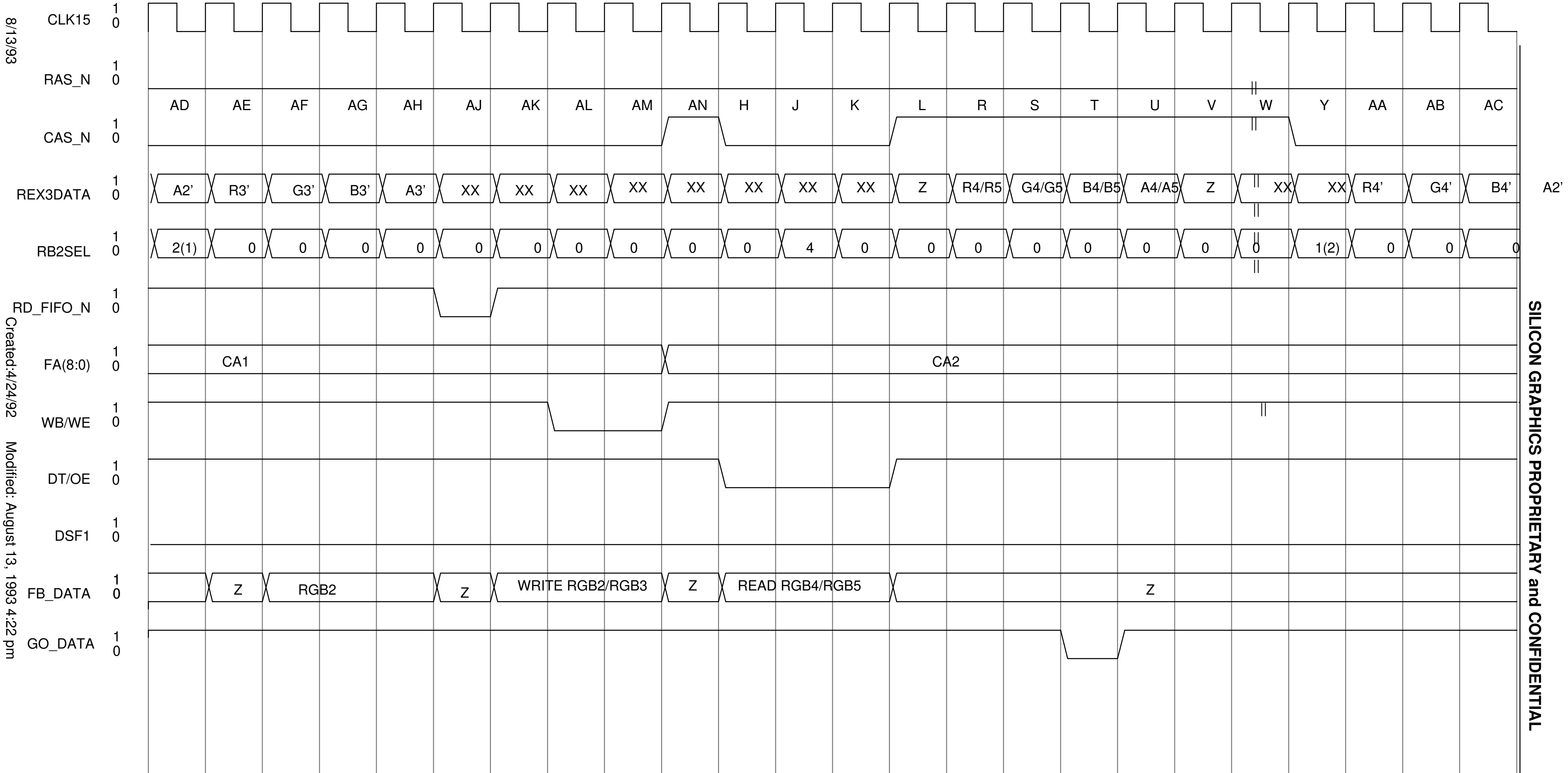
Created:4/24/92
Modified: August 13, 1993 4:22 pm



PAGE MODE READ MODIFY WRITE 8 AND 24 BITS. WITH VALID PIXELS IN BOTH SUBBANKS TO BE BLENDED.
(CONTINUED)

PIXEL OPS.

Non cid chkd blend in pixel planes



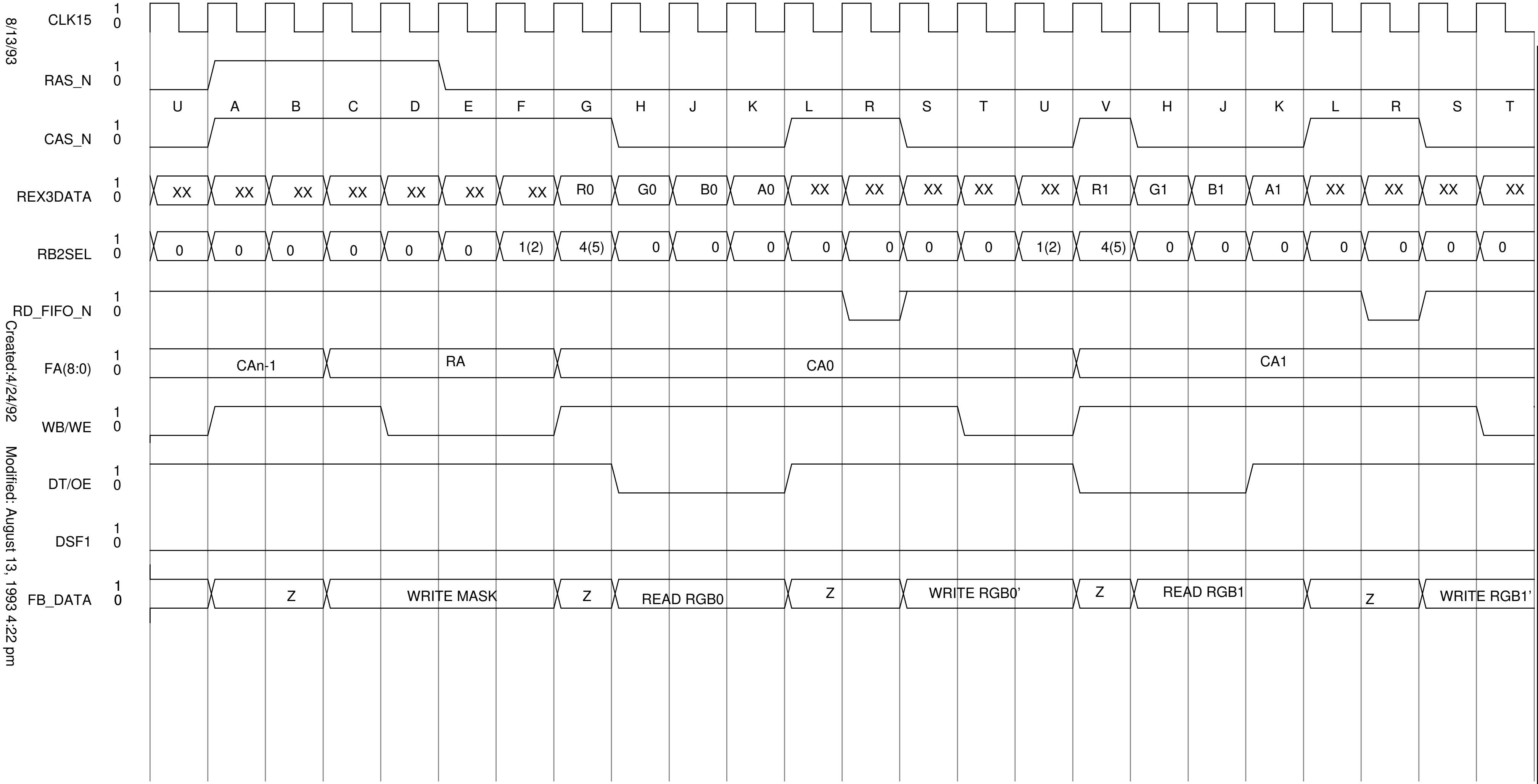
Created:4/24/92 Modified: August 13, 1993 4:22 pm

SILICON GRAPHICS PROPRIETARY and CONFIDENTIAL

PAGE MODE READ MODIFY WRITE 8 AND 24 BITS. WITH VALID PIXELS IN BOTH SUBBANKS TO BE BLENDED.
(CONTINUED)

PIXEL OPS.

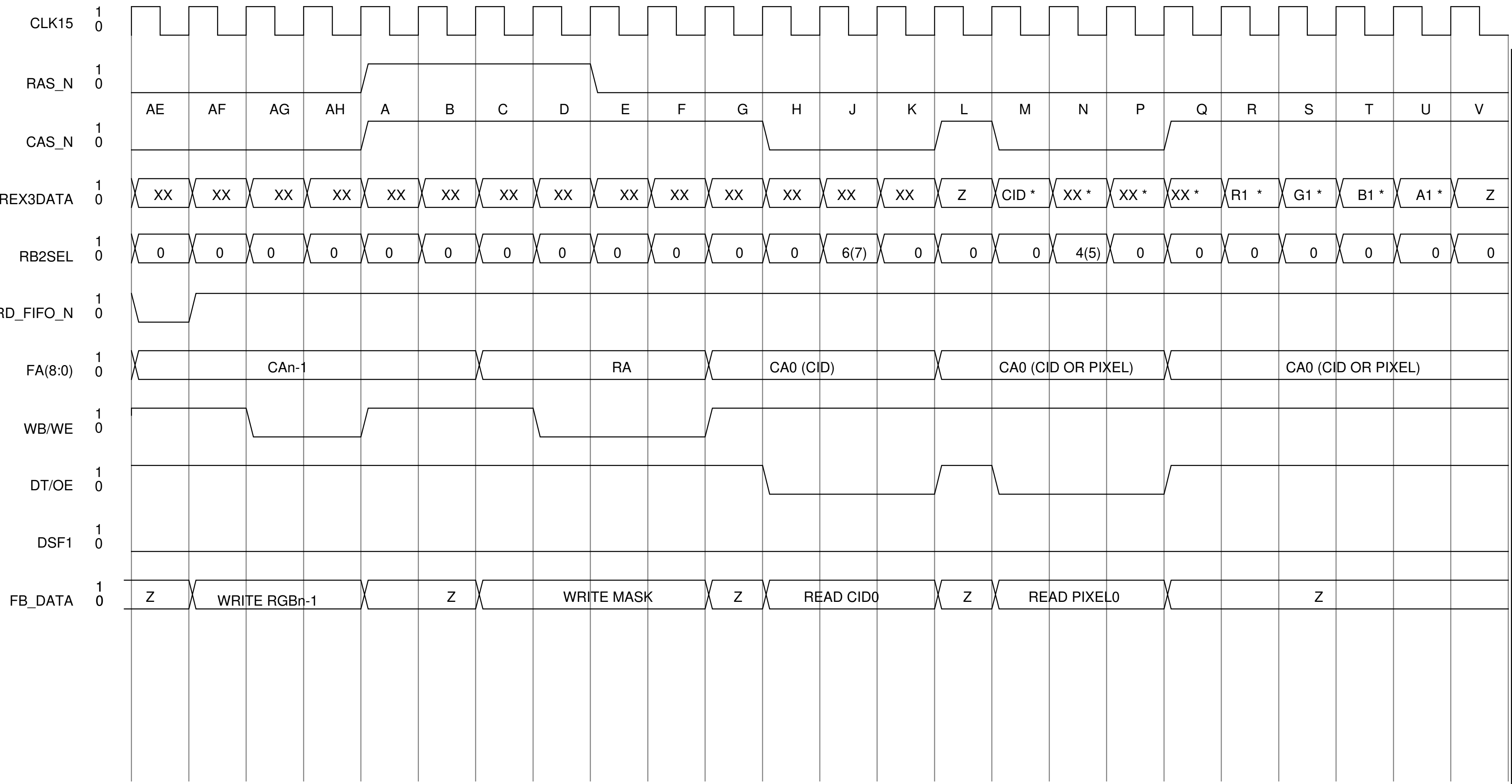
Non cid chkd blend in pixel planes



FRAME BUFFER READ MODIFY WRITE 8 OR 24 BITS (NON CID CHK LOGIC-OP PIX WRITE) FAST_X

LOGIC OP ONLY IN ANY PLANES

Created:4/24/92
Modified: August 13, 1993 4:22 pm



READ / READ MODIFY WRITE CYCLE

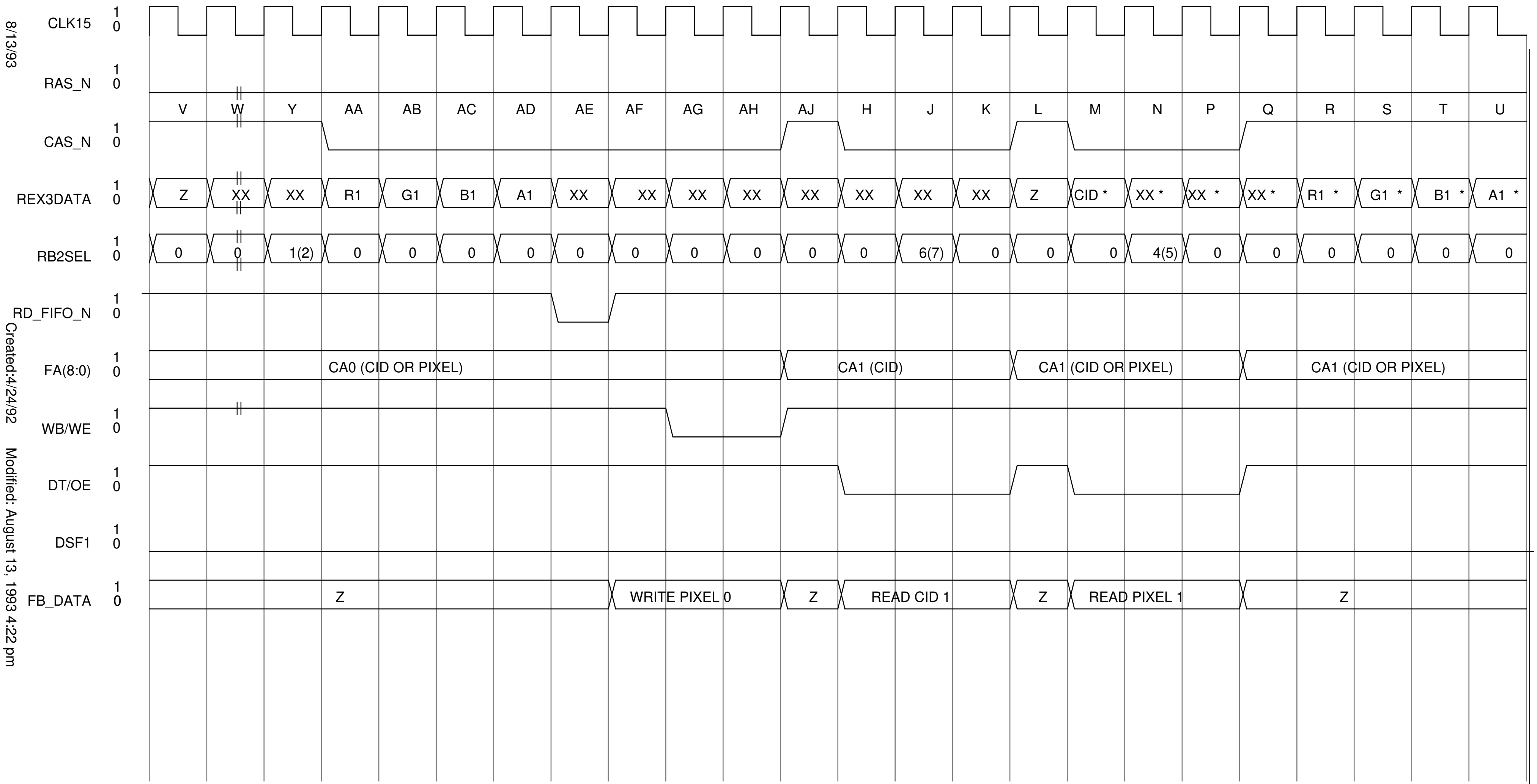
CONTINUED ON NEXT PAGE

PIXEL OPS

Cid chkd ccomp/logicop writes in any planes
Cid chkd blend in pixel planes

* Driven by RB2

Created:4/24/92 Modified: August 13, 1993 4:22 pm



READ / READ MODIFY WRITE CYCLE

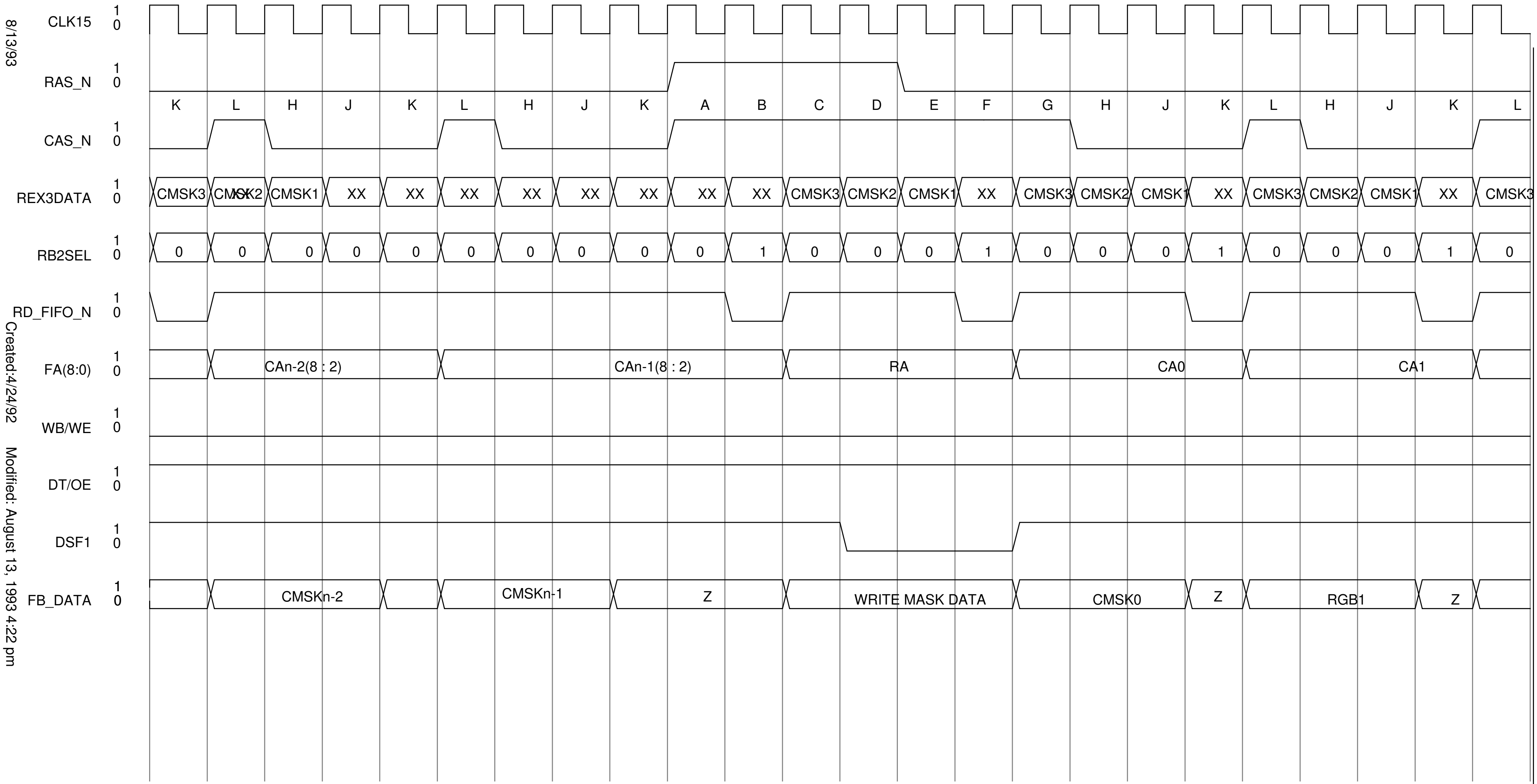
(CONTINUED)

PIXEL OPS

Cid chkd ccomp/logicop writes in any planes
Cid chkd blend in pixel planes

* Driven by RB2

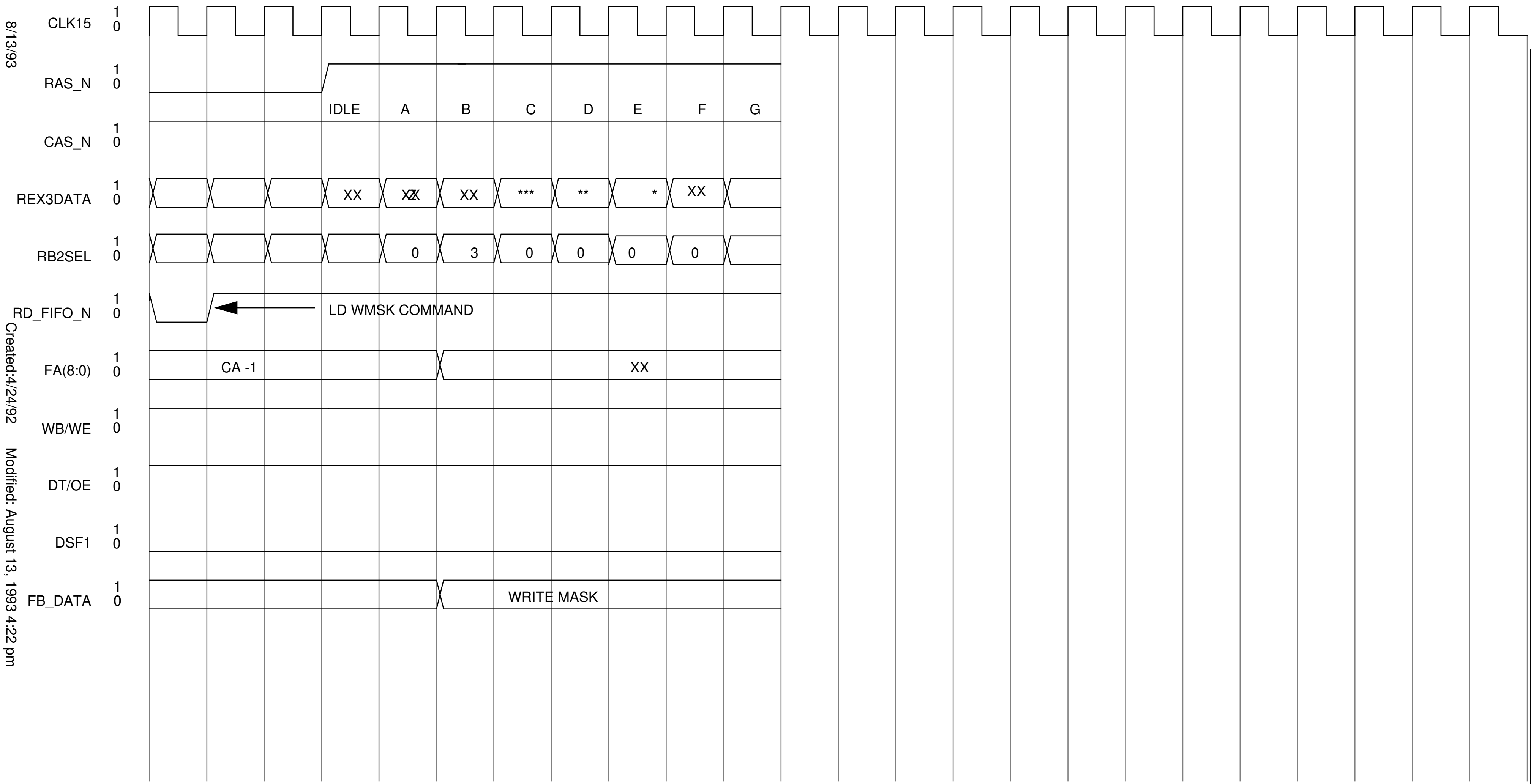
Created:4/24/92
Modified: August 13, 1993 4:22 pm



BLOCK MODE WRITE

Created:4/24/92

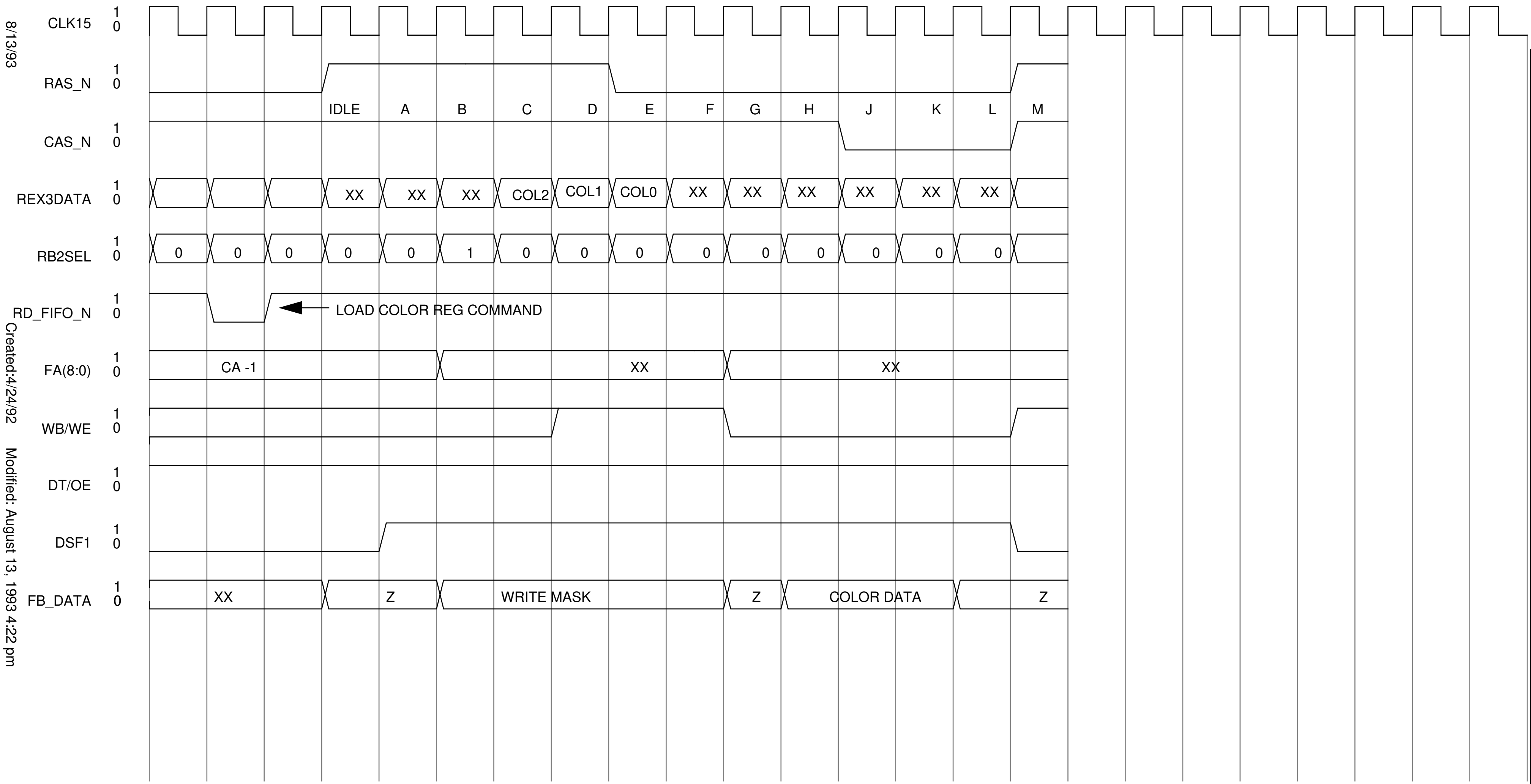
Modified: August 13, 1993 4:22 pm



LOAD WRITE MASK OPERATION.

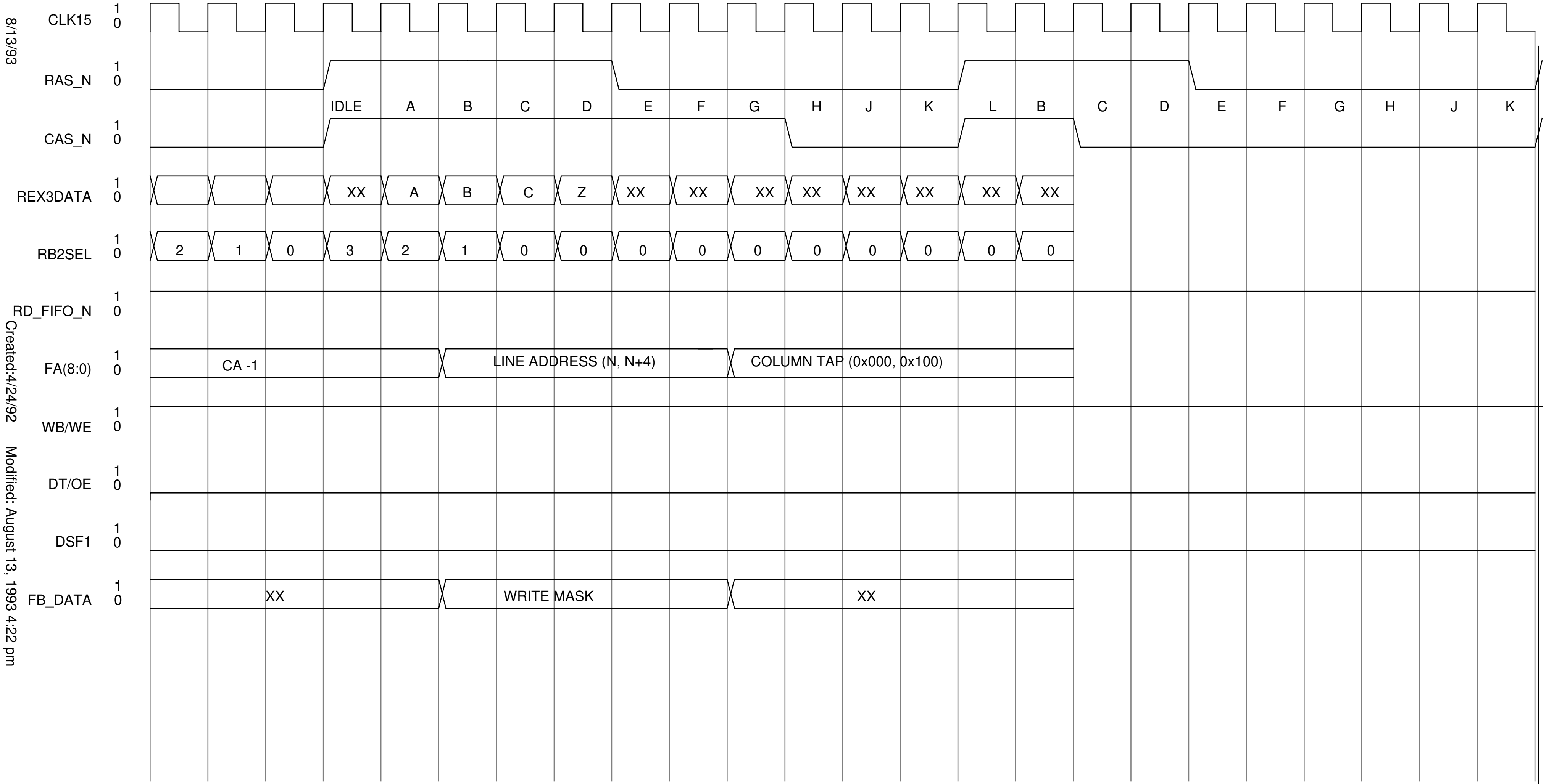
	SUB BANK0	SUB BANK1
*	WMSK3(23:16)	XX
**	WMSK2(15:8)	XXX, FASTCLEAR, RGBMODE, DBLSRC, PLANES(2)
***	WMSK1(7:0)	PLANES(1:0), DRAWDEPTH(1:0), LOGICOP(3:0)

Created:4/24/92 Modified: August 13, 1993 4:22 pm



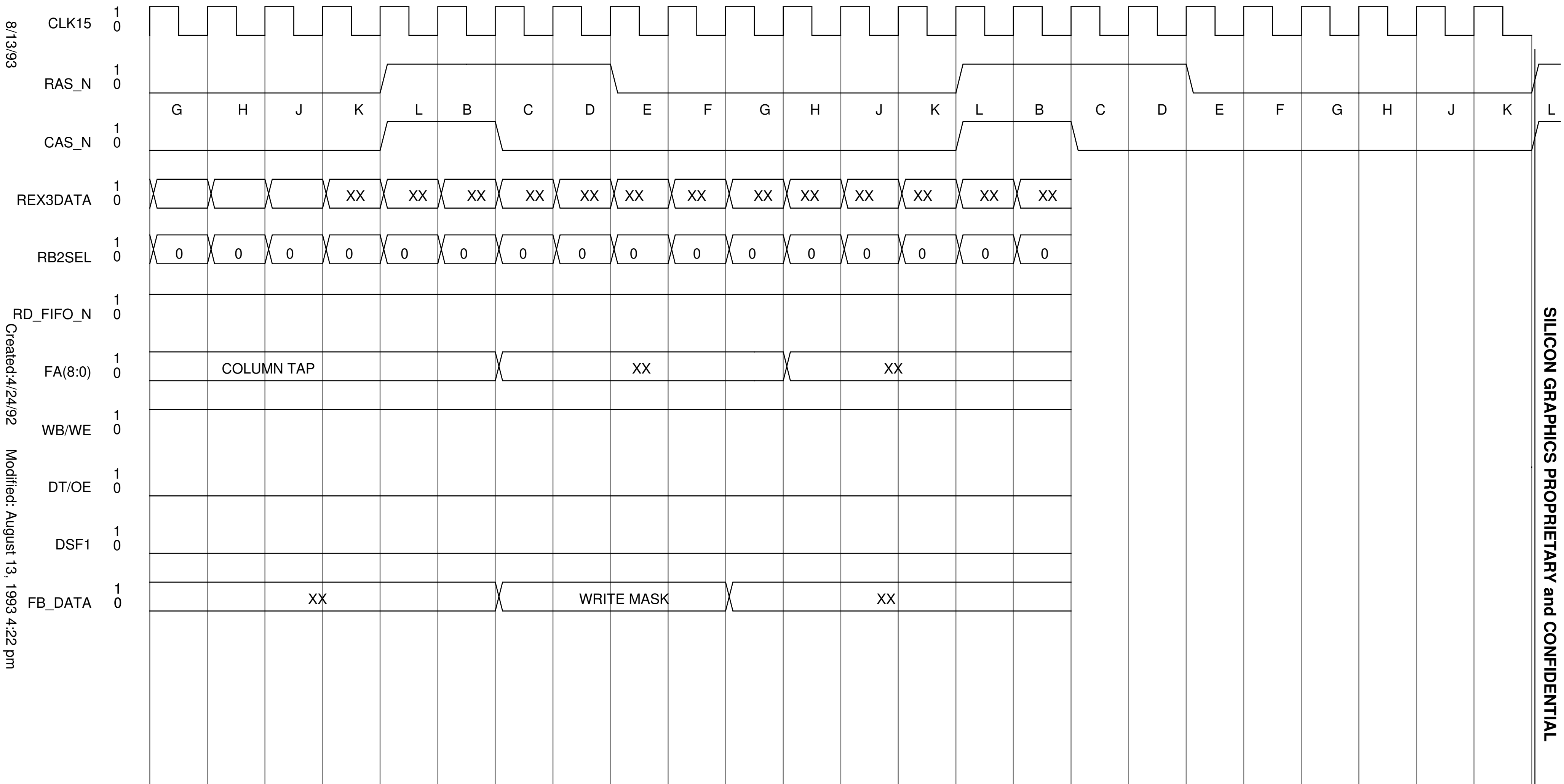
LOAD COLOR REGISTER.

Created:4/24/92 Modified: August 13, 1993 4:22 pm



READ TRANSFER CYCLE

Created:4/24/92
Modified: August 13, 1993 4:22 pm



CBRR CYCLE

Created:4/24/92

Modified: August 13, 1993 4:22 pm

4.4 Tester Interface

All bidirectional pins and all tri-stateable output pins are placed in their high impedance state when the TEI pin is driven low.

The JTAG_TMS pin, when driven low, enables the scan chain mux input into each storage element. When TP(1:0) = “11”, the JTAG_TCK pin is selected as the clock input to all REX3 flip-flops. JTAG_TDI is the scan input into the first storage element of the scan chain. Whenever TP(0) = “1”, the output of the last (4756th) element in the scan chain is muxed onto the JTAG_TDO pin. The first 229 elements in the scan chain are the flip-flops which drive REX3 bidirectional and output pins, and their output enables. The first 229 elements in the scan chain are:

RO_Y_DISP_0
RO_Y_DISP_1
BANK A TRI-STATE OE_N (’0’ enables outputs for the next 34 pins)
RB2_DATA_A_0_0
RB2_DATA_A_0_1
RB2_DATA_A_0_2
RB2_DATA_A_0_3
RB2_DATA_A_0_4
RB2_DATA_A_0_5
RB2_DATA_A_0_6
RB2_DATA_A_0_7
VRAM_WBWE_N_A
VRAM_DTOE_N_A
VRAM_DSF1_A
RB2_SEL_A_0
RB2_SEL_A_1
RB2_SEL_A_2
VRAM_ADDR_A_0
VRAM_ADDR_A_1
VRAM_ADDR_A_2
VRAM_ADDR_A_3
VRAM_ADDR_A_4
VRAM_ADDR_A_5
VRAM_ADDR_A_6
VRAM_ADDR_A_7
VRAM_ADDR_A_8
VRAM_RAS_A
VRAM_CAS_A_0
VRAM_CAS_A_1
RB2_DATA_A_1_0
RB2_DATA_A_1_1
RB2_DATA_A_1_2
RB2_DATA_A_1_3
RB2_DATA_A_1_4
RB2_DATA_A_1_5
RB2_DATA_A_1_6
RB2_DATA_A_1_7
BANK B TRI-STATE OE_N (’0’ enables outputs for the next 34 pins)
RB2_DATA_B_0_0
RB2_DATA_B_0_1
RB2_DATA_B_0_2
RB2_DATA_B_0_3
RB2_DATA_B_0_4
RB2_DATA_B_0_5
RB2_DATA_B_0_6
RB2_DATA_B_0_7
VRAM_WBWE_N_B
VRAM_DTOE_N_B
VRAM_DSF1_B
RB2_SEL_B_0
RB2_SEL_B_1
RB2_SEL_B_2
VRAM_ADDR_B_0
VRAM_ADDR_B_1
VRAM_ADDR_B_2
VRAM_ADDR_B_3
VRAM_ADDR_B_4
VRAM_ADDR_B_5
VRAM_ADDR_B_6
VRAM_ADDR_B_7
VRAM_ADDR_B_8
VRAM_RAS_B
VRAM_CAS_B_0
VRAM_CAS_B_1
RB2_DATA_B_1_0
RB2_DATA_B_1_1
RB2_DATA_B_1_2

RB2_DATA_B_1_3
RB2_DATA_B_1_4
RB2_DATA_B_1_5
RB2_DATA_B_1_6
RB2_DATA_B_1_7
BANK C TRI-STATE OE_N ('0' enables outputs for the next 34 pins)
RB2_DATA_C_0_0
RB2_DATA_C_0_1
RB2_DATA_C_0_2
RB2_DATA_C_0_3
RB2_DATA_C_0_4
RB2_DATA_C_0_5
RB2_DATA_C_0_6
RB2_DATA_C_0_7
VRAM_WBWE_N_C
VRAM_DTOE_N_C
VRAM_DSF1_C
RB2_SEL_C_0
RB2_SEL_C_1
RB2_SEL_C_2
VRAM_ADDR_C_0
VRAM_ADDR_C_1
VRAM_ADDR_C_2
VRAM_ADDR_C_3
VRAM_ADDR_C_4
VRAM_ADDR_C_5
VRAM_ADDR_C_6
VRAM_ADDR_C_7
VRAM_ADDR_C_8
VRAM_RAS_C
VRAM_CAS_C_0
VRAM_CAS_C_1
RB2_DATA_C_1_0
RB2_DATA_C_1_1
RB2_DATA_C_1_2
RB2_DATA_C_1_3
RB2_DATA_C_1_4
RB2_DATA_C_1_5
RB2_DATA_C_1_6
RB2_DATA_C_1_7
BANK D TRI-STATE OE_N ('0' enables outputs for the next 34 pins)
RB2_DATA_D_0_0
RB2_DATA_D_0_1
RB2_DATA_D_0_2
RB2_DATA_D_0_3
RB2_DATA_D_0_4
RB2_DATA_D_0_5
RB2_DATA_D_0_6
RB2_DATA_D_0_7
VRAM_WBWE_N_D
VRAM_DTOE_N_D
VRAM_DSF1_D
RB2_SEL_D_0
RB2_SEL_D_1
RB2_SEL_D_2
VRAM_ADDR_D_0
VRAM_ADDR_D_1
VRAM_ADDR_D_2
VRAM_ADDR_D_3
VRAM_ADDR_D_4
VRAM_ADDR_D_5
VRAM_ADDR_D_6
VRAM_ADDR_D_7
VRAM_ADDR_D_8
VRAM_RAS_D
VRAM_CAS_D_0
VRAM_CAS_D_1
RB2_DATA_D_1_0
RB2_DATA_D_1_1
RB2_DATA_D_1_2
RB2_DATA_D_1_3
RB2_DATA_D_1_4
RB2_DATA_D_1_5
RB2_DATA_D_1_6
RB2_DATA_D_1_7
DCB TRI-STATE OE_N ('0' enables outputs for the next 17 pins)
DCB_DATA_0
DCB_DATA_1
DCB_DATA_2
DCB_DATA_3
DCB_DATA_4
DCB_DATA_5
DCB_DATA_6
DCB_DATA_7
DCB_CRS_0

DCB_CRS_1
DCB_CRS_2
DCB_RW_N
DCB_CS_N
DCB_ADDR_0
DCB_ADDR_1
DCB_ADDR_2
DCB_ADDR_3
VR_INT_REG (if this bit is set, VV_INT_N will be asserted)
VIDEO_INT_D_REG (if this bit is set, VV_INT_N will be asserted)
FIFO_INT_N (if this bit is clear, FIFO_INT_N will be asserted)
GIO BUS TRI-STATE OE ('1' enables outputs for the next 65 pins)
P_AD_0
P_AD_1
P_AD_2
P_AD_3
P_AD_4
P_AD_5
P_AD_6
P_AD_7
P_AD_8
P_AD_9
P_AD_10
P_AD_11
P_AD_12
P_AD_13
P_AD_14
P_AD_15
P_AD_16
P_AD_17
P_AD_18
P_AD_19
P_AD_20
P_AD_21
P_AD_22
P_AD_23
P_AD_24
P_AD_25
P_AD_26
P_AD_27
P_AD_28
P_AD_29
P_AD_30
P_AD_31
P_GRXDLY
P_AD_32
P_AD_33
P_AD_34
P_AD_35
P_AD_36
P_AD_37
P_AD_38
P_AD_39
P_AD_40
P_AD_41
P_AD_42
P_AD_43
P_AD_44
P_AD_45
P_AD_46
P_AD_47
P_AD_48
P_AD_49
P_AD_50
P_AD_51
P_AD_52
P_AD_53
P_AD_54
P_AD_55
P_AD_56
P_AD_57
P_AD_58
P_AD_59
P_AD_60
P_AD_61
P_AD_62
P_AD_63

When TP(1:0) = "10", the parametric nand-tree/process monitor output is muxed onto JTAG_TDO. All bidirectional pins and all signal input pins (except for GIO64CLK, PLL_RESET_N, TP_0, and TP_1) are connected to the paremetric nand tree. VC_TX_REQ is the first pin in the nand tree, and the rest of the tree is connected in increasing LSI pin number order.

When TP(1:0) = “00”, the output of the VCO ripple counter is muxed onto JTAG_TDO, allowing testing of PLL VCO.Three pins are dedicated for scan chain based testing of the internal logic. SCAN_EN enables the scan chain mux input into each storage element. SCAN_IN feeds the first storage element in the scan chain. SCAN_OUT, which brings the scan chain off chip. The on chip Phase Lock Loop requires 5 pins for testing: PLL_TSTMD, which places the PLL into Test Mode, PLL_TCLK, the PLL test mode clock input, PTREE_PLLTCKO, the PLL test mode clock output.

5 Architectural Description

5.1 GIO64 Bus Interface

The REX3 is a slave device on the pipelined GIO64 bus. The GIO64 bus interface is the functional block of the REX3 that responds to data transfer requests from a GIO64 bus master. Commands and data are sent to the graphics pipeline through the graphics fifo (GFIFO), commands and data to the graphics back-end devices (VC2, XMAP, CMAP, RAMDAC, Video Option) are sent to the Display Control Bus Interface through the backend fifo (BFIFO). A block diagram of the GIO64 bus interface follows.

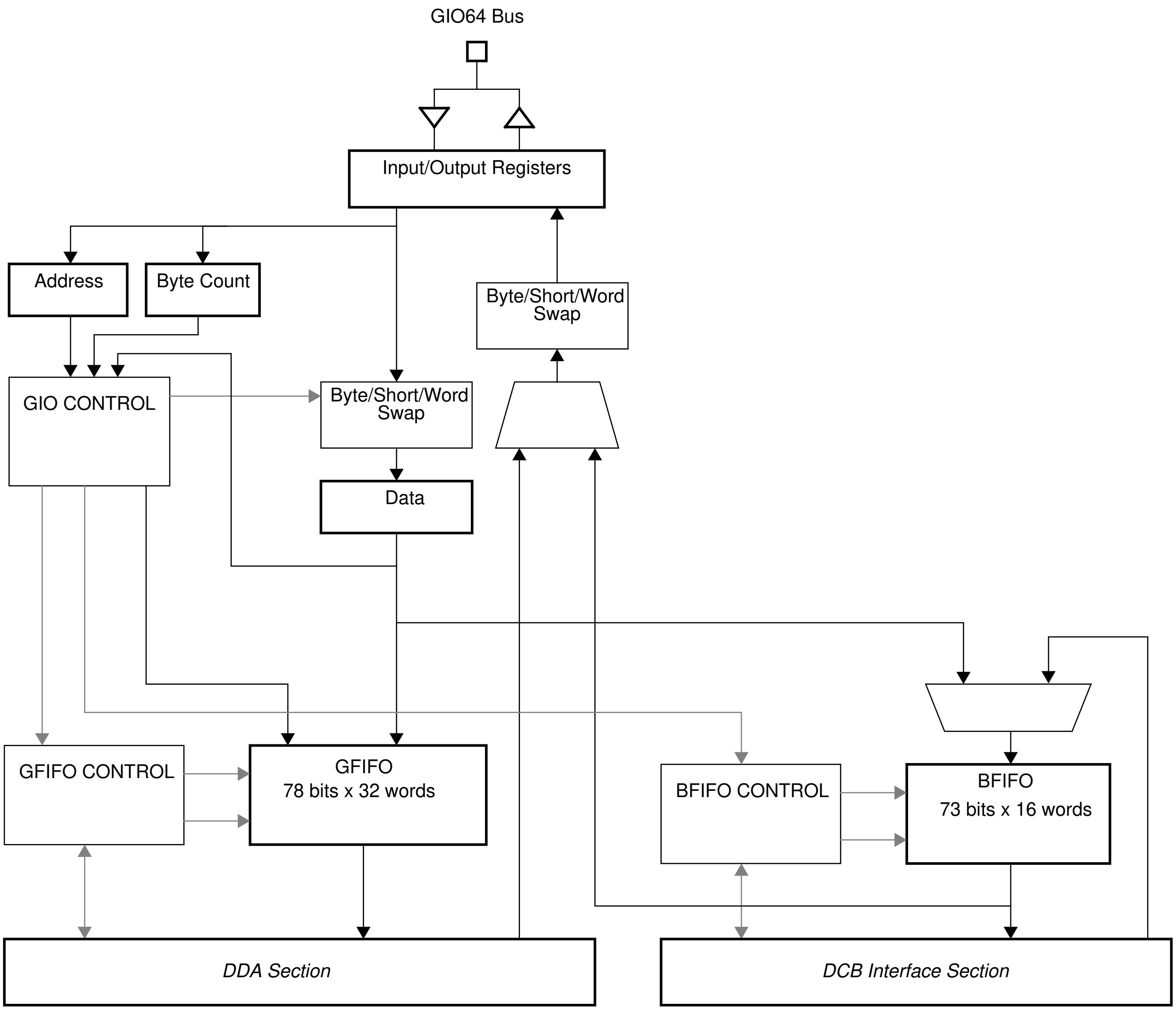


FIGURE 5. GIO64 Bus Interface Block Diagram

5.2 Display Control Bus Interface

The Display Control Bus Interface section of the REX3 unpacks DCBMODE and DCBDATA information from the BFIFO, and uses this information to execute data transfers on the Display Control Bus (DCB). When DCBMODE data is unpacked from the BFIFO, the operating mode (DCB protocol and timing, DCB slave device address, DCB slave register address) of the DCB state machine is defined. Data written by the host to the DCBDATA register is unpacked from the BFIFO, and sent out on the DCB by the DCB state machine, using the defined DCBMODE. When the host performs a read of the DCBDATA register, a DCB read request is pushed onto the BFIFO by the GIO interface. When the DCB read request is unpacked from the BFIFO, the DCB state machine will execute the read data transfer cycles on the DCB, packing the data it receives into the BFIFO. The GIO interface will then transfer the requested data from the BFIFO back to the host. A block diagram of the Display Control Bus Interface follows:

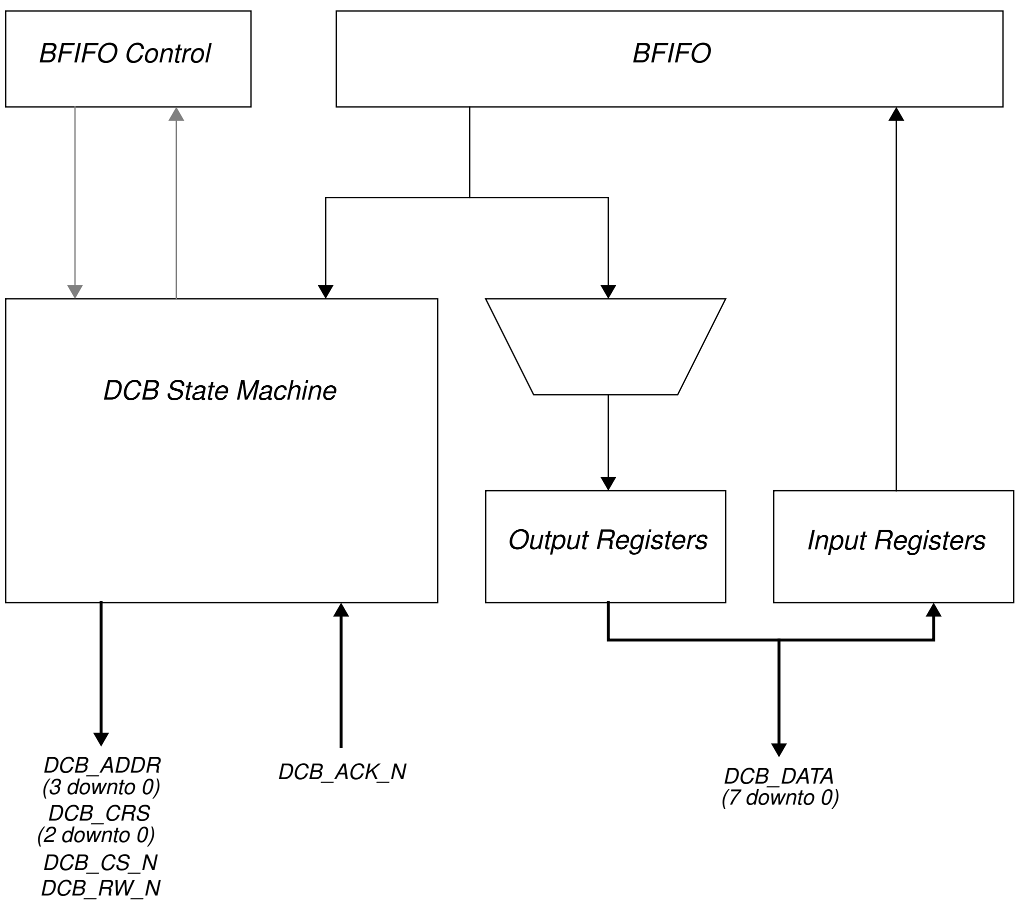


FIGURE 6. Display Control Bus Interface Block Diagram

5.3 DDA Unit

The DDA unit assembles the drawing context, by unloading commands and data from GFIFO, and executes drawing primitives. The main components of the DDA unit are the current drawing context registers, the 3-stage shade and address generation pipelines, and the host pixel swizzle and pack logic.

The top-level VHDL block is named DDA_TOP. Figure 7 shows the major functional blocks.

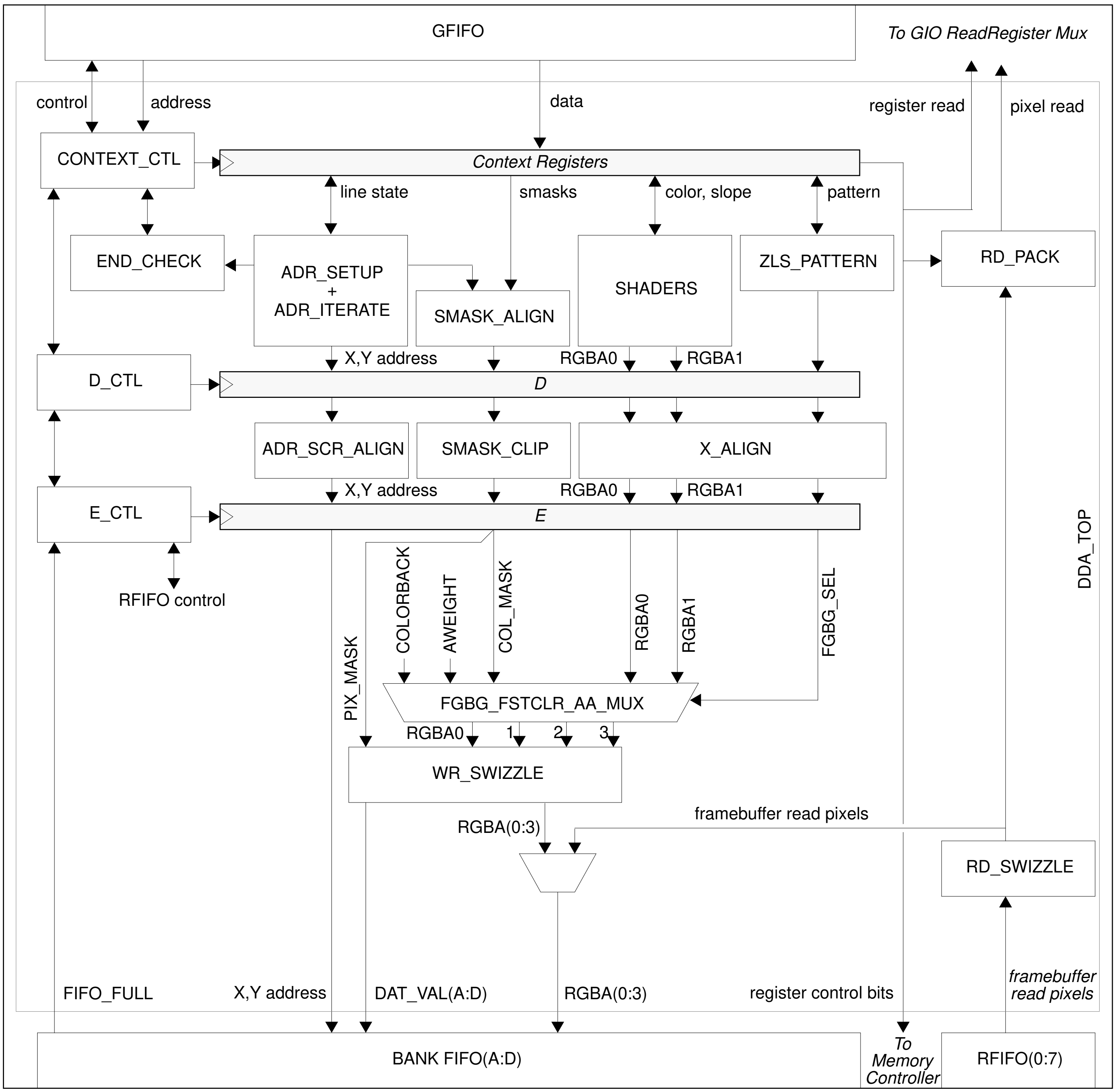


FIGURE 7. DDA_TOP block diagram.

5.3.1 DDA_TOP Port List

Figure 8 shows the port diagram for DDA_TOP. There are six functional interfaces: (1) The GFIFO section unloads GFIFO data and control into DDA_TOP next context; (2) The VRAM BANK FIFO interface loads the memory subsection BANKFIFOs with pixel address, data, and control bits; (3) VRAM CTL routes control bits from DDA_TOP non-pipelined registers to the memory controller; (4) VRAM READ FIFO consists of DDA_TOP RFIFO data input and handshake; (5) READ DATA routes DDA_TOP packed GIO read data to the GIO interface; (6) The STATUS signals communicate idle status between blocks. Table 26 outlines the function of DDA_TOP ports.

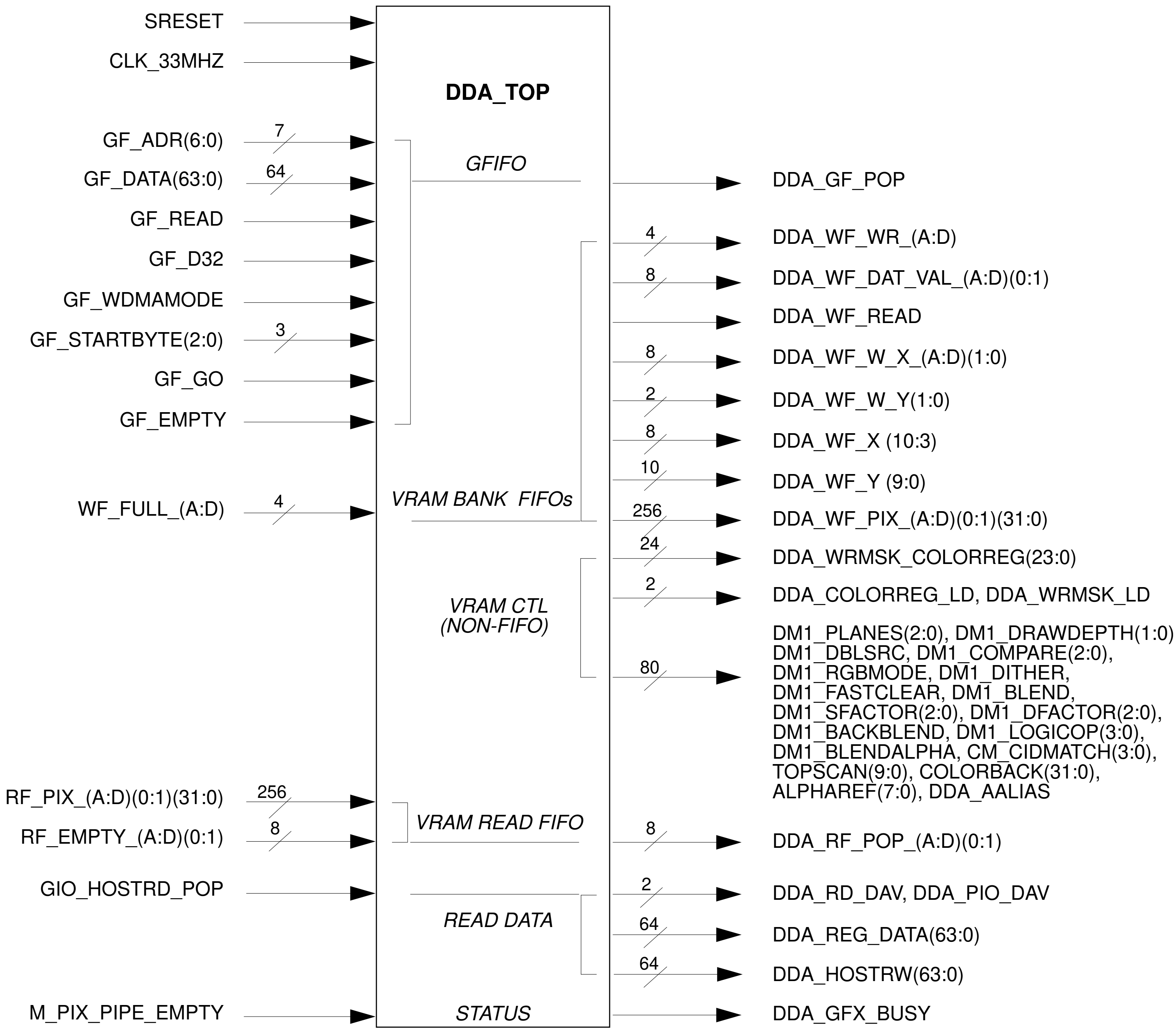


FIGURE 8. DDA_TOP port diagram.

.

Signal Name	Type	Active	Description
SRESET	I	H	Global synchronous reset.
CLK_33MHZ	I		33MHz chip clock (GIO clock).
GF_ADR(6:0)	I		GFIFO GIO address bits used for register decode: GIO ADR(9:8,6:2).
GF_DATA(63:0)	I		GFIFO write data.
GF_READ	I	H	GFIFO read/write flag.
GF_D32	I	H	GFIFO data width: 1=32-bit, 0=64-bit transfer, (from GIO BC/SB.)
GF_WDMAMODE	I	H	GFIFO DMA mode enable.
GF_STARTBYTE(2:0)	I		GFIFO GIO bus START BYTE field.
GF_GO	I	H	GFIFO GO bit, decoded from GIO address.
GF_EMPTY	I	H	GFIFO empty flag.
DDA_GF_POP	O	H	GFIFO read strobe.
WF_FULL_(A:D)	I	H	Bank FIFO full flags.
DDA_WF_WR_(A:D)	O	H	Bank FIFO write strobes.
DDA_WF_DAT_VAL_(A:D)(1:0)	O	H	Bank FIFO individual pixel valid flags, two per bank.
DDA_WF_READ	O	H	Bank FIFO read flag.
DDA_WF_W_X_(A:D)(1:0)	O		Bank FIFO window relative X address lsbs, for dithering.
DDA_WF_W_Y_(1:0)	O		Bank FIFO window relative Y address lsbs, for dithering.
DDA_WF_X(10:3)	O		Bank FIFO screen relative X address.
DDA_WF_Y(9:0)	O		Bank FIFO screen relative Y address.
DDA_WF_PIX_(A:D)(0:1)(31:0)	O		Bank FIFO RGBA pixel data (8-8-8-8).
DDA_WRMSK_COLORREG(23:0)	O		Muxed pixel writemask and VRAM color register data.
DDA_COLORREG_LD	O	H	VRAM color register write strobe.
DDA_WRMSK_LD	O	H	VRAM writemask write strobe.
DM1_PLANES(1:0)	O		DRAWMODE1 VRAM planes enabled for R/W access.
DM1_DRAWDEPTH(1:0)	O		DRAWMODE drawn pixel depth.
DM1_DBLSRC	O		DRAWMODE1 double-buffer mode pixel read source buffer select.
DM1_COMPARE(2:0)	O		DRAWMODE1 condition specifier for color compare function.
DM1_RGBMODE	O	H	DRAWMODE1 RGB (vs. color index) enable.
DM1_DITHER	O	H	DRAWMODE1 dither enable.
DM1_FASTCLEAR	O	H	DRAWMODE1 pixel FASTCLEAR write mode enable.
DM1_BLEND	O	H	DRAWMODE1 blendfunction enable.
DM1_SFACTOR(2:0)	O		DRAWMODE1 source blending factor.
DM1_DFACTOR(2:0)	O		DRAWMODE1 destination blending factor.
DM1_BACKBLEND	O	H	DRAWMODE1 COLORBACK destination blend enable.
DM1_BLENDALPHA	O	H	DRAWMODE1 source alpha/1.0 blendfunction select for source alpha
DM1_LOGIC_OP(3:0)	O		DRAWMODE1 logical operation type.
CM_CIDMATCH(3:0)	O		CLIPMODE CID check compare code.

Table 26: DDA_TOP port descriptions.

Signal Name	Type	Active	Description
TOPSCAN(9:0)	O		Y address for top of screen scan line.
COLORBACK(31:0)	O		Destination blend color when DM1_BACKBLEND=1.
DDA_AALIAS	O	H	Enables anti-alias mode (from DRAWMODE0 OPCODE).
RF_PIX_(A:D)_(0:1)(31:0)	I		RFIFO data output (framebuffer read).
RF_EMPTY_(A:D)(0:1)	I	H	RFIFO empty flags.
DDA_RF_POP_(A:D)(0:1)	O	H	RFIFO write strobes.
GIO_HOSTRD_POP	I	H	GIO read acknowledge strobe.
DDA_RD_DAV	O	H	GIO register/DMA read data available flag.
DDA_PIO_DAV	O	H	GIO PIO read data available flag.
DDA_REG_DATA(63:0)	O		Context register read bus.
DDA_HOSTRW(63:0)	O		HOSTRW read bus.
M_PIX_PIPE_EMPTY	I	H	VRAM controller pixel pipe idle flag.
DDA_GFX_BUSY	O	H	Graphics pipeline idle: M_PIX_PIPE_MT & DDA pipeline idle.

Table 26:   DDA_TOP port descriptions.

5.4 VRAM Controller Unit

The frame buffer controller is made up of four independent memory controllers (A thru D).Each memory controller controls two sub banks (A0, A1, B0....etc.). Following is a description of how the banks are interleaved.

5.4.1 Vram Interleave

The frame buffer is made up of 8 Vrams for the base system and 24 Vrams for the upgraded system. Each of the four banks has two sub banks that interface to 1 to 3 Vrams data port. The address is shared between two sub banks as is RAS, WB/WE and DT/OE. Each sub bank has its own CAS signal. Since the frame buffer supports 2 MBit Vrams, (512 x 512 x 8 array) there are two scan lines in each row of the Vram. Each sub bank holds the adjacent pixel along a scan line. In order to achieve a high writing rate for line drawing, the frame buffer has been scrambled as shown in Table 28. The left two columns show the Vram page number and the scan line number. the rest of the table shows which pixels are affected by which bank. Figure 6 shows the X vs. Y interleave of the frame buffer. The scan line packing is shown in Table 32. This interleaving format (making the frame buffer isotropic) lends itself to fast writing rates. e.g. for vertical lines, two pixels reside in the same Vram page and sub-bank, therefore, between all four banks REX3 can write 8 pixels in one full memory write cycle time + one page mode cycle time. All four banks write the two pixels in parallel. For lines of slope +/- 1, REX3 writes 8 adjacent pixels, one in each bank. For horizontal lines every other bank is accessed in parallel and in page mode. For spans, all banks are accessed in parallel and in page mode. This holds true when drawing lines and spans in reverse direction (i.e L-R Vs. R-L etc.)as well.

Table 27: Frame buffer format

Page 3	S-line7	D0	D1	A0	A1	B0	B1	C0	C1
Page 2	S-line 6	C0	C1	D0	D1	A0	A1	B0	B1
Page 1	S-line 5	B0	B1	C0	C1	D0	D1	A0	A1
Page 0	S-line 4	A0	A1	B0	B1	C0	C1	D0	D1
Page 3	S-line 3	D0	D1	A0	A1	B0	B1	C0	C1
Page 2	S-line 2	C0	C1	D0	D1	A0	A1	B0	B1
Page 1	S-line 1	B0	B1	C0	C1	D0	D1	A0	A1
Page 0	S-line 0	A0	A1	B0	B1	C0	C1	D0	D1
	Interleave	0	1	2	3	4	5	6	7

5.4.1.1 Aux/Pixel plane Interleave

In addition to the frame buffer interleave shown in Table 1, the AUX planes are also interleaved with the pixel planes as shown in Table 2. Starting from column address 0, the first four bytes are AUX planes followed by 8 bytes of pixel planes. There are two 2 AUX values per byte. Due to the 8 way interleave, the two AUX's are for pixel 0 and pixel 8 in the first byte of the row 0, etc. All operations on the AUX planes are read modify write operations. This is due to the fact that not all operations will be MOD2 aligned in the X-axis and there is considerable overhead in performing multiple write mask operations for end point conditions. Only one of the AUX planes (CID, OLY or PUP) planes can be written at any time.

5.4.1.2 Writemask

software has to be aware of the frame buffer formats and set the appropriate write mask pattern. In the case of writting one of the AUX planes, software does not have to worry about the pixel packing (i.e two pixels residing in one memory location) since the memory controller performs a read on both the pixels and modifies only the necessary one while keeping the other pixel in an unmodified state and then writes both the pixels back.

5.4.2 Vram address generation

The frame buffer is architected as 4 way interleave memory. Each interleave has two subbanks. The multiplexed Vram address is shared between the two banks. Since the pixel and aux planes are interleaved in the Vram, the column address for the aux planes is different from the pixel planes. For any pixel, the corresponding aux planes are in the same Vram and same page. Only the column address varies. The address calculation for pixel and aux as implemented in hardware is shown below.

5.4.2.1 Pixel planes column address

Y2 && [((X10 - X3)DIV8 x 12) + 4 + (X10 - X3)MOD8]

5.4.2.2 Aux planes column address

Y2 && [((X10 - X3)DIV8 x 12) + (X10 - X4)MOD4]

5.4.2.3 Row address

(Y9 - Y3) && (Y1 - Y0)

Table 27: Aux/pixel planes interleave

BANK A		BANK B		BANK C		BANK D	
0	1	0	1	0	1	0	1
VRAM 0	VRAM 1	VRAM 2	VRAM 3	VRAM 4	VRAM 5	VRAM 6	VRAM 7
PUP-CID 0 PUP-CID 8	PUP-CID 1 PUP-CID 9	PUP-CID 2 PUP-CID 10	PUP-CID 3 PUP-CID 11	PUP-CID 4 PUP-CID 12	PUP-CID 5 PUP-CID 13	PUP-CID 6 PUP-CID 14	PUP-CID 7 PUP-CID 15
PUP-CID 16 PUP-CID 24	PUP-CID 17 PUP-CID 25	PUP-CID 18 PUP-CID 26	PUP-CID 19 PUP-CID 27	PUP-CID 20 PUP-CID 28	PUP-CID 21 PUP-CID 29	PUP-CID 22 PUP-CID 30	PUP-CID 24 PUP-CID 31
PUP-CID 32 PUP-CID 40	PUP-CID 33 PUP-CID 41	PUP-CID 34 PUP-CID 42	PUP-CID 35 PUP-CID 43	PUP-CID 36 PUP-CID 44	PUP-CID 37 PUP-CID 45	PUP-CID 38 PUP-CID 46	PUP-CID 39 PUP-CID 47
PUP-CID 48 PUP-CID 56	PUP-CID 49 PUP-CID 57	PUP-CID 50 PUP-CID 58	PUP-CID 51 PUP-CID 59	PUP-CID 52 PUP-CID 60	PUP-CID 53 PUP-CID 61	PUP-CID 54 PUP-CID 62	PUP-CID 55 PUP-CID 63
PIXEL 0	PIXEL 1	PIXEL 2	PIXEL 3	PIXEL 4	PIXEL 5	PIXEL 6	PIXEL 7
PIXEL 8	PIXEL 9	PIXEL 10	PIXEL 11	PIXEL 12	PIXEL 13	PIXEL 14	PIXEL 15
PIXEL 16	PIXEL 17	PIXEL 18	PIXEL 19	PIXEL 20	PIXEL 21	PIXEL 22	PIXEL 23
PIXEL 24	PIXEL 25	PIXEL 26	PIXEL 27	PIXEL 28	PIXEL 29	PIXEL 30	PIXEL 31
PIXEL 32	PIXEL 33	PIXEL 34	PIXEL 35	PIXEL 36	PIXEL 37	PIXEL 38	PIXEL 39
PIXEL 40	PIXEL 41	PIXEL 42	PIXEL 43	PIXEL 44	PIXEL 45	PIXEL 46	PIXEL 47
PIXEL 48	PIXEL 49	PIXEL 50	PIXEL 51	PIXEL 52	PIXEL 53	PIXEL 54	PIXEL 55
PIXEL 56	PIXEL 57	PIXEL 58	PIXEL 59	PIXEL 60	PIXEL 61	PIXEL 62	PIXEL 63

Y MOD 4 \ X MOD 8	BANKA	BANKB	BANKC	BANKD
0	0,1	2,3	4,5	6,7
1	2,3	4,5	6,7	0,1
2	4,5	6,7	0,1	2,3
3	6,7	0,1	2,3	4,5

FIGURE 9. X,Y Interleave for the frame buffer

Table 28: Frame buffer scanline interleave.

511

SCAN LINE 1019	OSM	SCAN LINE 1023	OSM
ETC		ETC	
SCAN LINE 11	OSM	SCAN LINE 15	OSM
SCAN LINE 10	OSM	SCAN LINE 14	OSM
SCAN LINE 9	OSM	SCAN LINE 13	OSM
SCAN LINE 8	OSM	SCAN LINE 12	OSM
SCAN LINE 3	OSM	SCAN LINE 7	OSM
SCAN LINE 2	OSM	SCAN LINE 6	OSM
SCAN LINE 1	OSM	SCAN LINE 5	OSM
SCAN LINE 0	OSM	SCAN LINE 4	OSM

0

0   22   22   44   5
34   55   99   1
90   56   56   1

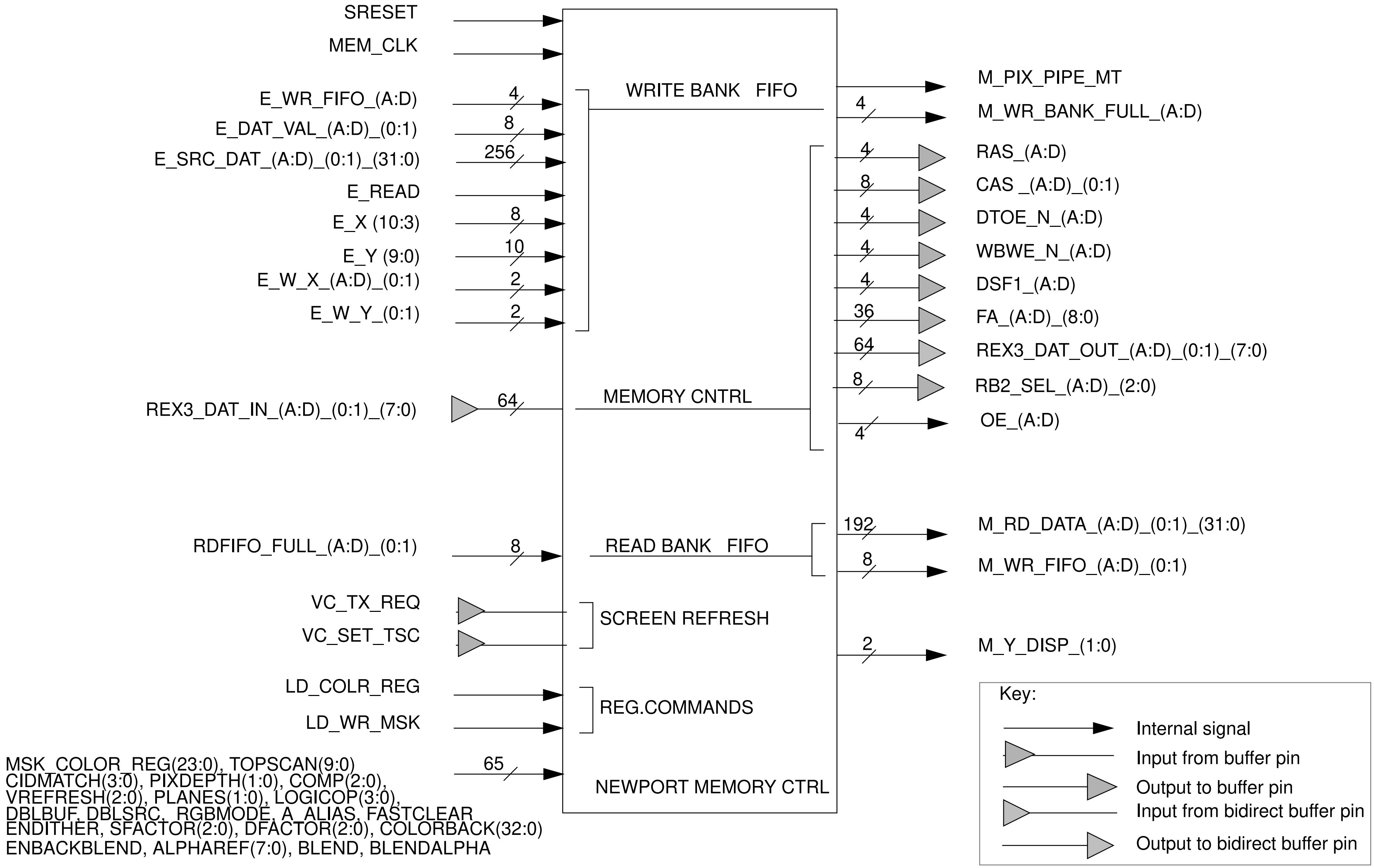


FIGURE 10. REX3 Memory Controller and Pixel Pipe Top-Level Block Diagram

Table 29: NEWPORT MEMORY CTRL PIN DESCRIPTION

Signal	Description	Type	Timing
	WRITE BANK FIFO		
E_WR_FIFO_(A:D)	Write command for bank fifos	I	33MHz
E_DAT_VAL_(A:D)_(0:1)	Data valid in bank fifo	I	33MHz
E_SRC_DAT_(A:D)_(0:1) _(31:0)	Source data written into bank fifos RGBA 8 Bits each.	I	33MHz
E_READ	Pixel read or write command	I	33MHz
E_X(10:3)	Screen X coordinates DIV8)	I	33MHz
E_Y(9:0)	Screen Y coordinates	I	33MHz
E_W_X_(A:D)_(0:1)	Window relative X bits for dithering	I	33MHz
E_W_Y_(0:1)	Window relative Y bits for dithering	I	33MHz
RDFIFO_FULL_(A:D)_(0: 1)	Read fifo full indicator	I	
M_WR_BANK_FULL_(A: D)	Write bank fifo full	O	33MHz
	GLOBAL SIGNALS		
SRESET	Synchronous reset for disabling drivers and test	I	Static/33
A_ALIAS	Anti alias bit	I	Static/33
ENDITHER	Enable dither	I	Static/33
VC_TX_REQ	Transfer request from VC	I	Async
VC_SET_TSC	Set transfer line # to TOPSCAN Reg.	I	Async
LD_COLR_REG	Load color reg. in Vram command (pulse) for block mode	I	33MHz
LD_WR_MSK	Load write mask reg in RB2 command (pulse)	I	33MHz
MSK_COLOR_REG(23:0)	Write mask/color reg data for LD_COLR_REG or LD_WR_MSK commands	I	Static/33
TOPSCAN(9:0)	First scanline for screen refresh (top of screen)	I	Static/33
CIDMATCH(3:0)	4bits. One of four CID values to match.	I	Static/33
PIXDEPTH(1:0)	4, 8, 12, or 24 bits/pixel	I	Static/33
COMP(2:0)	Color compare function If “111” => color comp disabled	I	Static/33
VREFRESH(2:0)	# of memory refreshes to follow a vram transfer cycle	I	Static/33
PLANES(1:0)	Planes to access	I	Static/33
SFACTOR(2:0)	Source factor for blend function	I	Static/33
DFACTOR(2:0)	Destination factor for blend operation	I	Static/33
COLORBACK(32:0)	Background color to be blended for textures	I	Static/33
ENBACKBLEND	Enable background color blend	I	Static/33
ALPHAREF(7:0)	Reference alpha for AFUNCTION test	I	Static/33
BLEND	Enable the blend function	i	Static/33
BLENDALPHA	Blend source alpha with alpha	i	Static/33
LOGIC_OP(3:0)	Logic operations to be performed on src/dst pixels	I	Static/33
DBLBUF	Double buffer mode	I	Static/33
DBLSRC	Read source buffer for double buffer mode	I	Static/33
FGASTCLEAR	Vram blockfill mode (for writing color reg into frame buffer	I	Static/33
RGBMODE	RGB mode Vs CI	I	Static/33
M_PIX_PIPE_MT	Pixel pipe empty	O	66MHz
M_Y_DISP(1:0)	LSB’s of screen refresh to XMAP9 for frame buffer descramble	O	66MHz
	FRAME BUFFER SIGNALS		
REX3_DAT_IN_(A:D)_(0:1)_(7:0)	Frame buffer read data (7:0) via RB2	I	66MHz
RAS_(A:D	Vram Row Address Strobe	O	66MHz
CAS _(A:D)_(0:1)	Vram Column Address Strobe	O	66MHz
DTOE_N__(A:D)	Vram transfer/ output enable signal	O	66MHz
WBWE_N__(A:D)	Vram write control signal	O	66MHz
DSF1_(A:D)	Vram special function signal	O	66MHz
FA)_(A:D)_(8:0)	Vram multiplexed address	O	66MHz
REX3_DAT_OUT_(A:D)_(0:1)_(7:0)	Frame buffer write dat(7:0) via RB2	O	66MHz

Table 29: NEWPORT MEMORY CTRL PIN DESCRIPTION

Signal	Description	Type	Timing
RB2_SEL_(A:D)_(2:0))	Byte select for RB2	O	66MHz
OE_(A:D)	Output enable for bidirect data drivers	O	66MHz
	READ BANK FIFO SIGNALS		
M_RD_DATA_(A:D)_(0:1)_(31:0)	Frame buffer read data (31:0) to read fifo's	O	66MHz
M_WR_FIFO_(A:D)_(0:1)	Write command for M_RD_DATA	O	66MHz
RDFIFO_FULL_(A:D)_(0:1)	Read fifo full (one per bank)	I	66MHz

5.3.2 Memory Controller

The memory controller for Newport is implemented as 4 independent bank controllers. The 4 controllers are identical to each other and operate independently from each other. Figure 1. shows the top-level of the frame buffer controller. The frame buffer controller includes all the dither, logic-op, blend, cid check, Afunction test, color compare, read format and write format functions. There is one copy of each function in each sub bank except for the blend unit which is shared between to sub banks.The pin description of the controllers (A thru D) is shown in Table 1. Each bank has its own data and address port as well as various control signals for RB2 and Vrams. The write bank fifos are incorporated into each bank while the read fifos reside in the DDA section.

Each bank has two sub banks (0 and 1) so for bank A references to the sub banks will be made as A0 and A1. The address port is shared between the sub banks of a bank while they have their own 8 bit bidirectional data ports interfacing to RB2's. The data ports serially send out 3 bytes of to assemble a 24 bit pixel in RB2 which interfaces to frame buffer. Reads from the frame buffer are serialized by RB2 and read on the 8 bit data port of REX3. The byte number to read/write to RB2 is selected by RB2_SEL(0:1) generated from REX3.

The memory controller of each bank operates the sub banks in parallel. Each bank has two valid bits so that the pixel write can be negated at the last moment. There are 6 functional state machine units in each bank, and one common general purpose unit (called general mc decode) which is common to all 4 banks. The general mc unit provides various decodes and also synchronized the screen transfer requests from VC2. The 6 state machine units in each bank are as follows:

a. CONTROL MODULE
b. RMW_FSM
c. WRITE_FSM
d. LD_REG_FSM
e. TR_FSM
f. OUT_BLOCK

Figure 3. shows the connection of the various state machines. The Control Module is responsible for enabling the various state machines. With the exception of CONTROL MODULE and OUT_BLOCK, only one state machine is active at any time.

The address pipeline is shown in Figure 2. Page comparison of the previous pixel address and the current pixel address is done at the first pipeline stage. If a page miss is encountered, further reading of the write bank fifo is inhibited and the previous two pixels are written before generating a precharge cycle for the Vrams.

Non-persistent write mask feature of the Vrams has been used. The write mask is loaded into RB2 by asserting RB2_LDWMASK and the appropriate byte select # on the RB2_SEL(0:1) lines. RB2 detects RAS being at a logic high and asserts the write mask value onto the frame buffer data bus. Each bank has a common RAS signal for the sub banks and two CAS signals, one for each sub bank. For lines, only one of the sub banks is operated on whereas for spans both banks are operated on simultaneously and their CAS signals are synchronous if Xaddress MOD2 is zero. For write only cycles, the controller performs early write cycles and for read modify write cycles, late write cycles are performed.The memory controller operates at 66MHz. Full page cycle takes 10 clks (150 nS) so 80nS or faster Vrams are required. The color reg in the Vrams is loaded by the memory controller. Block mode write cycle feature of the

Vrams is available for screen clear operations. In block mode, each bank can write 8 pixels in 4 clocks (60nS).

Following is the pin description of each state machine unit and their respective state diagrams.

5.3.3 VRAMS

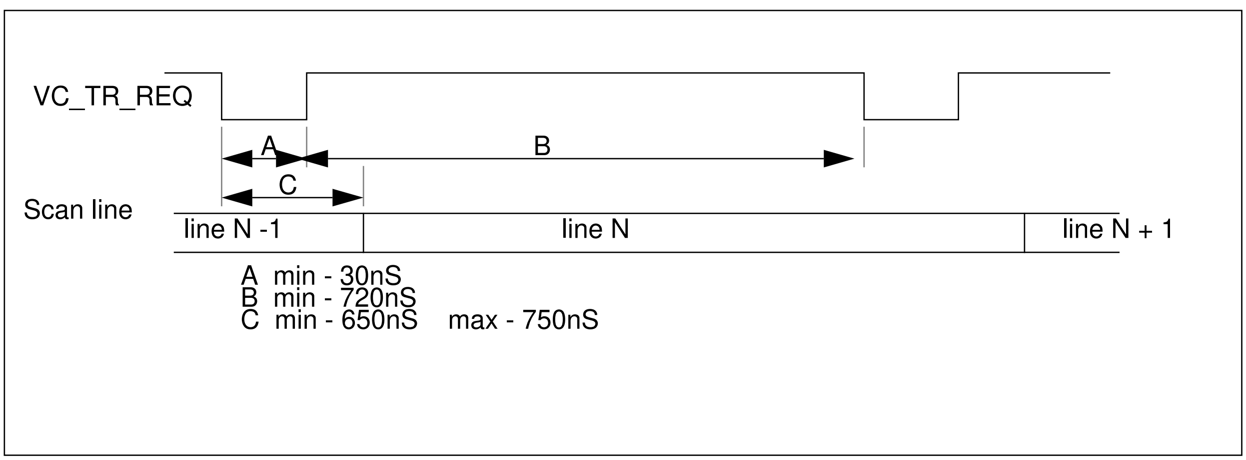
REX3 has been designed to work with VRAMS from the following vendors:

Toshiba	TC528257
Mitsubishi	M5M482256
Hitachi	HM538253
Micron	MT42C8255
Fujitsu	MB8128xx
NEC	uPD482234
TI	TMX55160
Vitelic	V53C851
Samsung	KM428C256

All of the above have to be 70nS or faster.

5.4 Scan Refresh Latency

Scan refresh is initiated by asserting VC_TR_REQ signal. A falling edge is detected on this signal which triggers a 480nS timer in REX3. Once the timer has timed out, if there are no more pixels in the pipe then the transfer state machine is invoked. When a falling edge on VC_TR_REQ signal is detected the display line number (initialized from VREFRESH reg.) first increments and then does a Vram serial read transfer cycle. During the timeout period another falling edge on VC_TR_REQ signal may be generated (doing so will not restart the 480nS timer) to increment the display line number, hence generating interlaced mode. The minimum time to realize a new scan line in Vram after asserting VC_TR_REQ is 650nS and the maximum time is 750nS. The timing constraints are shown in the diagram below.



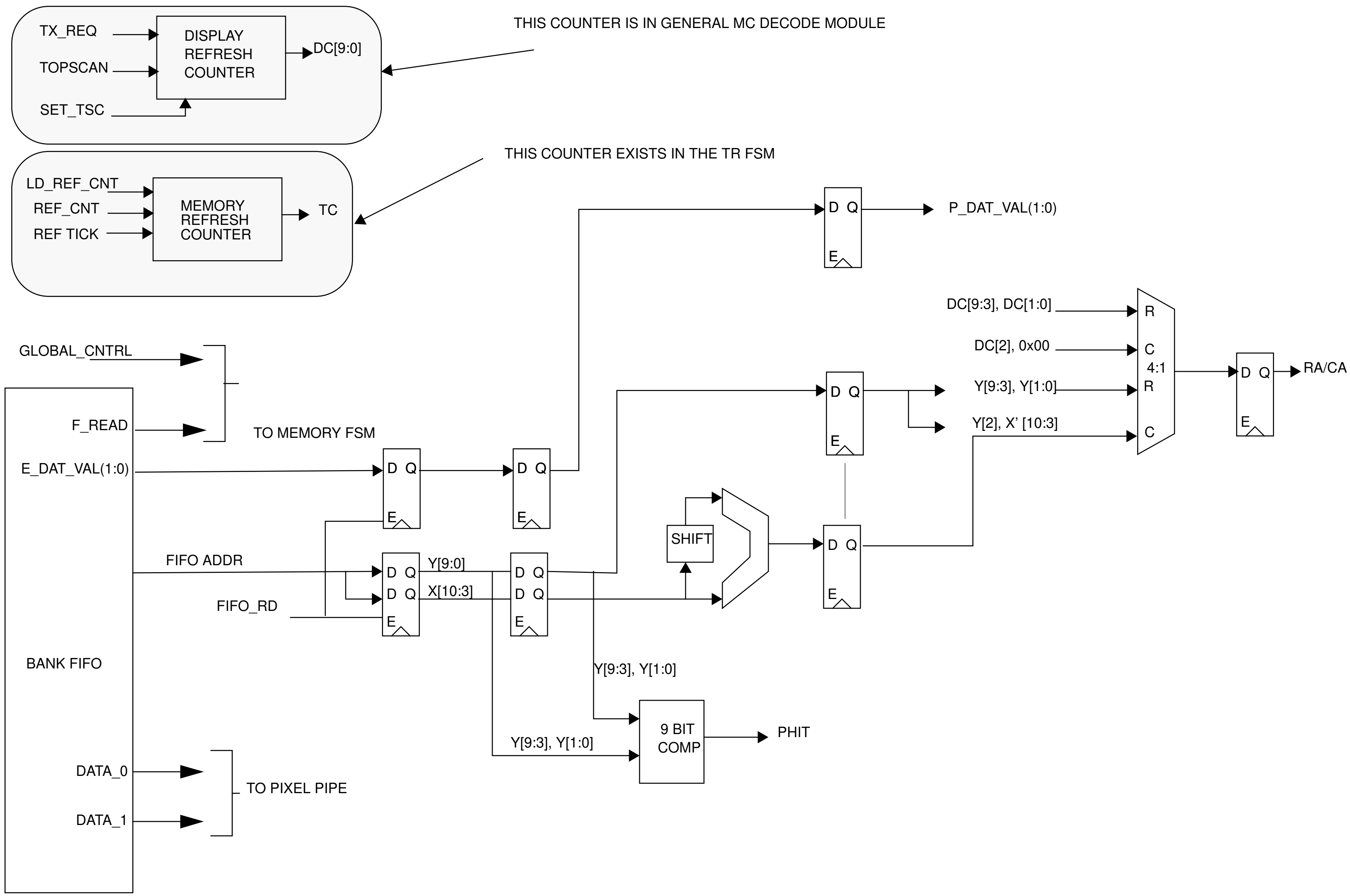


FIGURE 11. Newport graphics memory controller address pipeline for one bank

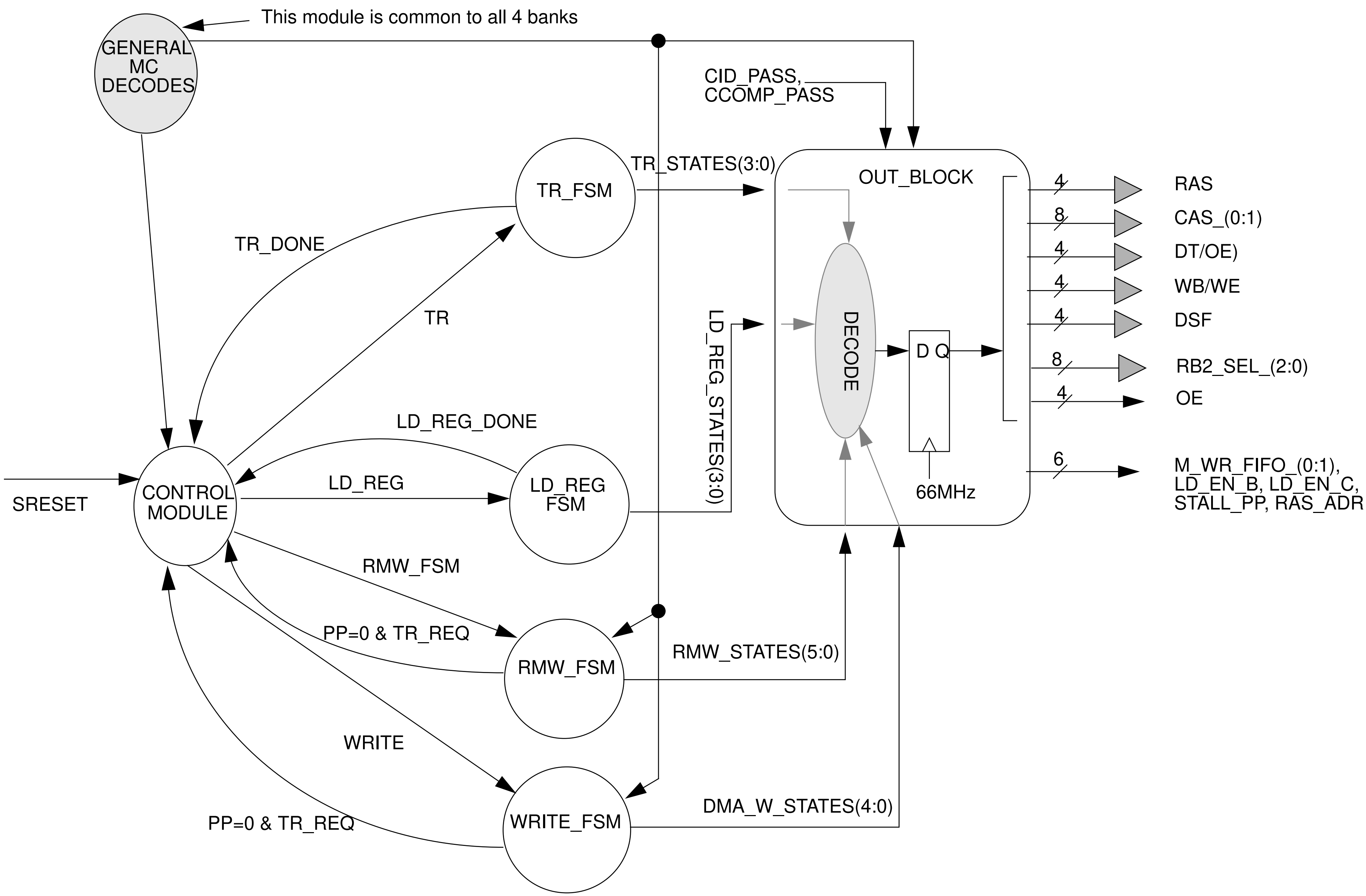
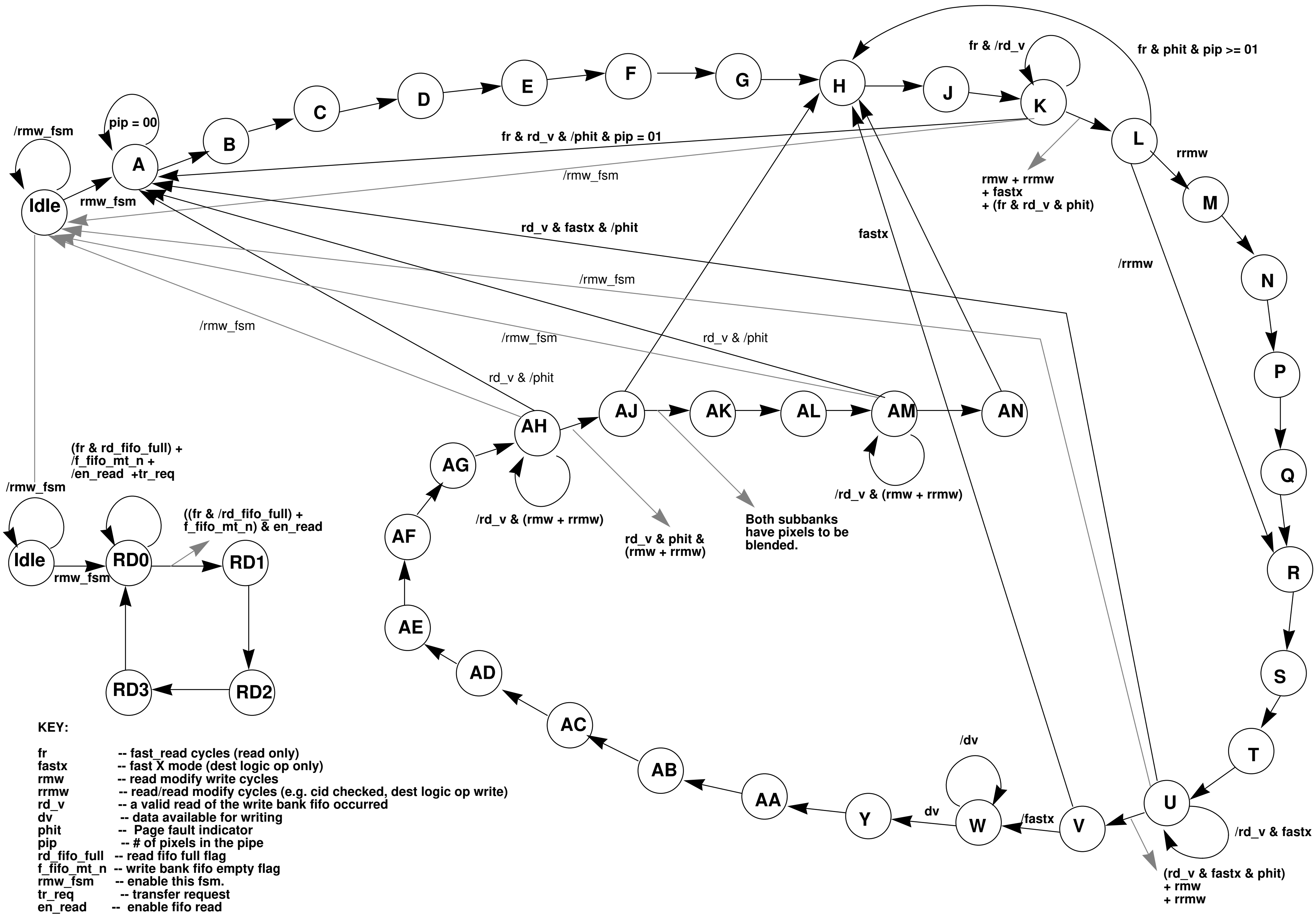


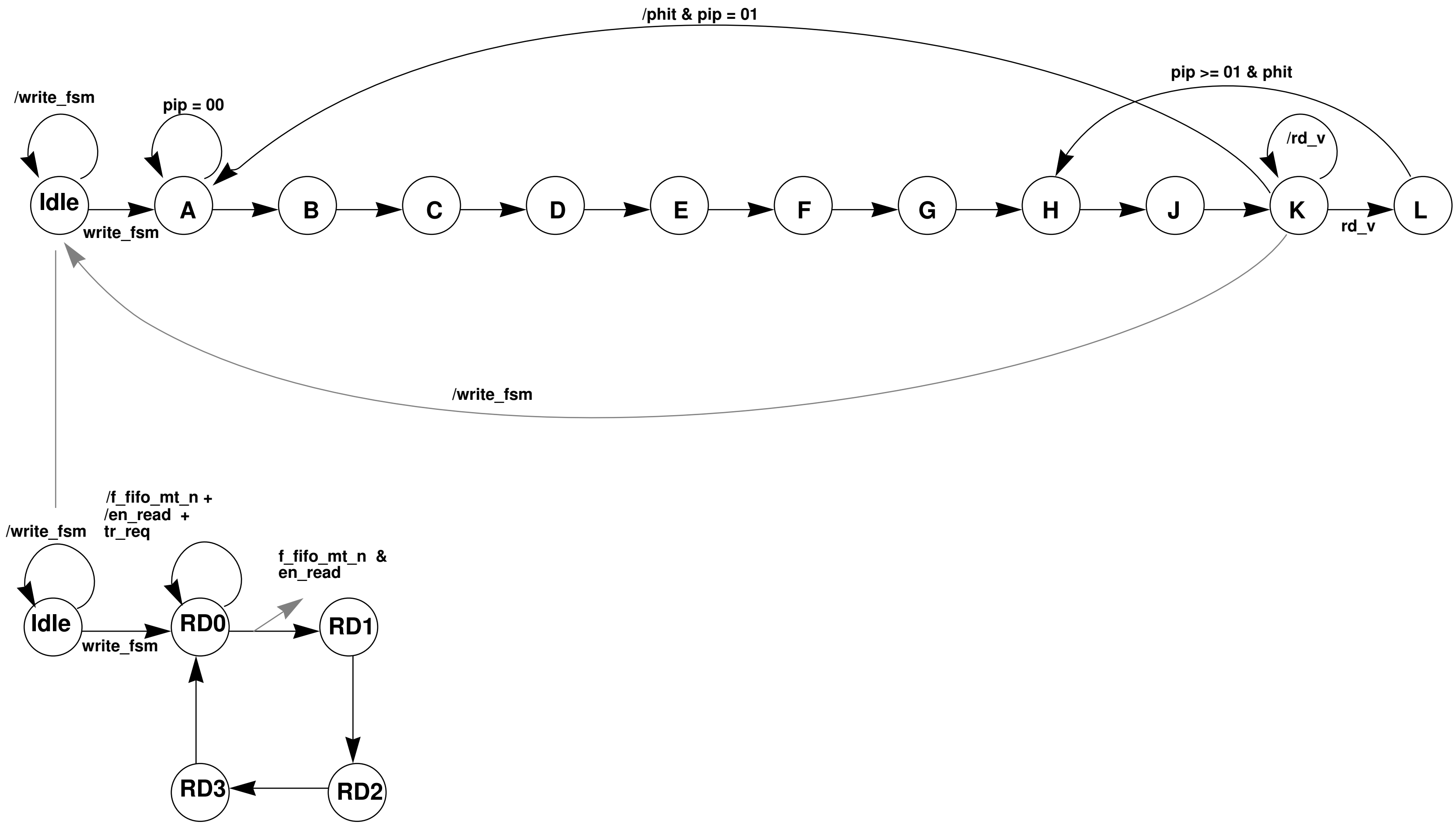
FIGURE 12. Newport graphics memory control block diagram for one bank



KEY:

fr -- fast_read cycles (read only)
fastx -- fast X mode (dest logic op only)
rmw -- read modify write cycles
rrmw -- read/read modify cycles (e.g. cid checked, dest logic op write)
rd_v -- a valid read of the write bank fifo occurred
dv -- data available for writing
phit -- Page fault indicator
pip -- # of pixels in the pipe
rd_fifo_full -- read fifo full flag
f_fifo_mt_n -- write bank fifo empty flag
rmw_fsm -- enable this fsm.
tr_req -- transfer request
en_read -- enable fifo read

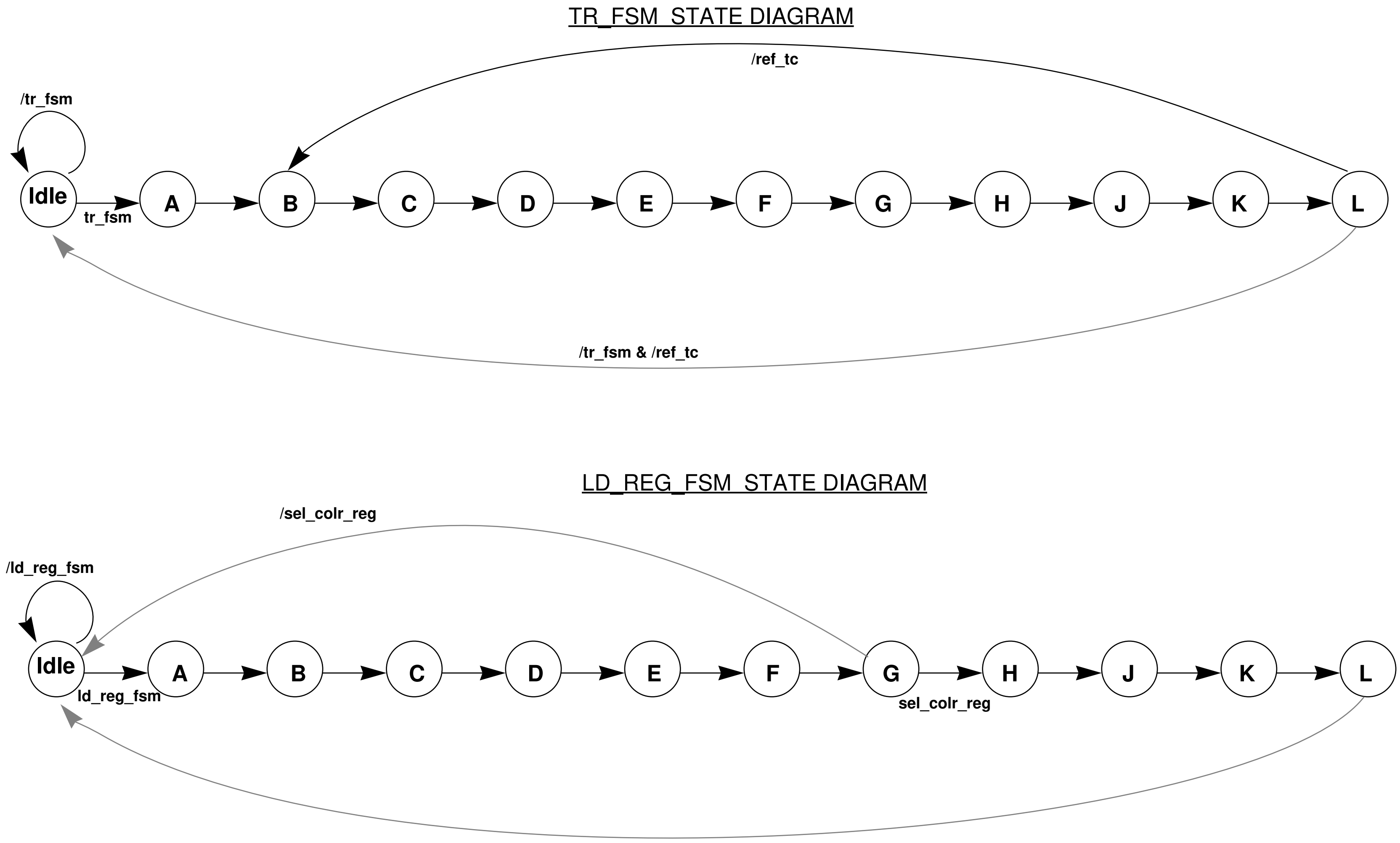
FIGURE 13. RMW_FSM State Diagram



KEY:

dv -- data available for writing
phit -- Page fault indicator
pip -- # of pixels in the pipe
tr_req -- transfer request
f_fifo_mt_n -- write bank fifo empty flag
write_fsm -- enable this fsm
en_read -- enable fifo read

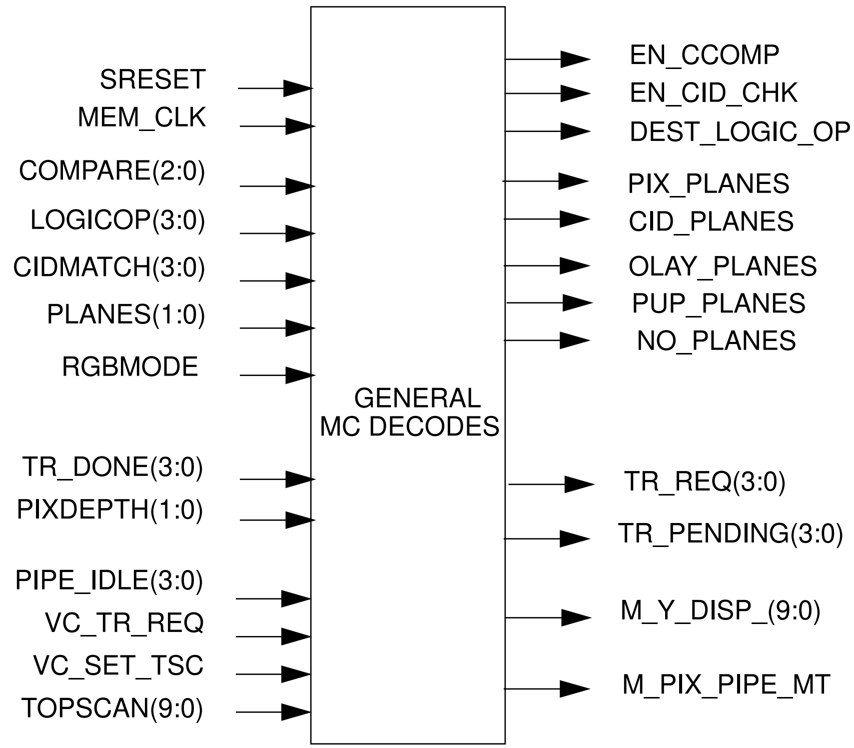
FIGURE 14. WRITE FSM State Diagram



KEY:

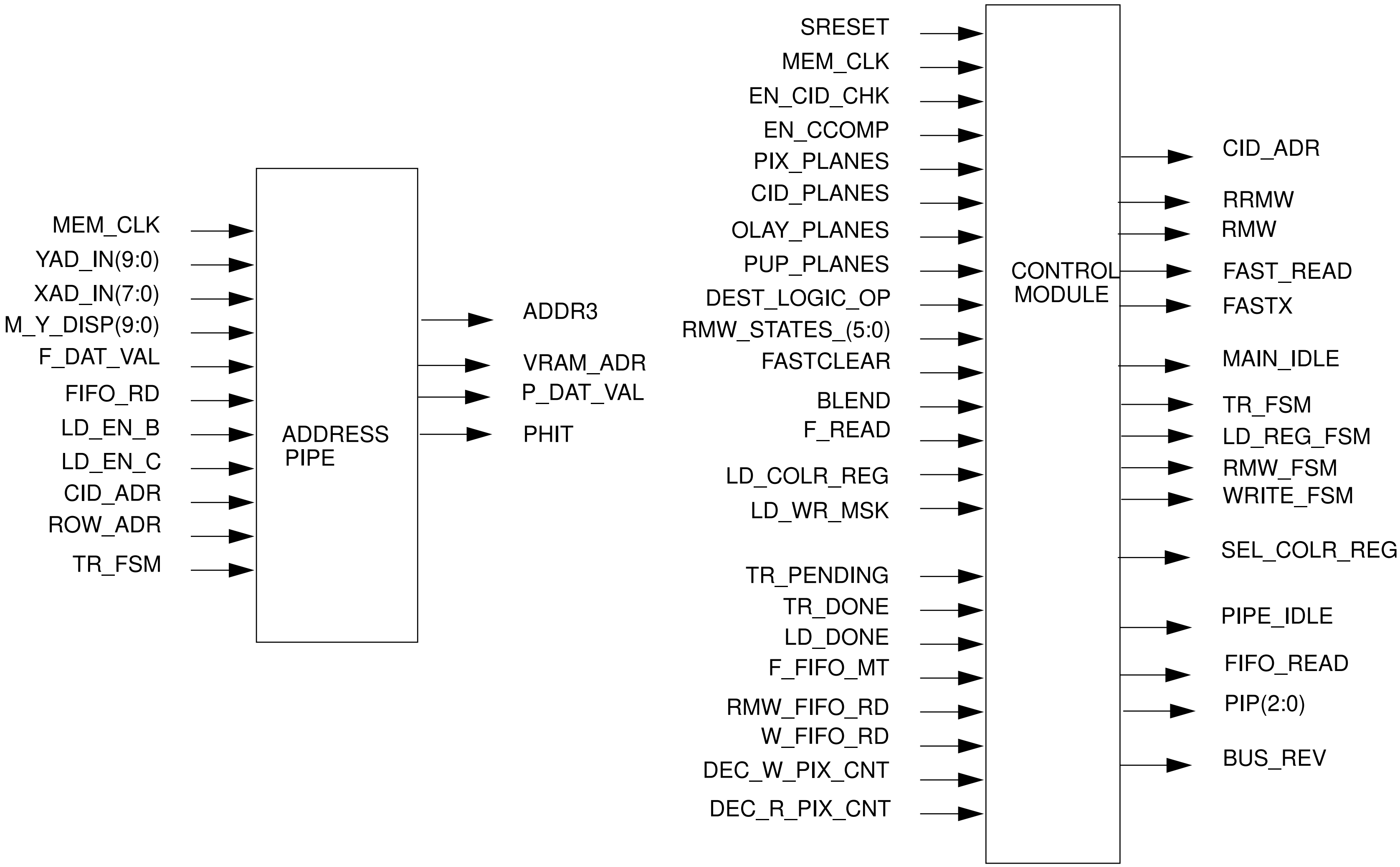
tr_fsm -- transfer fsm enabled
ref_tc -- refresh count is zero
sel_colr_reg -- load color register (instead of write mask)
ld_reg_fsm -- load register fsm enabled

FIGURE 15. TR_FSM and LD_REG_FSM State Diagrams



THIS MODULE IS SHARED BETWEEN ALL 4 BANKS

FIGURE 16. General MC Decodes pinout



1 ADDRESS PIPE AND 1 CONTROL MODULE PER BANK

FIGURE 17. Address pipe and Control Module pinout

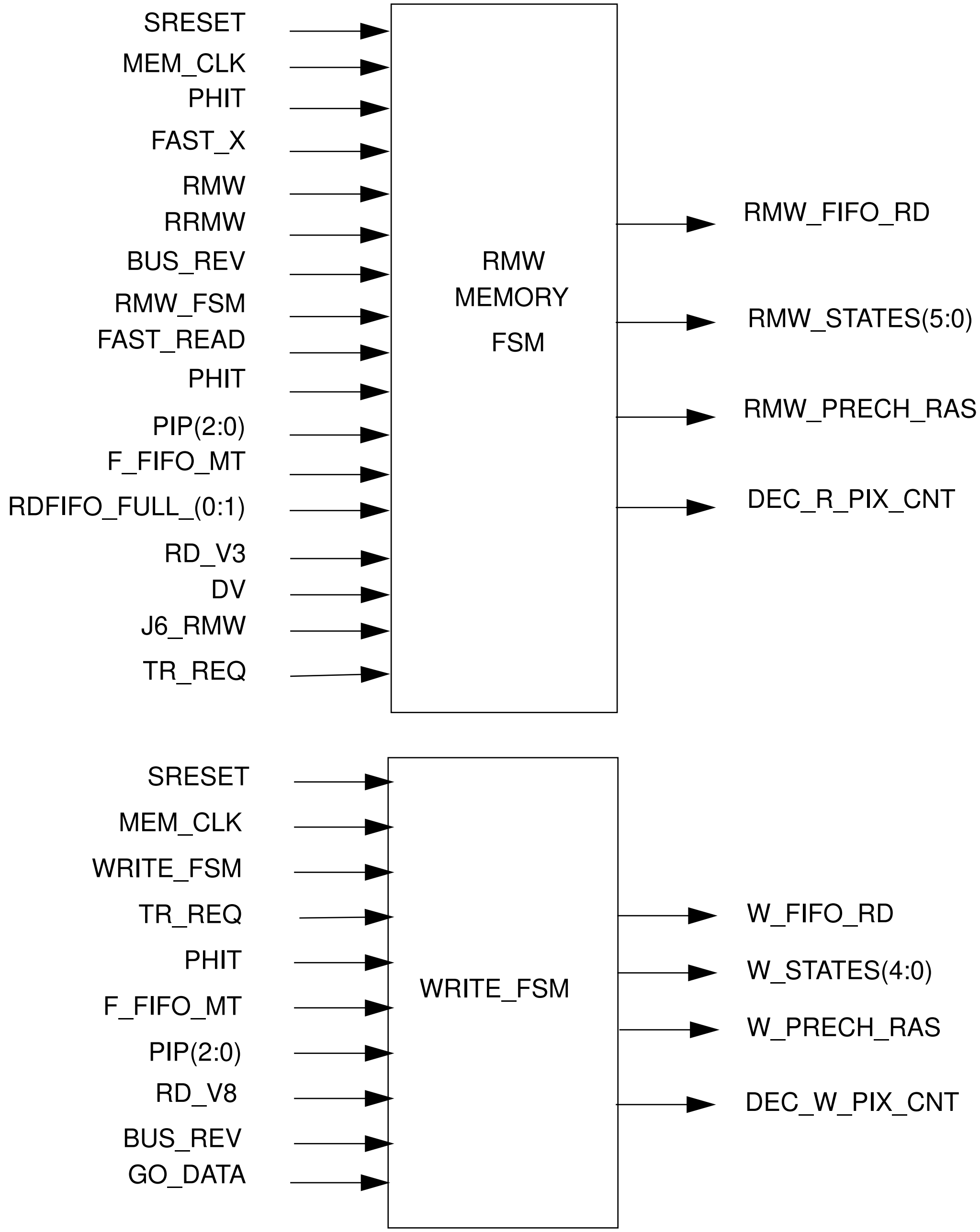


FIGURE 18. RMW_FSM and WRITE_FSM Pinout

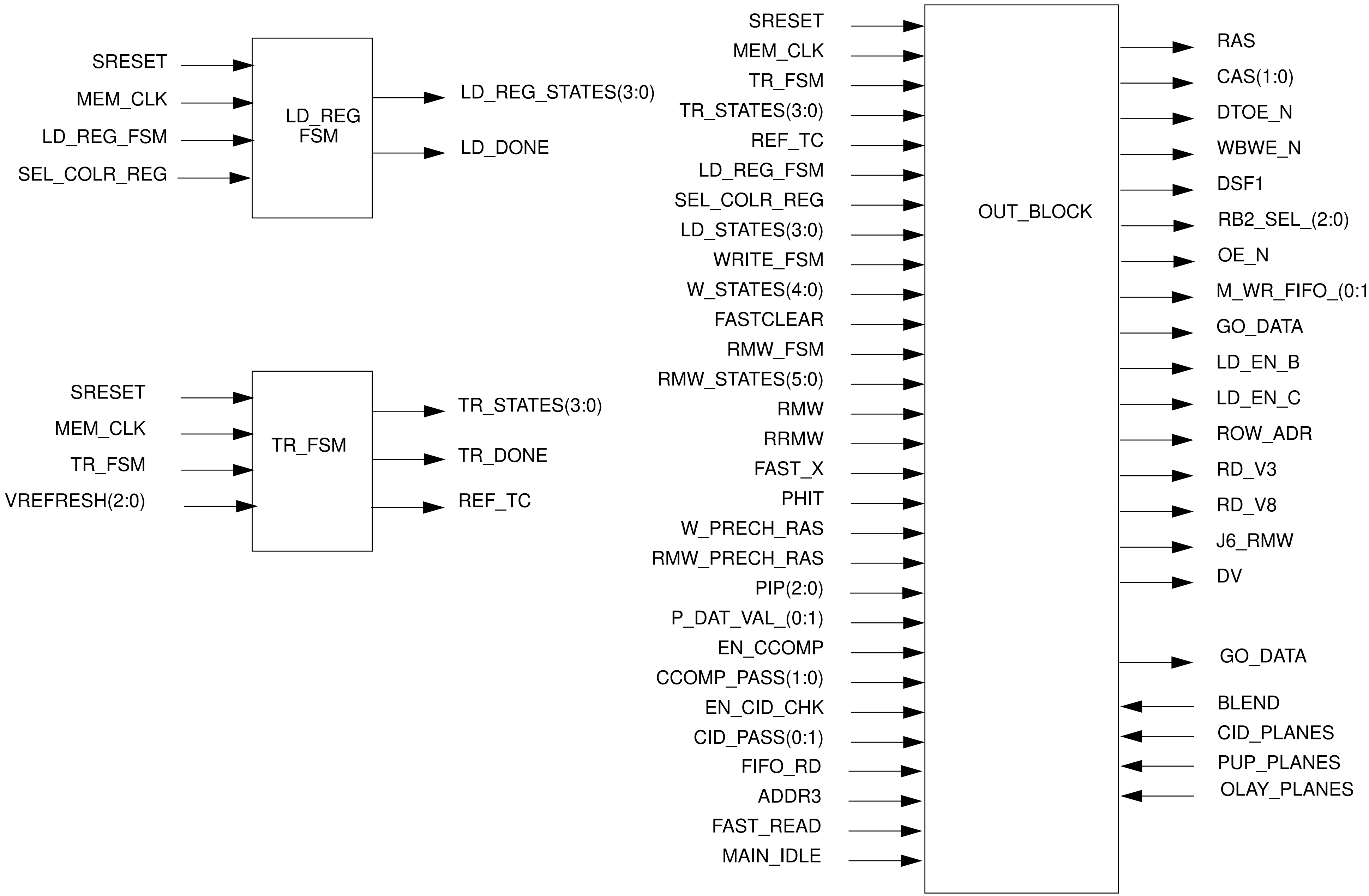


FIGURE 1. LD_REG_FSM, TR_FSM and OUT_BLOCK pinout

Table 30: GENERAL MC DECODES MODULE PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
SRESET	Synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
COMPARE(2:0)	Color compare bits from DRAWMODE1 REG.	I
LOGICOP(3:0)	Pixel logicop from DRAWMODE1 REG.	I
CIDMATCH(3:0)	Cid match bits from DRAWMODE1 REG.	I
PLANES(1:0)	Planes bits to indicate which planes to write. From DRAWMODE1 REG.	I
RGBMODE	RGB Vs CI mode from DRAWMODE1 REG.	I
TR_DONE	Transfer request is done.	I
PIXDEPTH(1:0)	Drawn pixel depth From DRAWMODE1 REG.	I
PIPE_IDLE(3:0)	Pixel pipe idle from each bank.	I
VC_TR_REQ	Transfer request from VC.	I
VC_SET_TSC	Set top of scan from VC. To set first displayable line after Vsync.	I
TOPSCAN(9:0)	Top of scan set by host in GIO reg block.	I
EN_COMP	Decode to enable color compare	O
EN_CID_CHK	Decode to enable CID checking	O
DEST_PIX_OP	Destination pixel is required (hence atleast RMW for frame buffer)	O
PIX_PLANES	Read/write pixel planes	O
CID_PLANES	Read/write cid planes	O
OLAY_PLANES	Read/write overlay planes	O
PUP_PLANES	Read/write pop up planes	O
NO_PLANES	No planes are selected for transaction	O
TR_REQ(3:0)	Transfer request to control module. This is asserted after VC_TX_REQ rising edge.	O
TR_PENDING(3:0)	Transfer request pending to control module. Asserted 500nS after TR_REQ	O
M_Y_DISP_(9:0)	Current line being displayed. Updated by VC_TR_REQ	O
M_PIX_PIPE_MT	PIPE_IDLE(3:0) AND'ed to indicate all MC banks are idle	O

5.4.0.1 General decodes module

This module decodes various global signals and provides information to the state machines as to which planes are being accessed currently. It also contains the line counter for screen refresh and communicates to the DDA section that all the banks are idle. The line counter increments every time the leading edge of VC_TR_REQ is detected. VC_SET_TSC resets the line counter to the value in TOPSCAN(9:0). This module is shared between al 4 banks. VC_TR_REQ and VC_SET_TSC should have a minimum pulse width of 30nS

Table 31: CONTROL MODULE PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
SRESET	synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
EN_CID_CHK	CID checking enabled	I
EN_CCOMP	Enable color compare	I
PIX_PLANES	Read/write pixel planes	I
CID_PLANES	Read/write cid planes	I
OLAY_PLANES	Read/write overlay planes	I
PUP_PLANES	Read/write pop up planes	I
NO_PLANES	No planes are selected for transaction	I
DEST_PIX_OP	Dest pixel required	I
FASTCLEAR	Enact block write cycles of Vram	I
RMW_STATES(5:0)	State vector from RMW FSM	I
BLEND	Enable blend function	I
F_READ	Read bit from write bank fifo	I
LD_COLR_REG	Load color reg command from host	I
LD_WR_MSK	Load write mask reg command from host	I
TR_PENDING	Transfer pending. Do transfer only after this signal is active.	I
TR_DONE	Transfer is done	I
LD_DONE	Reg load due to LD_COLR_REG or LD_WR_MSK is done	I
F_FIFO_MT	Write bank fifo is empty	I
RMW_FIFO_RD	Write bank fifo read command from RMW FSM (increment PP counter)	I
W_FIFO_RD	Write bank fifo read command from DMA FSM (increment PP counter)	I
DEC_W_PIX_CNT	Decrement PP counter command from DMA FSM (i.e pix is written)	I
DEC_R_PIX_CNT	Decrement PP counter command from RMW FSM (i.e pixel is read/written)	I
CID_ADR	Select CID address for the address pipe	O
RRMW	Mem cycle requires read/read/modify write	O
RMW	Memory cycle requiring read modify write	O
FAST_READ	Pipelined read of FB (due to Scr2Scr or Host DMA/PIO read)	O
FAST_X	Destination logic-op cycles only	O
MAIN_IDLE	Main state machine is in idle state	O
TR_FSM	Serial transfer has been requested.	O
LD_REG_FSM	Enable LD_REG FSM (do color or write mask reg load)	O
RMW_FSM	Enable RMW FSM (do pipeline reads or “r/mw type” mem cycles)	O
WRITE_FSM	Enable DMA_WRITE FSM (do pipelined writes to the FB)	O
SEL_COLR_REG	Mux select for color data	O
PIPE_IDLE	No more pixels in the pipe or to process	O
FIFO_READ	AND of RMW_FIFO_RD_N, DMA_FIFO_RD_N	O
PIP(2:0)	# of pixels in the pipe.	O
BUS_REV	Indicator for bus reversal on next pixel	O

5.4.0.2 Control module

The Control Module is responsible for enabling one of 4 state machines (write_fsm, rmw_fsm, tr_fsm or ld_reg_fsm). It also indicates to rmw_fsm as to the type of cycle to perform. The PIP counter resides in this module as does the arbiter for screen/memory refresh.

Table 32: Address Pipe Pin Description

Signals	Description	Type
MEM_CLK	Memory clock	I
YAD_IN(9:0)	Y address from write bank fifos	I
XAD_IN(7:0)	X address from write bank fifos.	I
M_Y_DISP(9:0)	Display refresh address for showing the next line.	I
F_DAT_VAL(0:1)	Data valid bits from write bank fifos.	I
FIFO_RD	Fifo read strobe	I
LD_EN_B	Load stage B in the address pipeline	I
LD_EN_C	Load stage C in the address pipeline	I
CID_ADR	Command to compute the AUX planes address	I
ROW_ADR	Enable the row address to the Vram address port	I
TR_FSM	Indication of serial read transfer	I
ADDR3	Address X3 for selecting high/low pixel in AUX planes.	O
VRAM_ADR(8:0)	Multiplexed Vram address	O
P_DAT_VAL(0:1)	Pipelined version of F_DAT_VAL(0:1)	O
PHIT	Page hit indicator	O

5.4.0.3 Address Pipe

The address pipe module contains the Vram address calculator and the address mux. This module is responsible for the address data path. The addresses have a 4 clock pipe line and can hold 3 different addresses at any time. The Vram page comparator also resides in this module.

Table 33: RMW_FSM PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
SRESET	Synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
PHIT	Page hit form FB page comparator in ADDR_PIPE module	I
FAST_X	Decode for 8 bit CI and logicop with no cid chk (for X perf)	I
RMW	Enables this state machine. Otherwise idle	I
RRMW	Mem cycle requires read/read/modify write	I
BUS_REV	Bus reversal has occurred on current fifo read	I
RMW_FSM	Enable the RMW state machine	I
FAST_READ	Pipelined read of FB (due to Scr2Scr or Host DMA read)	I
PIP(2:0)	# of pixels left in pipeline	I
F_FIFO_MT	Write bank fifo empty	I
RDFIFO_FULL_(0:1)	Read bank fifo full	I
RD_V3	Write bank fifo read delayed 3 clocks	I
DV	Data valid for rmw_state = W	I
J6_RMW	State J of the RMW_FSM delayed by 6 clocks	I
TR_REQ	Refresh transfer request	O
RMW_FIFO_RD	Write bank fifo read command	O
RMW_STATES(5:0)	State machine state bits	O
RMW_PRECH_RAS	Precharge in next clock for RMW_FSM	O
DEC_R_PIX_CNT	Decrement PP counter	O

5.4.0.4 RMW state machine

The RMW state machine performs the following types of cycles:

a. Read frame buffer
b. Read modify write
c. Read/read modify write
d. Fast_X

The read cycles can read any of bit planes in pipe line mode. The read data is written into the read bank fifos. Read cycles can be used for PixBlit or host read operations.

The read modify write cycles can do the following:

a. CID checked writes in the pixel planes
b. Non CID checked blended writes in the pixel planes
c. Non cid checked color compared writes in any of the planes
d. Non cid checked writes in the aux planes.

The Read / read modify cycles are used for the following:

a. Cid checked color compared writes in any planes
b. Cid checked blends in the pixel planes.

The Fast_X mode is used for destination Logic-op's only.

Table 34: WRITE_ FSM PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
SRESET	Synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
WRITE_FSM	Enable WRITE_FSM	I
~~TR_REQ~~	~~Refresh request for screen update~~	~~I~~
PHIT	Page hit form FB page comparator in ADDR_PIPE module	I

Table 34: WRITE_ FSM PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
PIP(2:0)	# of pixels left in pipeline	I
F_FIFO_MT	Write bank fifo empty	I
BUS_REV	Bus reversal due to current read of fifo	I
W_FIFIO_RD	Write bank fifo read command	O
W_STATES(4:0)	State machine state bits	O
W_PRECH_RAS	Precharge ras in next clock for WRITE_FSM	O
DEC_W_PIX_CNT	Decrement PP counter	O

5.4.0.5 Write state machine

The WRITE_FSM executes memory cycles only. These cycles exclude any destination pixel processing. Block-mode writes into the Vrams are also executed by this state machine. The PIP counter increments or decrements whenever the write bank fifo is read or written respectively.

Table 35: LD_REG_FSM PIN DESCRIPTION

SIGNAL	DESCRIPTION	TYPE
SRESET	Synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
LD_REG	Enable LD_REG FSM	I
SEL_COLR_REG	Mux select for color data	I
LD_REG_STATES(3:0)	State machine state bits	O
LD_DONE	Register load done	O

5.4.0.6 Load registers state machine

LD_REG_FSM is responsible for loading the color register of the Vrams as well as the writemask register in the RB2. SEL_COLR_REG indicates when the color register needs to be loaded in the Vrams, otherwise the writemask is loaded in the RB2. Neither of these registers are readable from the RB2 or the Vram. A copy of these registers exists in the REX3 register file. Since the read/write format logic is in the RB2, a copy of the PLANES(2:0), DRAWDEPTH(1:0), RGB-MODE, LOGIC-OP(3:0) and DBLSRC bits are sent along sub bank 1 data bus while the writemask data is sent along sub bank 0. Every time the write mask or above mentioned registers are modified, a load write mask command is executed to keep the RB2 up-to-date with the current context. The color data value for loading the color register in the Vrams is sent on both the sub banks at the same time, one byte at a time.

Table 36: TR FSM PIN DESCRIPTION

SIGNALS	DESCRIPTION	TYPE
SRESET	Synchronous reset	I
MEM_CLK	Memory clock 66MHz.	I
TR_FSM	Enable TR FSM	I
VREFRESH(2:0)	Number of memory refreshes to do in burst refresh	I
TR_STATES(3:0)	State machine state bits	O
TR_DONE	Transfer and refresh done	O
REF_TC	# of specified refreshes are done	O

5.4.0.7 Refresh

The leading edge of transfer request is detected and a 480nS timer is activated and the line counter is incremented. During the 480nS period, a second transfer can be asserted by VC2 so the line counter can be incremented again (for interlace mode). The timer will not be reset due to the second transfer request. At the end of the 480nS period a refresh request is made to the arbiter. The TR_FSM first does a read transfer cycle followed by zero to eight memory refresh cycles depending on the value of VREFRESH(2:0).

VREFRESH(2:0) being “000” disables memory refresh.

Table 37: OUT_BLOCK DESCRIPTION

SIGNALS	DESCRIPTION	TYPES
SRESET	Synchronous reset signal	I
MEM_CLK	Memory clock 66MHz.	I
TR_FSM	Enables transfer fsm	I
REF_TC	# of specified memory refreshes are done	I
LD_REG_FSM	Enables load register fsm	I
SEL_COLR_REG	Load color reg. in Vram	I
LD_STATES(3:0)	LD_REGFSM state bits	I
WRITE_FSM	Enables the write fsm	I
W_STATES(4:0)	WRITE_FSM state bits	I
FASTCLEAR	Perform Vram blockfill (see block write function of Vram)	I
RMW_FSM	Enables rmw fsm	I
RMW_STATES(5:0)	RMW_FSM state bits	I
RMW	Read modify write cycles for rmw_fsm	I
RRMW	Read/read modify write cycles for rmw_fsm	I
FASTX	Special cycles for rmw_fsm to speed up X11 operations	I
PHIT	Page hit indicator for frame buffer	I
W_PRECH_RAS	Precharge ras in next clock for write_fsm	I
RMW_PRECH_RAS	Precharge ras in next clock for rmw_fsm	I
PIP(2:0)	# of pixels in the pipe	I
P_DAT_VAL(0:1)	Pipelined data valid bit from write bank fifo	I
EN_CCOMP	Enable color compare (also set for A function since comparators are shared)	I
CCOMP_PASS(0:1)	Color compare (or A function) pass bits	I
EN_CID_CHK	Enable cid checking	I
CID_PASS(0:1)	Cid pass bits	I
MAIN_IDLE	Main state machine is idle	I
ADDR3	Address X3 from module addr_pipe1	I
FIFO_RD	Write bank fifo read signal	I
FAST_READ	Fast read type cycles for the rmw state machine	I
OLAY_PLANES	Overlay planes are to be accessed	I
CID_PLANES	Access CID planes	I
PUP_PLANES	Access PUP planes	I
BLEND	Blend function is enabled	I
RAS	Row Address Strobe to Vrams	O
CAS(0:1)	Column Address Strobe to Vrams	O
DTOE_N	Data transfer / Output enable signal to Vrams	O
WBWE_N	Write per bit / write enable signal to Vrams	O
DSF1	Special function signal to Vrams	O
RB2_SEL_(2:0)	Byte select codes to RB2	O
OE_N	Output enable for the data bus to RB2	O
M_WR_FIFO_(0:1)	Write strobes for the read bank fifos	O
GO_DATA	Signal to start the data flowing from the fifo read register thru the dither and out.	O
LD_EN_B	Load stage B of address pipe	O
LD_EN_C	Load stage C of address pipe	O
ROW_ADR	Select row address for the address pipe	O
RD_V3	Write bank fifo read delayed 3 clocks	O
RD_V8	Write bank fifo read delayed 8 clocks	O
J6_RMW	State J of RMW_FSM delayed by 6 clocks	O
DV	Data valid for rmw state machine in state W	O

5.4.0.8 Out block module

The OUT_BLOCK module decodes the state bits of each state machine and generates frame buffer control signals. It also controls the address pipeline and generates go_data to the data pipe to control data flow. RB2 controls are also generated here by decoding the state bits. The encoding of the RB2_SEL(2:0) is as shown in Table 38.

RB2_SEL(2:0)	Function
0	NOOP
1	Write (4 components), also used to write LO pixel in AUX planes
2	Write HI pixel in AUX planes (else if blend then hold data in sub-bank 1 output of RB2)
3	Load write mask and partial DRAWMODE1 Reg. into RB2
4	Read (4 components), also used to read LO pixel in AUX planes.
5	Read HI pixel in AUX planes
6	Read LO pixel CID bits for cid checking.
7	Read HI pixel CID bits for cid checking.

Table 38: RB2_SEL(2:0) Function codes.

When performing a Load Write Mask operation, the write mask is sent on sub bank 0 data bus and the drawmode1 register bits are sent on sub bank 1 data bus to RB2. The drawmod1 registers will be sent in the following order starting with the
LSB of the first byte: This is shown in Table 39.

Bit 12	Bit 11	Bit 10	Bit 9	Bit (8:6)	Bit (5:4)	Bit (3:0)
Blend	Fastclear	RGBmode	Dblsrc	Planes(2:0)	Drawdepth(1:0)	Logicop(3:0)

Table 39: Transmission order of Drawmode1 reg. bits to RB2

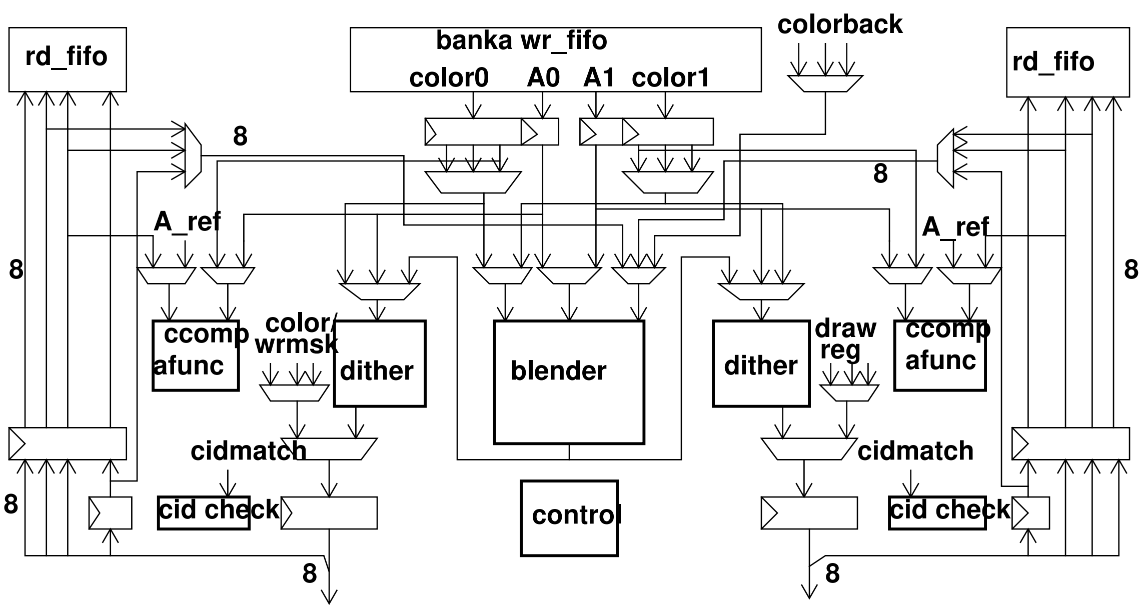
5.4.1 Gate Count

Block	Description	Gate count	Total(4Banks)
ADDRESS PIPE	Address pipeline to VRAMs	950	3800
LD_REG_FSM	State machine for loading registers	80	320
OUT_BLOCK	Output decode block	650	2600
RMW_FSM	Read modify write state machine	400	1600
TR_FSM	Transfer and refresh state machine	150	600
WRITE_FSM	Write state machine	200	800
GEN. MC DECODE	General MC decoder	100	400
CONTROL MOD	Master state machine controller	200	800
Grand Total	-	2730	10920

Table 40: Gate count of memory controller state machine and address pipe.

5.4.3 Pixel Processing Pipe

There are two pixel pipes in each bank to process the two adjacent pixels before writing into framebuffer. The processes can be dithering, logicop, alpha blending, cid checking, color comparison and afunction. In order to reduce gate count in REX3, logical operation, write formating and read formating are performed in RB2. The block diagram of the pixel pipes in each bank is shown below. The colors and alpha's of two adja-



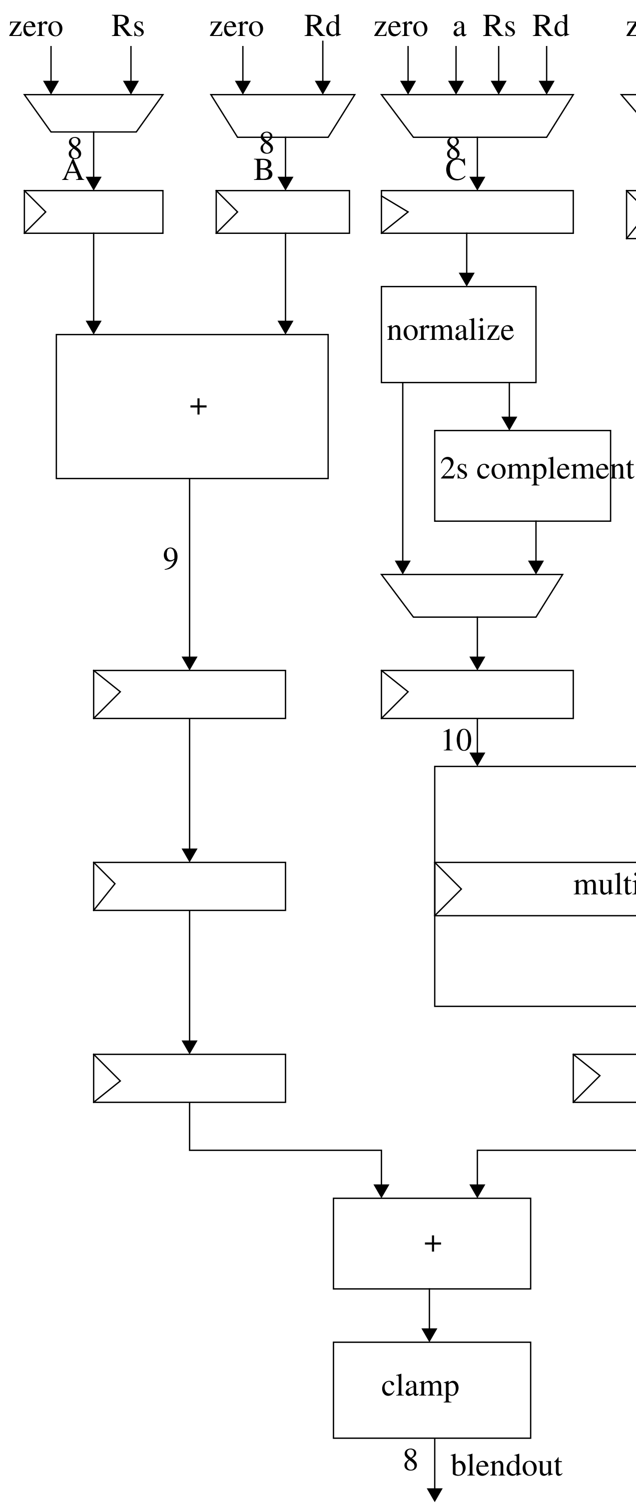
cent pixels in the bank can be read from write fifo. Each color component and alpha are pipelined through dithering block and are collected and formated to frame buffer format in RB2 chip. The cycle time of pixel processing pipe is 15 ns. Processing 4 color components of a pixel takes 4 cycles(60 ns), which matchs the memory cycle time. When blender is enabled, the source color component blends with the destination color component through the blender pipe and then go through the dithering block. The color compare block not only compare the source color index with the destination color index but also compare the source alpha with reference alpha. Whenever the drawing contex is changed, REX3 loads write mask from pipe0 and some draw mode bits from pipe1 to RB2 to support RB2 for performing read write formating and logicop.

5.4.3.1 Blender

The blender performs the blending function Cb=Fs*Cs + Fd*Cd, where Cb is blended color, Cs is source color and Cd is destination color. The source factor Fs and destination factor Fd are defined in section 3.8.5 Table 21 and Table 22. In order to reduce gate count, the equation Fs*Cs + Fd*Cd is decomposed to A + B + (+/-C)*(D+/-E) so that only one multiplier is required. The following table shows all the source factor and destination factor combinations and the terms after decomposition. The block diagram of the blender is also shown in the next page.

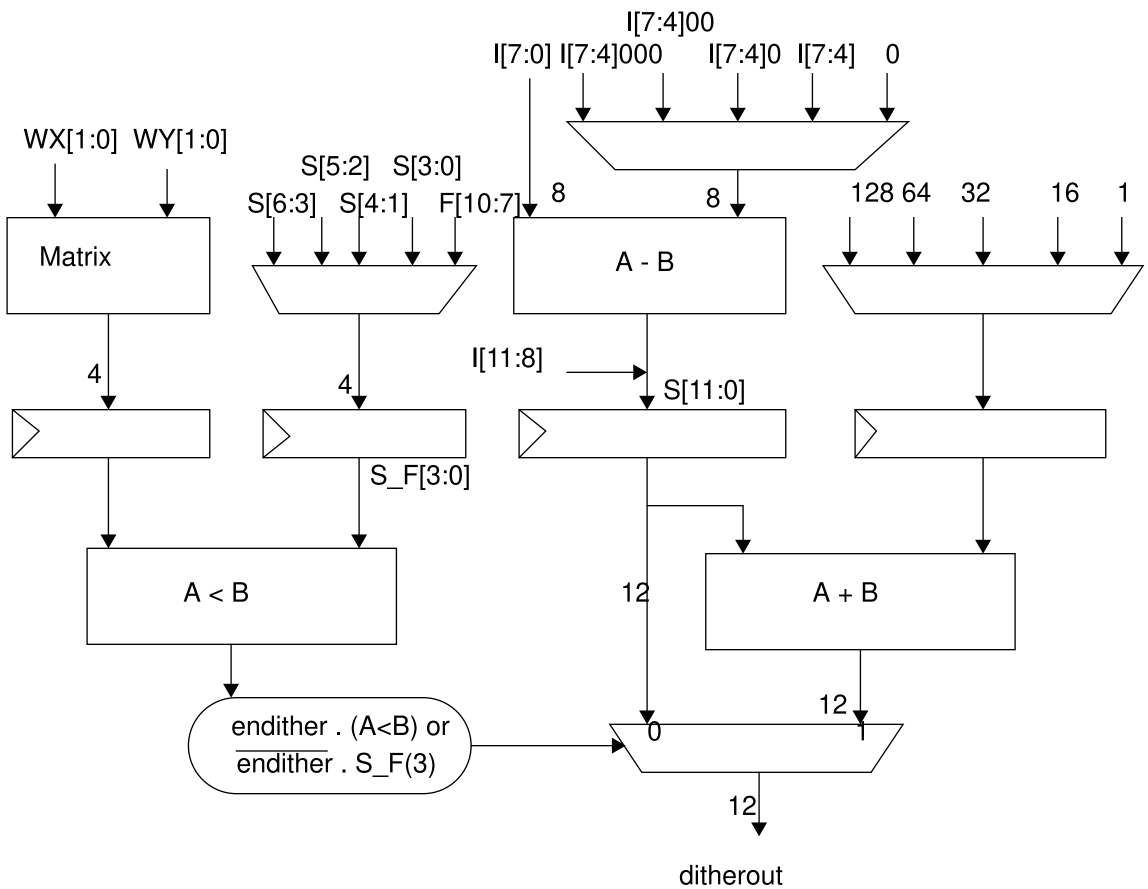
Table 28: Decompose Fs*Rs+Fd*Rd to A+B+(+/-C)*(D +/- E)

SF	DF	Fs*Rs+Fd*Rd	A	B	+/-C	D	+/-E
0	0	0	0	0	0	0	0
	1	Rd	0	Rd	0	0	0
	2	Rs*Rd	0	0	Rs	0	Rd
	3	(1-Rs)*Rd	0	Rd	Rs	0	-Rd
	4	A*Rd	0	0	A	0	Rd
	5	(1-A)*Rd	0	Rd	A	0	-Rd
1	0	Rs	Rs	0	0	0	0
	1	Rs+Rd	Rs	Rd	0	0	0
	2	Rs+Rs*Rd	Rs	0	Rs	0	Rd
	3	Rs+(1-Rs)*Rd	Rs	Rd	Rs	0	-Rd
	4	Rs+A*Rd	Rs	0	A	0	Rd
	5	Rs+(1-A)*Rd	Rs	Rd	A	0	-Rd
2	0	Rd*Rs	0	0	Rd	0	Rs
	1	Rd*Rs+Rd	0	Rd	Rd	0	Rs
	2	Rd*Rs+Rs*Rd	0	0	Rd	Rs	Rs
	3	Rd*Rs+(1-Rs)*Rd	0	Rd	0	0	0
	4	Rd*Rs+A*Rd	0	0	Rd	Rs	A
	5	Rd*Rs+(1-A)*Rd	0	Rd	Rd	Rs	-A
3	0	(1-Rd)*Rs	Rs	0	-Rd	0	Rs
	1	(1-Rd)*Rs+Rd	Rs	Rd	-Rd	0	Rs
	2	(1-Rd)*Rs+Rs*Rd	Rs	0	0	0	0
	3	(1-Rd)*Rs+(1-Rs)*Rd	Rs	Rd	-Rd	Rs	Rs
	4	(1-Rd)*Rs+A*Rd	Rs	0	-Rd	Rs	-A
	5	(1-Rd)*Rs+(1-A)*Rd	Rs	Rd	-Rd	Rs	A
4	0	A*Rs	0	0	A	Rs	0
	1	A*Rs+Rd	0	Rd	A	Rs	0
	2	A*Rs+Rs*Rd	0	0	Rs	A	Rd
	3	A*Rs+(1-Rs)*Rd	0	Rd	Rs	A	-Rd
	4	A*Rs+A*Rd	0	0	A	Rs	Rd
	5	A*Rs+(1-A)*Rd	0	Rd	A	Rs	-Rd
5	0	(1-A)*Rs	Rs	0	-A	Rs	0
	1	(1-A)*Rs+Rd	Rs	Rd	-A	Rs	0
	2	(1-A)*Rs+Rs*Rd	Rs	0	-Rs	A	-Rd
	3	(1-A)*Rs+(1-Rs)*Rd	Rs	Rd	-Rs	A	Rd
	4	(1-A)*Rs+A*Rd	Rs	0	-A	Rs	-Rd
	5	(1-A)*Rs+(1-A)*Rd	Rs	Rd	-A	Rs	Rd


zero
Rs
zero
Rd
zero
a
Rs
Rd
zero
a
Rs
Rd
zero
a
Rs
Rd
8
A
8
B
8
C
8
D
8
E
normalize
+
+ / -
2s complement
9
10
10
10
multiplier
+
clamp
8
blendout

5.4.3.2 Dithering

The following dithering block implements the dithering and rounding algorithms described in section 3.8.2 and 3.8.3.



5.4.3.3 Gate count of pixel processing pipe

The following estimated gate count including the gates for ATPG.

Table 29: Gate count of pixel processing pipe

Block	Description	Gate count	Total
mem_data_io	read and write registers , mux's, cid check	720 X 8	5760
dither	fifo read registers , dithering and rounding	1260 X 8	10080
ccomp	mux's and color compare	200 X 8	1600
blender	blender including multiplier	3100 X 4	12400
pipe_ctrl	pipe controller	540 X 4	2160
blend_ctrl	global blendfunction selector	50 X 1	50
write fifo	90x3 write fifo	1350X4	5400
read fifo	32x5 read fifo	800X8	6400
GRAND TOTAL	overall pixel pipe gate count		43,850

6 Revision History

0.1 First release 4-24-92

0.2 5-4-92

0.3 8-11-92 rws

Linedraw: new opcodes (A_EDGE replaced by T_LINE, B_LINE); endpoint filtering now enabled by DRAWMODE0 bit ENDPTFILTER; new secondary pixel calculation register BRESS2 added, and linedraw algorithm updated to reflect this change; octant mirroring effect on minor axis fraction simplified to be function (1.0 minus .frac) including case of .frac = zero; endpoint filtering algorithm completed; coverage function BRESS renamed BRESS1, and its range behavior changed to [-1.0 thru 1.0].

Z buffered, antialiased linedraw: behavior modified so that ZPATTERN contains primary pixel zmask, and the LSPATTERN register contains secondary pixel mask, for cases of (A_LINE, T_LINE, B_LINE) with (ENZPATTERN and ENLSPATTERN both asserted).

Setup: overhead is one clock for quadrant calculation (span or block), four clocks for I_LINE, and eleven clocks for the F_LINE, A_LINE, T_LINE, B_LINE cases.

Pipeline stalls: a one-clock delay is added for case of X,Y end condition reached, due to H/W implementation issues at this clock speed.

Context Switching: host overhead added for handling the SLOPERED register; see Section 3.12 for more details.

Off-Screen Memory: reduced to 64 pixels wide (at right of screen).

Screen-to-Screen Moves: XYMOVE is now treated as offset to destination; also, the XYMOVE is interpreted based on YFLIP, so the S/W can treat it as being YFLIP-independent.

0.4 8-19-92 Adrian

Added the explicit request for reloading the AWEIGHT table every time the slope (e1) of an antialiased line changes (in effect AWEIGHT must be reloaded for every antialiased line or edge).

0.5 11-04-92 rws ERATTA LIST etc.

Z-Buffered Antialiased lines: modified so that zpattern is now used for bottom of clockwise-rotated line, whereas lspattern is used for top. REX3 determines what is top or bottom. tline, bline functionality can be achieved via setting masks above for desired edge. However, tline,bline selection of top versus bottom is correct only for odd-numbered octants (1,3,5,7) and is reversed for even octants (2,4,6,8).

Subpixel positioned lines: the setup algorithm is modified to handle case where first pixel, when selected as closest pixel center to tangent of line, is different than specified vertex (along the minor axis). This adds the E and G tests. The benefit of this is that pixel selection is exact and independent of vertex swapping.

Bresd term is changed from s17.8 to s18.8 format to handle full range of values.

Default octant is “111” where octant is defined as XMAJOR & XDEC & YDEC where $DEC indicates direction of stepping of axis $ is in negative direction (if =1) or positive direction (if=0). There is one exception to this: if doing line setup, then minor axis $DEC is zeroed if (BRESINC1=0 and (major axis $DEC=1)). This is done so that horizontal or vertical edges will have opposing behavior for antialiasing when direction is reversed.

BRESROUND interpretation is: msb pertains to octant 1, lsb to octant 8.

Host must zero the color fractions in the COLOR$ registers whenever drawing with host data, or doing screen to screen moves. Else rounding may occur: not an issue for CI.

Addresses never to be written with GO: LSSAVE, LSRESTORE, SETUP.

COLORRED format change: if written to with DRAWMODE set at 12b CI mode, the h/w will take assumed format of o12.9 and place it in a o12.11 field, shifting up by 4 and zero filling. Context restore must then be done with DRAWMODE not in 12b CI mode. This new COLORRED format is a hack to handle certain constraints imposed by floating point format (23b mantissa).

Lines cannot be read back to host or dma'd as lines.

Endpoint filtering coverage details: pixel coverage along major axis is 0x10 to 0x1f for each pixel. This value is comprised of a start coverage and an end coverage. If current pixel contains the starting point, a value of [XSTARTFRAC xnor XDEC] yield starting coverage (note: xmajor case assumed here for discussion). If the current pixel does not contain the starting point, a value of 0x0f is used. The ending point coverage is calculated similarly, but using value when current pixel is the endpoint of [XENDFRAC xor XDEC]. The pixel coverage is the sum of (starting coverage plus ending coverage plus one). The 5b result is used in the following way: if coverage<3>=1, then the AWEIGHT coverage value selected by the minor axis antialiasing lookup is passed through unchanged as the coverage value; elsif coverage(2>=1, the AWEIGHT value is shifted right one position; and so forth to a minimum coverage value of AWEIGHT shifted down three positions (divided by eight). This naturally handles the case of starting and ending points being within a single pixel square.

Programming restrictions: never do a write with "GO" set to the following registers or fields, unless DRAWMODE bit DOSETUP=1:

> xend (all formats), xymove, xywin, drawmode, octant;

the issue is related to changing the end condition in the X direction having some latency within REX3. So, if the write does not change any bit affecting the X end condition, there is no problem. (from tarolli: xend is only used with GO for drawing flat shaded triangle spans; drawmode mostly written with DOSETUP=1;)

Fastclear and CID checking: REX3 will disable FASTCLEAR mode if CID checking is enabled. Therefore host must setup fast clear operation by writing COLORVRAM and also setting up DRAWMODE and COLORI (for example) register. This is necessary because GL will not know if window system invokes CID check. Fastclear and dithering: host (GL) will not invoke FASTCLEAR if dithering is enabled.

Screenmasks: Host must add XYWIN offset to SMASKS1-4; host must add XYMOVE offset to all SMASKs, as REX3 does not do this.

Setting REX3 color registers to value using sub-24b RGB color value: although the host may initialize color for drawing with a simple write to COLORI, this register only supports 8-8-8 B-G-R packing when in RGB mode (no problem for CI mode). In order for host to initialize color to a 3-3-2 value, for instance, more work must be done. The simplest solution is for host to replicate the desired color value into the two most significant pixel fields within HOSTRW format and do the write. This allows flat fill to occur at the full 100Mpix/sec rate. REX3 performs replication into 24b field automatically, so color rounding does not affect the result written (X11 issue). An alternate solution is for host to replicate each component into 8b fields of COLORI, clearly this is more CPU cycles. [note: in REX1 we simply would put the chip into CI mode to write sub-24b RGB values into framebuffer; in REX3 we have not swizzled the CI bits the same way as the RGB bits so this will not work, unfortunately].

0.6 11-10-92 rws

f_line no longer does G test, and has no minor axis adjustment step during setup.

AWEIGHT ordering across AWEIGHT0&AWEIGHT1 is simply most to least coverage, nibbles..

Endpoint filtering, when enabled by DRAWMODE0<22>, is applied to both endpoints of the line. Use of SKIPFIRST or SKIPLAST result in no filtering for the (masked) endpoint. Filtering of the starting vertex is done for each "GO" event, so when a line is broken up into segments endpoint filtering should be turned off for all but the first segment, and the ending vertex. Ending vertex filtering is only done for case of major axis end value reached. Therefore, unless a line is drawn atomically and/or with SKIPLAST set, filtering of the endpoint requires host intervention. (Not only must the endpoint filter be separated out of the loop, drawing the endpoint alone is considered as both a start AND end condition, so host must modify the major axis starting fraction to inhibit it from interfering with REX3 endpoint coverage calculation). Next time we may want to have separate startptfilter, endptfilter bits to alleviate this problem.

0.7 11-11-92 adrian

Removed the G test. Removed the start point coordinate adjustment based on the E test.

0.8 11-20-92 rws

XYOFFSET feature (adds XYMOVE to XYSTARTI during execution) is now only supported for framebuffer writes, not reads. (changed to improve timing, assumed not useful for reads).

0.9 12-03-92, mod 1-25-93 rws

Point draw notes: in order to circumvent the rule of no write to XEND with "GO" set, host may simply forego writing to XEND, YEND registers. Point is drawn in span (or block) mode with just the XSTART,YSTART pair. For DDA data, simply set STOPONX=STOPONY=0; for HOSTRW data source, set COLORHOST and/or ALPHAHOST as needed; and also set HOSTPACKED=0.

0.A 12-08-92 rws

Setup overhead update: some increase here. Spans/Blocks: 3 clocks; Integer Lines: 5 clocks; Subpixel Lines: 15 clocks. The Spans/Blocks increase due to chip timing modifications; the Subpixel Lines increase due mainly to E test. (Each increased by one clock for state machine timing issue, also.)

0.B 01-25-93 rws

XYWIN hints for GL: since the X,Y coordinates are biased by some number to facilitate simple vertex float-to-fixed point conversion, the GL may typically send to REX3 highly biased values. The YFLIP feature negates the Y values (YSTART, YEND, YMOVE, SMASK0Y) thereby subtracting the result from YWIN to calculate the Y value on the screen. In order for this to work correctly, the YWIN value must then be biased by TWO times the float-to-fixed bias whenever YFLIP is invoked. Internally, REX3 relies on 4K,4K bias for X,Y to be relative to screen (upper left) origin.

0.C 02-11-93 rws

Antialiased line coverage is a function of host-supplied slope BRESE1. This value must be calculated by using coordinate X,Y values which have the 7 lsb's cleared (lsb's of mantissa, assuming float-to-fixed point values being written by the GL to XSTARTF, YSTARTF, XENDF, YENDF). This in essence matches REX3 coordinate snapping to the 4b of subpixel precision, and will yield the best results.

0.D 06-21-93 rws

Point draw hints: it is simplest to use adrmode=iline, then there is no need to initialize the octant or to maintain consistent state of endpoint. No setup/dosetup is required; this approach preferred over that of 12-03 above.

X11 tiling hints: unfortunately REX3 does not provide support for arbitrary tiling, other than the primitive approach of host sending packed pixels and GO for each HOSTRW write. Performance test relies somewhat on 4x4 tiles, which REX3 doesn't support in the most efficient way. (an easy thing to add "next time").

Revision 0 bugs:

- dma preemption protocol bug is fixed in prototypes via rework, will fix in Rev 1 rex.
- dma resumption protocol bug (unspec'd MC behavior) causes 32b dma mode to fail, will fix Rev 1.
- memory controller bug for R-M-W screen-to-screen moves, state machine fix in Rev 1.
- stipple rotate bug for xby2 case of xstarti=xendi, s/w workaround by init'ing octant; fix Rev 1 (unfortunately lronly behavior required host to do the check, fixed Rev 1.)
- octant bit "xdec" was '1' for xstart=xend, changed to '0' to simplify above solution in Rev 1.

subpixel bresenham setup Etest unnecessary, deleted in Rev 1. (s/w workaround meanwhile).

aline coverage term incorrect, algorithm modified to base on sign (BRESS1), not S2. fix in Rev 1.

context switch of colorgreen, colorblue do not preserve sign, must treat same as colorred (s/w fix).

ystartf write format bug, s/w to use other address/format. fix in Rev 1.

opcode=read bit not pipelined, affects loop of pio read and writes; fix in Rev 1.

fastclear doesn't work for TI vrams, only does 4b per vram (not 8). new mode bit in Rev 1.

lronly span reject for polygons doesn't autoincrement ystarti for case of block; s/w does spans

write of register after write of HOSTRW1 sometimes ignored; assume process/fab problem (lsi). s/w workaround is to write that register two or three times.

Revision 1 features added:

additional register (USER_STATUS) for reading interrupt status without resetting any bits.

Changed FB24 bit in CONFIG register to FB_TYPE for non-TI/TI column mask-fastclear mode

ystride bit added to Drawmode0, to bump YSTARTI by +/-2 at block row end, for video option.

revision register value changed to '1'.

Revision 1 bugs: (07-21-93 JEL)

Preempting a burst read from DCBDATA when the last requested byte is being transferred to REX3 will result in REX3 continually asserting GRXDLY in response to subsequent reads from DCBDATA.

The (double)word in which the last requested byte resides will be lost.

The recovery procedure involves writing to DCBRESET, and re-initiating the burst read from a point prior to where the "lost" data may be reacquired.

This failure mode only has only showed up in DMA reads from the KALEIDOSCOPE VIDEO option with an adequate software workaround.

It is not possible to resume a preempted burst read from HOSTRW, if a read from DCBDATA occurs during the preemption period. Similarly, it is not possible to resume a preempted burst read from DCBDATA, if a read from HOSTRW occurs during the preemption period. Therefore, prior to performing a read from HOSTRW or DCBDATA, in addition to making sure that the graphics pipe or the backend is not busy, the host should insure that a DMA read from DCBDATA or HOSTRW is not in progress.