SUN Graphics Board Rev C

Documentation Package

February 28, 1982

VLSI SYSTEMS INC.

Table of Content

This is the engineering documention for the SUN graphics board.
It includes:

Conventions

The logic diagrams are best understood in conjunction with the wirelist.
The wirelist has two parts.
Part one are the components listed by location and indicating
on which logic drawing they are used.
Part two are the signal nets sorted in alphabetical order
and showing all components connected to a signal,
as well as the electrical loading considerations.

Signal Definitions:

Negated logic signals, that is signals that are asserted active low,
are indicated by a backslash "\" following the signal name.

For signals with multiple meanings or synonyms,
the synonyms are listed separated by a slash "/".
If a signal has multiple functions that are exclusive of each other,
the names are listed separated by a vertical bar or "|".

For example, the signal name for a read-write signal
that is active low for write is "READ/WRITE\".
The inverted version of this signal would be "READ\/WRITE".
A clock that is asserted either in state 3 or 5 will be marked: C.S3|5.

Often a group of signals share the same property.
Such groups of signals typically share the same prefix,
separated from a suffix with ".".
For example, all signals driving the Multibus are prefixed with "B.".

Signals that are part of busses usually share the same prefix
followed by a number. For example, the 16 data bus signals are labelled
"D0", "D1", "D2", and so on up to "D15".

Clock signals names typically contain information about
their nature as part of their signal name.
Periodic clocks are labelled C##.##-##,
where the first number is the clock period, the second number
the beginning of the active clock phase, and the third number
the end of the active clock phase.

Component Labels

Part Numbers consist of one letter followed by three digits.
The letter indicates the type of component and is one of:

C	discrete Capacitor
J	Jumper of Connector
R	discrete Resistor
S	single-in-line component
U	dual-in-line component

The three digits give the approximate component position on the board,

with the first digit indicating the row position and the last two digits numbering the position along each row.

GENERIC	QTY	BRAND	PART NUMBER	DESCRIPTION
25LS09	2	AMD	AM25LS09PC	FOUR-BIT REGISTER, INPUT MUX
3622	3	SIG	N82S131	512-BY-4 BIPOLAR PROM
4164	16	ANY	4164	64K-BY-1 DYNAMIC RAM 150 NSEC
74LS393	3	TI	SN74LS393N	DUAL 4-BIT BINARY COUNTER
74LS04	1	TI	SN74LS04N	HEX INVERTER
74LS138	1	TI	SN74LS138N	3-TO-8 DECODER
74LS145	1	TI	SN74LS145N	BDC-TO-DECIMAL DECODER
74LS244	4	TI	SN74LS244N	OCTAL NONINVERTED BUFFERS
74LS251	16	TI	SN74LS251N	1-OF-8 DATA SELECTOR
74LS273	1	TI	SN74LS273N	OCTAL D-TYPE FLIPFLOP WITH CLEAR
74LS374	15	TI	SN74LS374N	OCTAL REGISTER
74LS533	6	AMD	SN74LS533N	OCTAL TRANSPARENT LATCH INVERTING
74LS670	6	TI	SN74LS670N	4-BY4 REGISTER FILES
74LS74	1	TI	SN74LS74	DUAL D-TYPE FLIPFLOPS
74S00	2	TI	SN74S00N	QUAD 2-INPUT NAND GATES
74S02	1	TI	SN74S02N	QUAD 2-INPUT NOR GATES
74S139	2	TI	SN74S139N	DUAL 2-TO-4 LINE DECODER
74S240	1	TI	SN74S240N	OCTAL INVERTING BUFFER
74S283	2	TI	SN74S283N	4-BIT ADDER
74S288	1	TI	TBP18S030N	32-BY-8 BIPOLAR PROM
74S288	1	TI	TBP18S030N	32-BY-8 BIPOLAR PROM
74S299	2	TI	SN74S299N	8-BIT UNIVERSAL SHIFT REGISTER
74S37	1	TI	SN74S37N	QUAD 2-INPUT NAND BUFFERS
74S374	4	TI	SN74S374N	OCTAL D-TYPE LATCHES
74S472	2	TI	TBP28S42N	512-BY-8 BIPOLAR PROM
74S74	2	TI	SN74S74N	DUAL D-TYPE FLIPFLOP
AM25S09	1	AMD	AM25S09PC	FOUR-BIT REGISTER, INPUT MUX
AM25S10	8	AMD	AM25S10PC	SCHOTTKY FOUR-BIT SHIFTER,
C	38	AVX	MD015C104MAA	DIPGUARD CAPACITORS 0.1 UF
DIPSW	1	CUTLER	SM-2AV-951-8	DIPSWITCH WITH 8 POSITIONS
DS1648	4	NAT	DS1648	QUAD MEMORY DRIVER
K1114A	1	MOTOROL	K1114A	CRYSTAL OSCILLATOR 31.25 MHZ
R9.SIP	2	BURNS	4310R-101-103	RESISTOR SIP, 9 RESISTORS, 1K

LOC	DIPTYPE	BODY	FILE	POS
C5	C	C	G6	D1
C101	C	C	G6	D1
C103	C	C	G6	D2
C106	C	C	G6	D2
C111	C	C	G6	D1
C112	C	C	G6	D2
C113	C	C	G6	D2
C114	C	C	G6	D3
C115	C	C	G6	D3
C116	C	C	G6	D4
C117	C	C	G6	D4
C201	C	C	G6	D2
C206	C	C	G6	D2
C301	C	C	G6	D3
C303	C	C	G6	D3
C306	C	C	G6	D3
C311	C	C	G6	D5
C312	C	C	G6	D5
C313	C	C	G6	D6
C314	C	C	G6	D6
C315	C	C	G6	D6
C316	C	C	G6	D7
C317	C	C	G6	D7
C401	C	C	G6	D4
C406	C	C	G6	D3
C501	C	C	G6	D4
C517	C	C	G6	D1
C601	C	C	G6	D5
C606	C	C	G6	D4
C617	C	C	G6	D2
C701	C	C	G6	D5
C706	C	C	G6	D4
C806	C	C	G6	D5
C811	C	C	G6	D2
C813	C	C	G6	D3
C815	C	C	G6	D3
C917	C	C	G6	D4
J1	J.10	J.10	G7	A7
J2	J.10	J.10	G7	C7
J3	J.4	J.4	G3	D4
K101	C	C	G3	B6
K102	C	C	G3	B5
K901	C	C	G6	D1
K902	C	C	G6	D2
R106	R	R	G4	A7
R107	R	R	G4	A7
R108	R9.SIP	R9.SIP	G4	C5
R308	R9.SIP	R9.SIP	G4	D5

U3	74S37	74S37	G3	A7
			G3	A7
			G3	A7
			G3	A7
U4	74LS374	74LS374	G3	C5
U101	7660	7660	G3	B5
U102	10124Q	10124	G3	B7
U104	3622	3622	G3	C4
U105	3622	3622	G3	C4
U106	74S374	74S374	G4	A7
U107	74S288	74S288	G4	A6
U108	74S472	74S472	G4	C1
U109	74LS374	74LS374	G4	C3
U110	4164	4164	G1	A1
U111	4164	4164	G1	A2
U112	4164	4164	G1	A3
U113	4164	4164	G1	A4
U114	4164	4164	G1	A5
U115	4164	4164	G1	A6
U116	4164	4164	G1	A7
U117	4164	4164	G1	A8
U202	74S00	74S00	G6	C5
			G6	C6
			G3	D3
			G3	B2
U203	74 393	74LS393	G3	D2
U204	74 393	74LS393	G3	C2
U205	74 393	74LS393	G3	C2
U210	74LS251	74LS251	G1	B1
U211	74LS251	74LS251	G1	B2
U212	74LS251	74LS251	G1	B3
U213	74LS251	74LS251	G1	B4
U214	74LS251	74LS251	G1	B5
U215	74LS251	74LS251	G1	B6
U216	74LS251	74LS251	G1	B7
U217	74LS251	74LS251	G1	B8
U301	K1114A	K1114A	G3	D6
U302	AM25S09	AM25S09	G4	A4
U303	74LS74	74LS74	G6	D6
			G3	D4
U304	DS1648	DS1648	G3	C7
U305	DS1648	DS1648	G3	C7
U306	74S374	74S374	G4	B7
U307	74S288	74S288	G4	B6
U308	74S472	74S472	G4	D1
U309	74LS374	74LS374	G4	D3
U310	4164	4164	G1	C1
U311	4164	4164	G1	C2
U312	4164	4164	G1	C3
U313	4164	4164	G1	C4
U314	4164	4164	G1	C5
U315	4164	4164	G1	C6
U316	4164	4164	G1	C7
U317	4164	4164	G1	C8
U402	74S74	74S74	G6	C6

			G3	D8	
U403	74S139	74S139	G2	C8	
			G2	D8	
U404	DS1648	DS1648	G4	D7	
U405	DS1648	DS1648	G4	C7	
U410	74LS251	74LS251	G1	D1	
U411	74LS251	74LS251	G1	D2	
U412	74LS251	74LS251	G1	D3	
U413	74LS251	74LS251	G1	D4	
U414	74LS251	74LS251	G1	D5	
U415	74LS251	74LS251	G1	D6	
U416	74LS251	74LS251	G1	D7	
U417	74LS251	74LS251	G1	D8	
U502	74S139	74S139	G6	B8	
			G6	B8	
U503	74S374	74S374	G4	B7	
U505	25LS09	AM25LS09		G4	B1
U506	74LS273	74LS273	G6	C5	
U507	74LS374	74LS374	G2	D5	
U508	25LS09	AM25LS09		G4	B4
U509	74LS374	74LS374	G2	C5	
U510	74LS374	74LS374	G2	A3	
U512	74LS374	74LS374	G2	A3	
U514	74LS374	74LS374	G2	B3	
U516	74LS374	74LS374	G2	C3	
U601	74LS04	74LS04	G4	D3	
			G4	D6	
			G4	D7	
			G6	A4	
			G6	A4	
			G6	A5	
U602	74S74	74S74	G6	B6	
			G6	B6	
U603	74S283	74S283	G4	B6	
U604	74S283	74S283	G4	C6	
U605	3622	3622	G4	B4	
U610	74LS244	74LS244	G2	A8	
U612	74LS244	74LS244	G2	A8	
U614	74LS244	74LS244	G2	B8	
U616	74LS244	74LS244	G2	C8	
U701	74S00	74S00	G4	A6	
			G6	A3	
			G6	A5	
			G6	D7	
U702	74S02	74S02\	G6	A2	
		74S02	G6	A1	
		74S02\	G6	A2	
			G3	B3	
U703	74LS670	74LS670	G5	B6	
U704	74LS670	74LS670	G5	B6	
U705	74LS670	74LS670	G5	A6	
U706	74LS374	74LS374	G3	A2	
U707	74S299	74S299	G3	A3	
U708	74S299	74S299	G3	A3	
U709	74LS374	74LS374	G3	A2	

U710	AM25S10	AM25S10	G2	A6
U712	AM25S10	AM25S10	G2	A6
U714	AM25S10	AM25S10	G2	B6
U716	AM25S10	AM25S10	G2	B6
U800	J.16	J.16	G6	C3
U801	DIPSW	DIPSW	G6	C3
U802	74S240	74S240	G6	A7
U803	74LS670	74LS670	G5	D6
U804	74LS670	74LS670	G5	D6
U805	74LS670	74LS670	G5	C6
U810	AM25S10	AM25S10	G2	A4
U812	AM25S10	AM25S10	G2	A4
U814	AM25S10	AM25S10	G2	B4
U816	AM25S10	AM25S10	G2	B4
U901	74LS138	74LS138	G6	C2
U902	74S374	74S374	G4	A1
U903	74LS533	74LS533	G5	C2
U904	74LS145	74LS145	G6	C8
U905	74LS533	74LS533	G5	B2
U906	74LS533	74LS533	G2	D3
U907	74LS533	74LS533	G2	D1
U908	74LS533	74LS533	G2	C3
U909	74LS533	74LS533	G2	C1
U910	74LS374	74LS374	G2	A1
U912	74LS374	74LS374	G2	A1
U914	74LS374	74LS374	G2	B1
U916	74LS374	74LS374	G2	C1
P	CON			

Mnemonic	Description
A0..A16	on-board latched address
B.A0\..A19\	Multibus address lines
B.AACK\	UNUSED, Multibus advanced acknowledge
B.BCLK\	UNUSED, Multibus bus clock
B.BHEN\	UNUSED, Multibus byte high enable
B.BPRN\	UNUSED, Multibus priority in
B.BPRO\	UNUSED, Multibus priority out
B.BREQ\	UNUSED, Multibus bus request
B.BUSY\	UNUSED, Multibus busy
B.CBRQ\	UNUSED, Multibus common bus request
B.CCLK\	UNUSED, Multibus constant clock
B.D0\..D15\	Multibus data lines
B.INH1\..2\	UNUSED, Multibus inhibit lines
B.INIT\	Multibus init
B.INT0\..INT7\	Multibus interrupt request
B.INTA\	UNUSED Multibus interrupt acknowledge
B.IORC\	UNUSED Multibus I/O read control
B.IOWC\	UNUSED Multibus I/O write control
B.MRDC\	Multibus memory read control
B.MWTC\	Multibus memory write control
B.XACK\	Multibus trasfer acknowledge
BUSREAD	READ*BUSSEL
BUSSEL	Bus Select
C.S0	Clock State 0
C.S3	Clock State 3
C.S7	Clock State 7
C32.0-16	Pixel Clock
C64.0-32	System Clock
CAS0	Column-Address-Strobe 0
CAS1	Column-Address-Strobe 1
CYCLE0	Cycle Type 0
CYCLE1	Cycle Type 1
D0..D15	Data Bus
DISPEN	Display Enable
DS	Data Strobe
EN.INT	Enable Interrupts
EN.VIDEO	Enable Video
FUN0..FUN7	Function Register
GND	Ground
GX600..GX617	Memory Controller Internals
H0..H6	Horizontal Counter Outputs
HQ0..HQ3	Horizontal State Machine Internals
HRESET	Horizontal Counter Reset
HSYNC\	Horizontal Sync Pulse
INT0..INT2	Interrupt Level Select
INT\	Interrupt
J1.HSYNC	ECL horizontal sync
J1.VIDEO	ECL video

J1.VSYNC	ECL vertical sync
J2.HSYNC	TTL horizontal sync
J2.VIDEO	TTL video
J2.VSYNC	TTL vertical sync
LD.BUF	Load Buffer Strobe
LD.PRT	Load Data Port Strobe
LD.REG\	Load Register Strobe
LD.X\	Load X-Register Strobe
LD.Y\	Load Y-Register Strobe
M.A0\..A7\	Memory Address Bus
M.CAS0\..CAS15\	Memory Column Address Strobes
M.DI0..M.DI15	Memory Data In Bus
M.RAS\	Memory Row-Address Strobe
M.WE\	Memory Write Enable Strobe
MASK0..MASK15	Mask Register
OE.PRT\	Output Enable Data Port
OE.SRX\	Output Enable Shifter Buffer
PU	Pullup
PU1	Pullup 1
RASADRS	Row Address Select
RASTEROP	Raster Operation Bit
RAS	Row-Address-Strobe
RDSET0..RDSET1	Read-Set Select
READ	Read Strobe
READOP	Read Operation
READREQ	Read Request
REFRESH	Refresh Operation
REGSEL0..1	Register Select
REQ	Request
SD0..SD15	Shift Data
SETREQ\	Set Request
SHIFT0..3	Shift Amount
SRC0..15	Source Register
SRS0..15	Source Register Shifter Internal
SRX0..15	Source Register Shifter
STATE0..3	State
V0..10	Vertical Counter Outputs
VEND	Vertical Counter End
VBLANK	Vertical Blank
VCC	+5V
VCLOCK\	Vertical Counter Clock
VCLOCK1\	Vertical Counter Clock delayed
VEE	-5V
VIDEO	Video Data
VINT	Vertical Interrupt
VODD	Vertical Odd (Interlace Toggle)
VQ0..3	Vertical Decode PROM Internals
VRESET	Vertical Reset
VSYNC	Vertical Sync
W0..3	Width
WE.CTL\	Write Enable Control Register
WE.FUN\	Write Enable Function Register
WE.INT\	Write Enable Interrupt Flag
WE.MSK\	Write Enable Mask Register
WE.RAM\	Write Enable RAM

WE.SRC\	Write Enable Source Register
WE.STATUS\	Write Enable Status Register
WE.WIDTH\	Write Enable Width Register
WE	Write Enable
WIDTH0..3	Width Register
WRITE	Write
WRITEOP	Write Operation
WRITEX	Write Latched
X0..9	X-Address
XACK	Transfer Acknowledge
XCAS0..15	Cas Decoder Internals
XS0..3	X-Address of Source
XX4..9	X-Address after adder
XXX4..9	X-Address after adder latched
Y0..9	Y-Address

Decoding of Multibus Interface

D0..D15 New 16-bit data, read or write

A1..10 New X or Y address

A11 0 -> X, 1 -> Y

A12..13 Selects one of the four sets of (x,y) registers

A14..15 Select data register to be updated
On @i(read cycles), data is supplied from frame buffer
On @i(write cycles), data is supplied from processor

0 -> No Register
1 -> Control Registers
2 -> Source Register
3 -> Mask Register

Control register are further decoded with A1..A2 as follows:

0 -> Function Register
1 -> Width Register
2 -> Control Registers
3 -> Interrupt Acknowledge

A16 Enable raster operation on frame buffer

0 -> no update operation
1 -> execute update operation

A17..19 Module Select

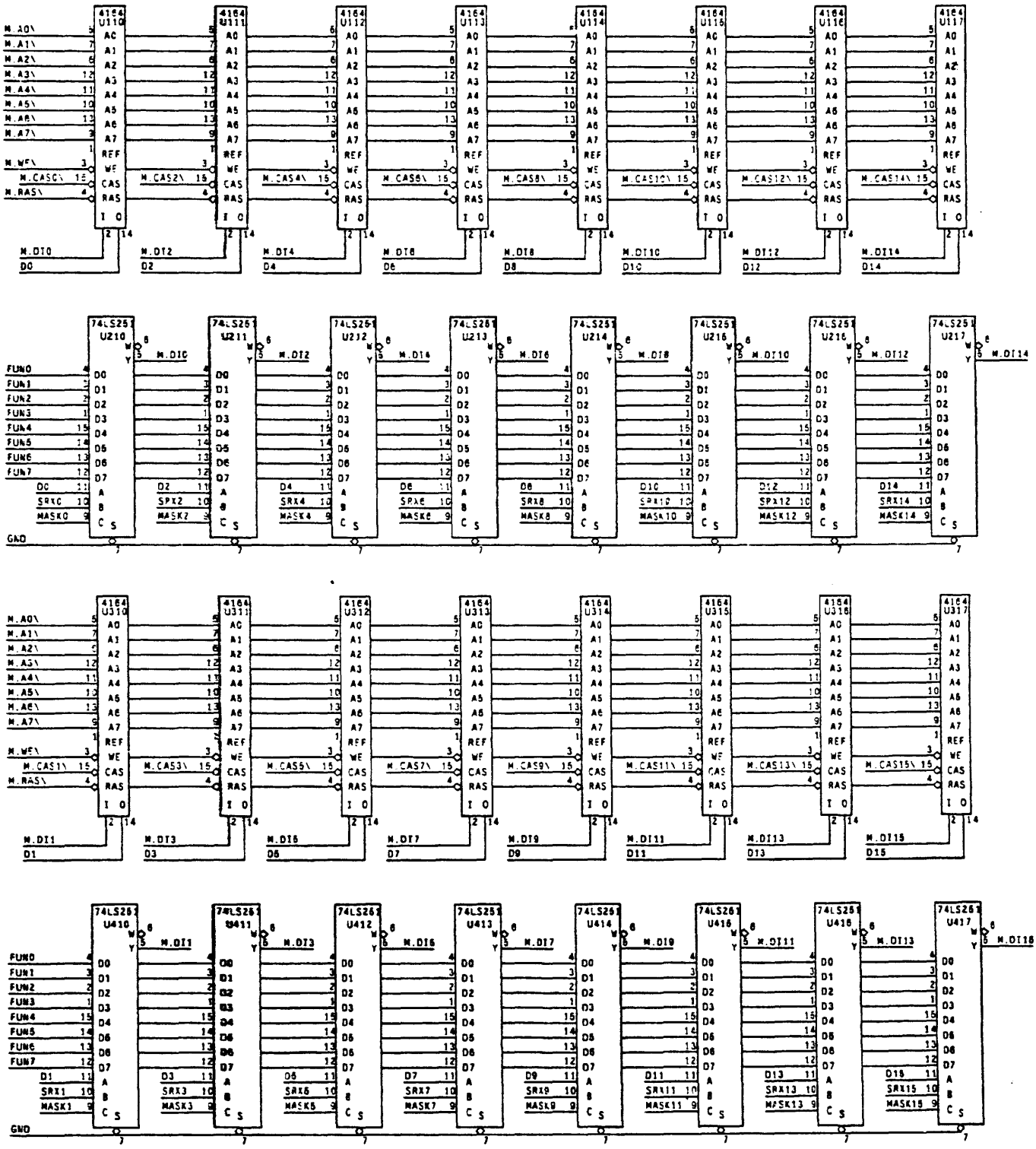
MEMR IEEE-796 Read Strobe

MEMW IEEE-796 Write Strobe

XACK IEEE-796 Acknowledge

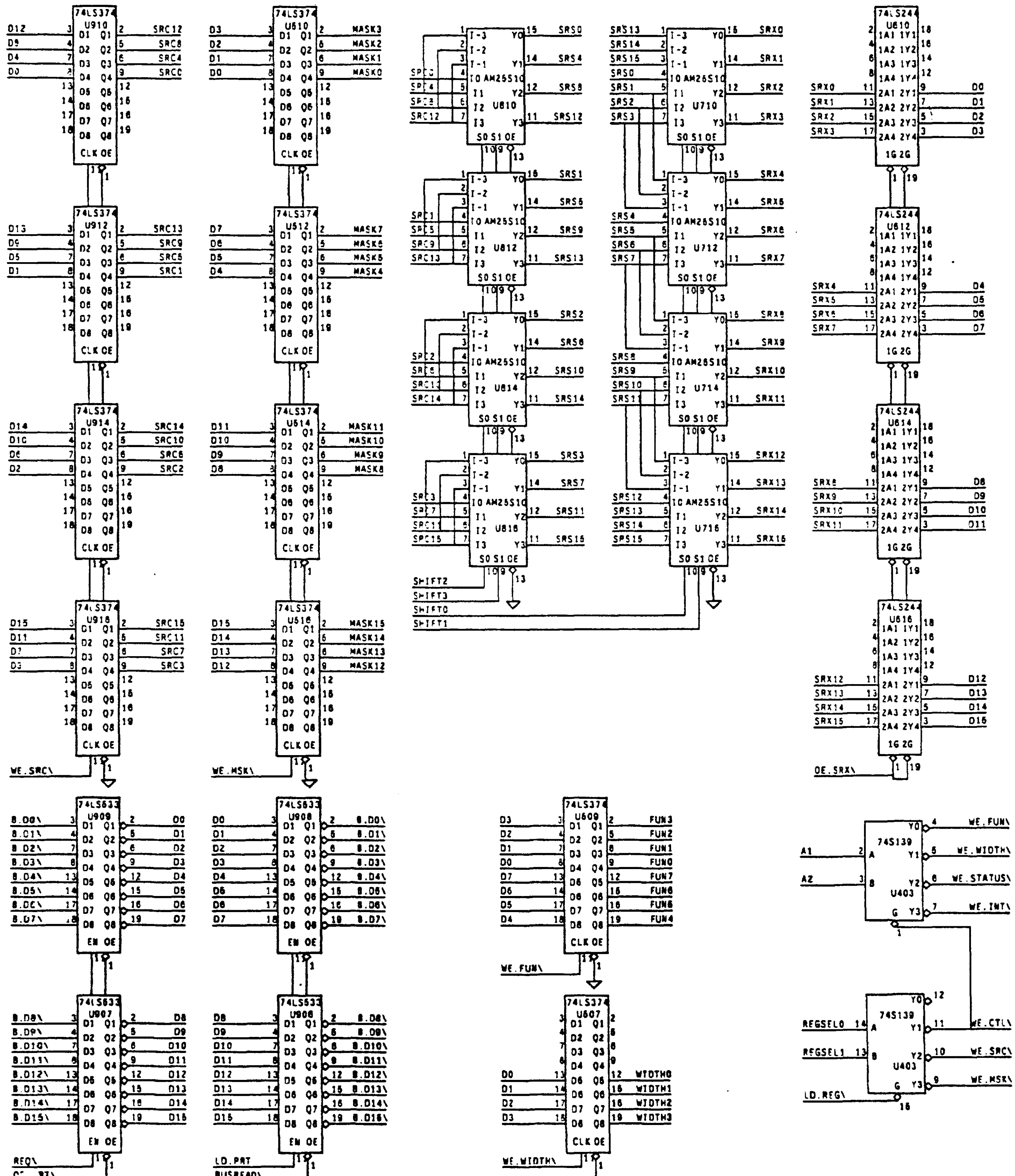
INT0..7 Interrupt request. Priority level selectable in software.

INIT Initialization, clears control register.



TITLE	FILE	DATE	PAGE	OF
(C) 1981 VSI. SUN GRAPHICS, FRAME BUFFER,	G1[SUNAVB]	10-Dec-81 21:47	1	7

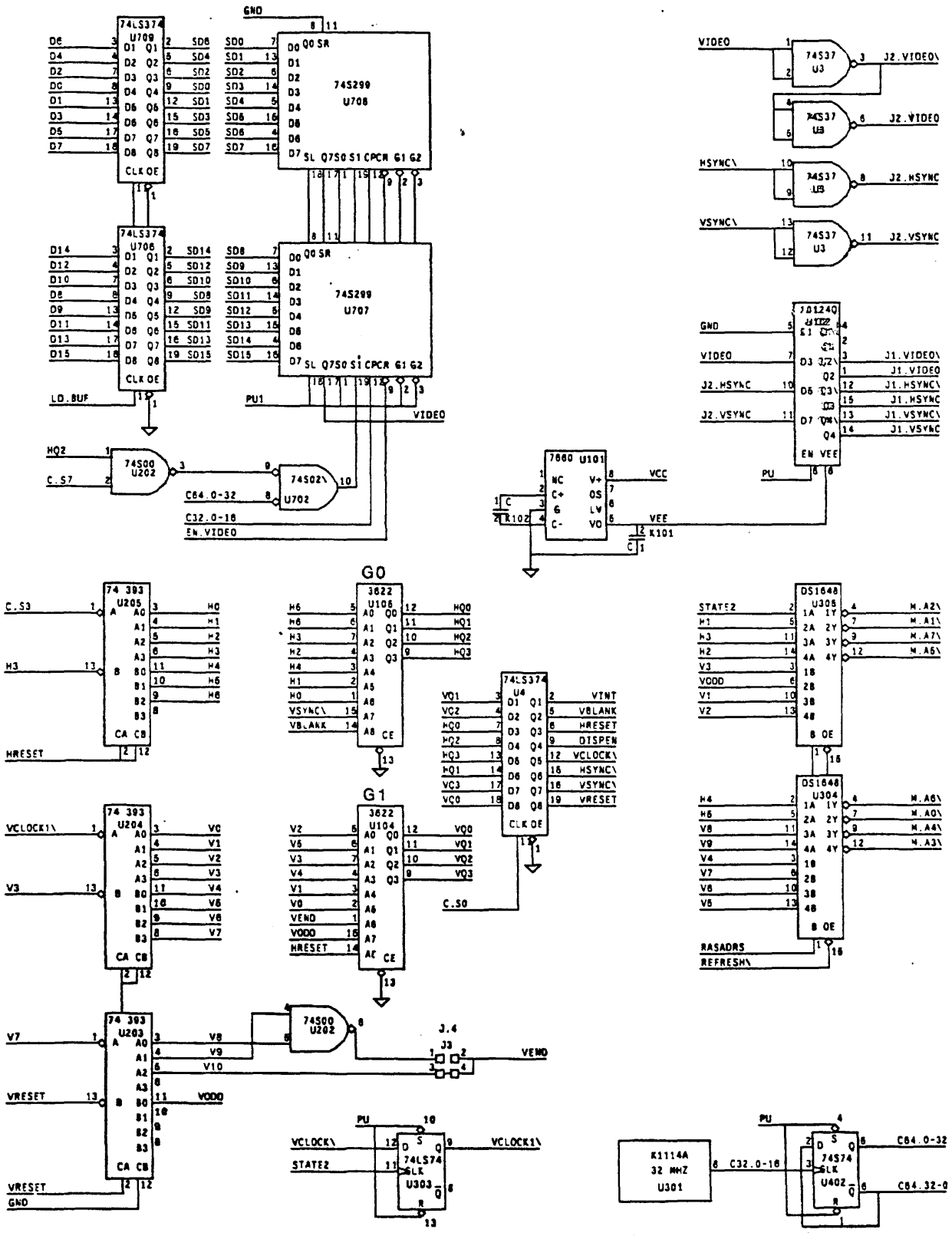
COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305



SUN GRAPHICS, DATA PATHS.

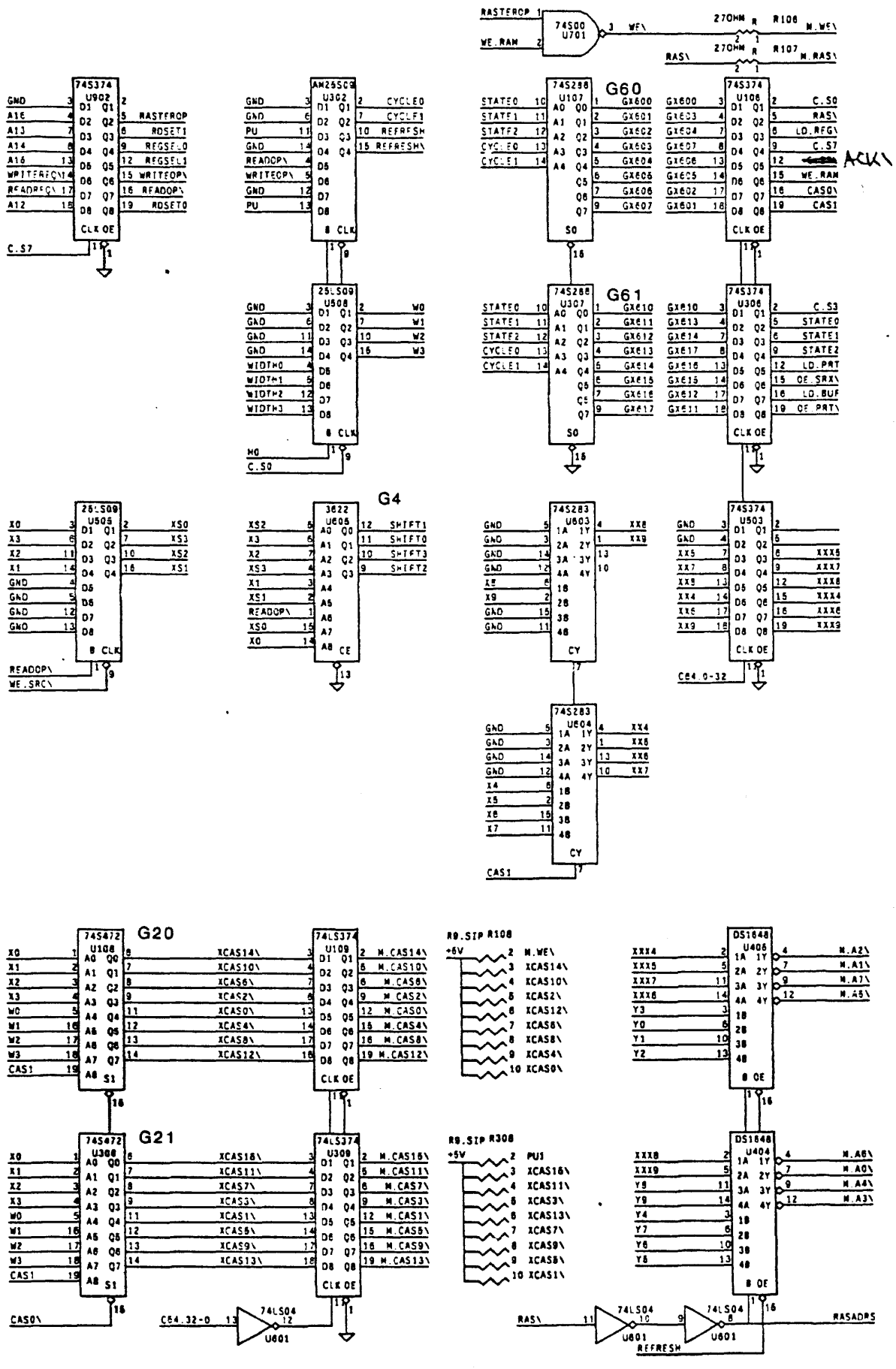
FILE	DATE	PAGE	OF
G2[SUNAVB]	10-Dec-81 21:48	2	7

T, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305



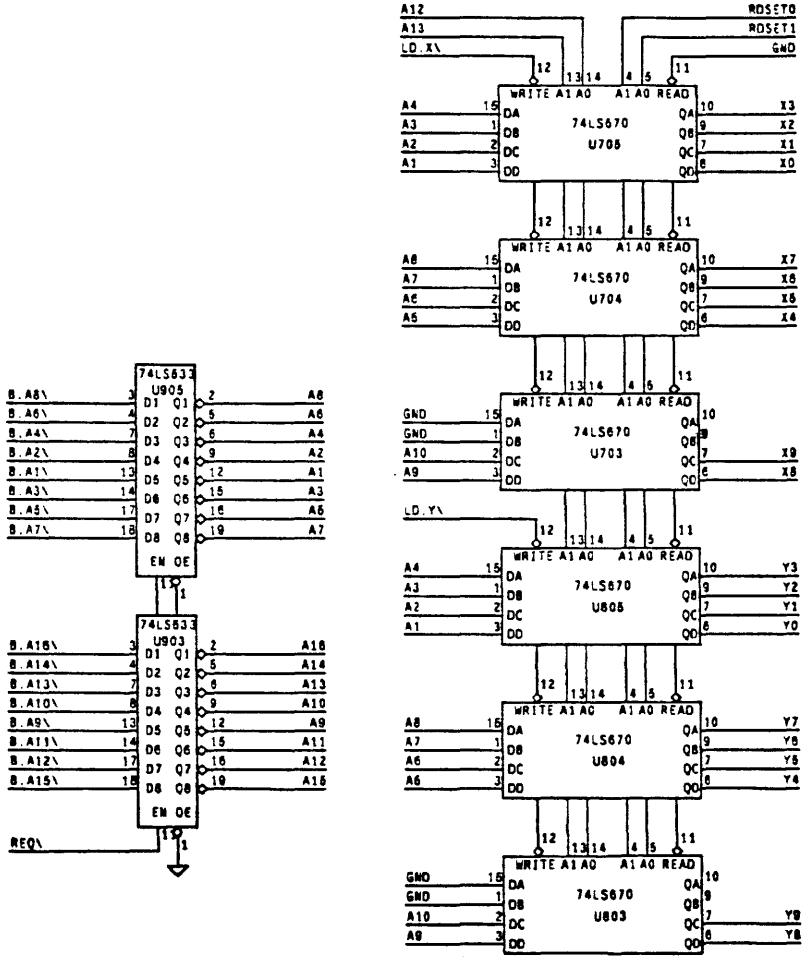
TITLE	FILE	DATE	PAGE	OF
(C) 1981 VSI. SUN GRAPHICS, VIDEO CONTROLLER,	G3[SUNAVB]	15-Dec-81 22:18	3	7

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305



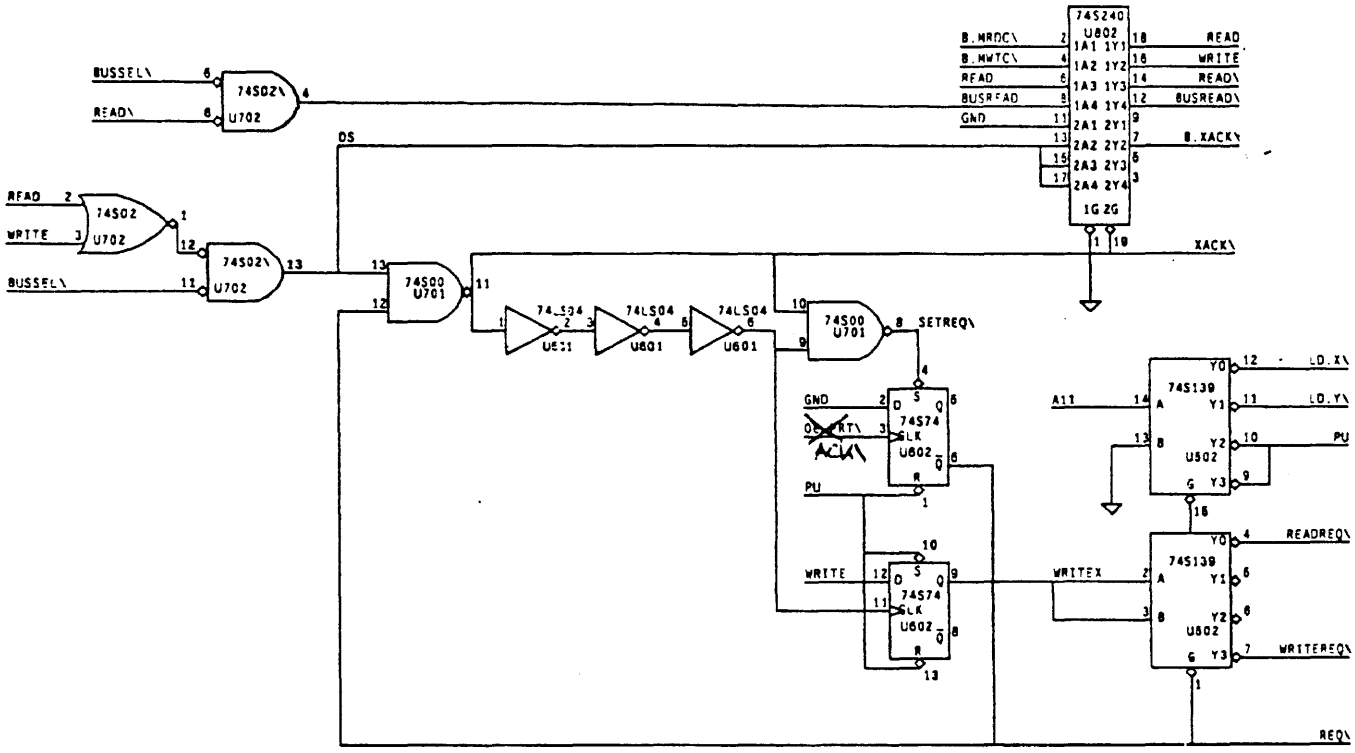
TITLE	FILE	DATE	PAGE
(C) 1981 VSI. SUN GRAPHICS, MEMORY CONTROL,	G4 [SUNAVB]	10-Dec-81 21:50	4 of 7

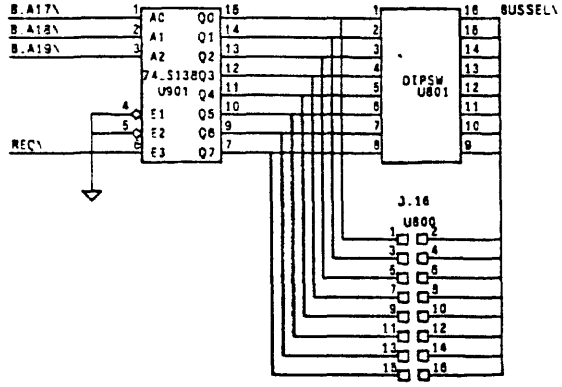
COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305

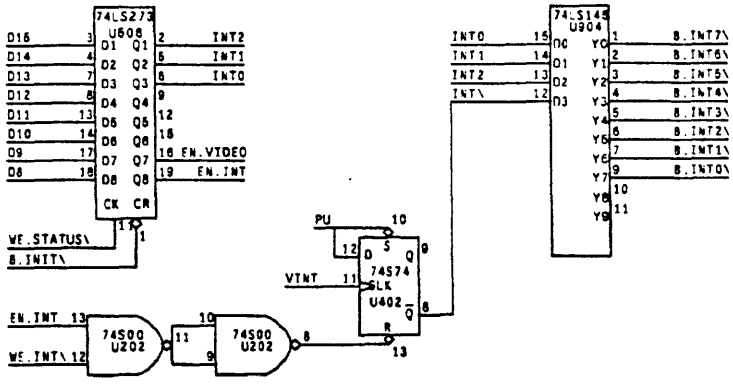


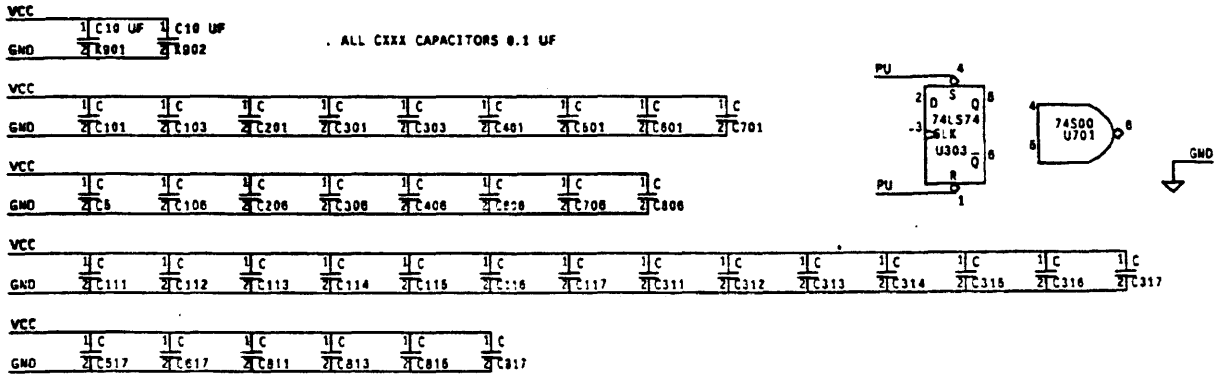
TITLE	FILE	DATE	PAGE	OF
(C) 1981 VSI, SUN GRAPHICS, XY REGISTER,	G5[SUNAVB]	10-Dec-81 21:51	5	7

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305



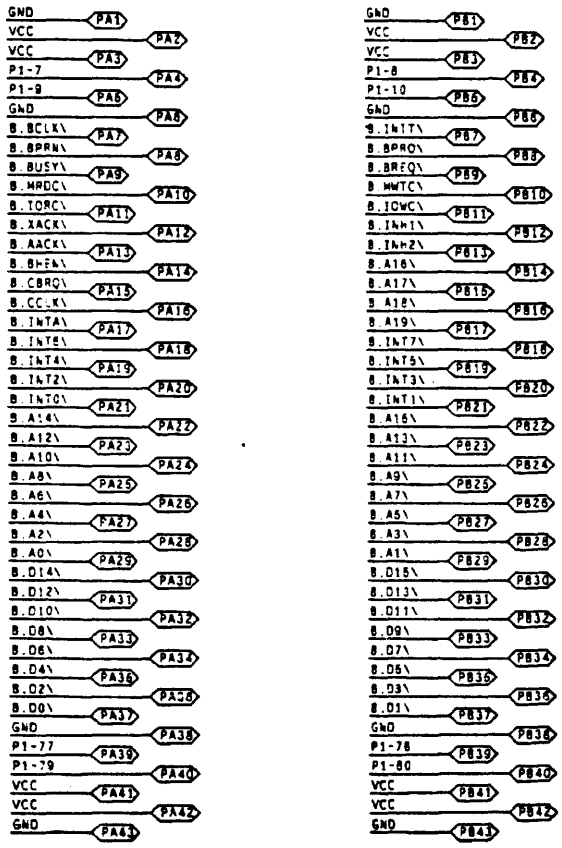


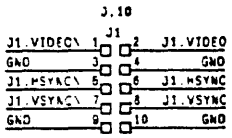


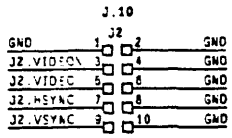


TITLE	FILE	DATE	PAGE	OF
(C) 1981 VSI. SUN GRAPHICS, MULTIBUS INTERFACE	G6[SUNAVB]	15-Dec-81 22:16	6	7

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94306







TITLE	FILE	DATE	PAGE	OF
(C) 1981 VSI. SUN GRAPHICS, CONNECTORS,	G7[SUNAVB	15-Dec-81 22:45	7	7

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305

```
comment This information proprietary to VLSI SYSTEMS INC.;
begin "prmgen"
require "prom.sai" sourcefile;
$512;

define

h5      =[a0],
h6      =[a1],
h3      =[a2],
h2      =[a3],
h4      =[a4],
h1      =[a5],
h0      =[a6],
vsync   =[a7],
vblank  =[a8],


h       =[(cvb(h0)*d0
         + cvb(h1)*d1
         + cvb(h2)*d2
         + cvb(h3)*d3
         + cvb(h4)*d4
         + cvb(h5)*d5
         + cvb(h6)*d6)],
hsync   =[(53 ≤ h < 59)],
dispen  =[((1 ≤ h < 51) ∧ ¬vblank)],
hreset  =[(h = 63)],
vclock  =[((h = 24) ∨ (h = 56))];

prombegin

prom(0,d0,      hreset);
prom(0,d1,      ¬hsync);
prom(0,d2,      dispen);
prom(0,d3,      ¬vclock);

promend;
writeprom("g0",0);
end;
```

```
:100000000A0E0A0A0E0E0A0A0E0E0A0A0E0E0A0A34
:100010000E0E0A0A06000A0A0E0A0A0A0E0A0A0A3E
:100020000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0A10
:100030000E0E0A0A0E080A0A0E080A0A0E0A0A0A10
:100040000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0AF0
:100050000E0E0A0A0E080A0A0E080A0A0E0A0A0AF0
:100060000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0AD0
:100070000E0A0A0A0E0A0A0A0E080A0A0E0B0A0AD1
:100080000A0E0A0A0E0E0A0A0E0E0A0A0E0E0A0AB4
:100090000E0E0A0A06000A0A0E0A0A0A0E0A0A0ABE
:1000A0000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0A90
:1000B0000E0E0A0A0E080A0A0E080A0A0E0A0A0A90
:1000C0000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0A70
:1000D0000E0E0A0A0E080A0A0E080A0A0E0A0A0A70
:1000E0000E0E0A0A0E0E0A0A0E0E0A0A0E0E0A0A50
:1000F0000E0A0A0A0E0A0A0A0E080A0A0E0B0A0A51
:100100000A0A0A0A0A0A0A0A0A0A0A0A0A0A0A4F
:100110000A0A0A0A02000A0A0A0A0A0A0A0A0A51
:100120000A0A0A0A0A0A0A0A0A0A0A0A0A0A0A2F
:100130000A0A0A0A0A080A0A0A080A0A0A0A0A23
:100140000A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0F
:100150000A0A0A0A0A080A0A0A080A0A0A0A0A03
:100160000A0A0A0A0A0A0A0A0A0A0A0A0A0A0AEF
:100170000A0A0A0A0A0A0A0A0A080A0A0A0B0AE0
:100180000A0A0A0A0A0A0A0A0A0A0A0A0A0A0ACF
:100190000A0A0A0A02000A0A0A0A0A0A0A0A0AD1
:1001A0000A0A0A0A0A0A0A0A0A0A0A0A0A0A0AAF
:1001B0000A0A0A0A0A080A0A0A080A0A0A0A0AA3
:1001C0000A0A0A0A0A0A0A0A0A0A0A0A0A0A0A8F
:1001D0000A0A0A0A0A080A0A0A080A0A0A0A0A83
:1001E0000A0A0A0A0A0A0A0A0A0A0A0A0A0A0A6F
:1001F0000A0A0A0A0A0A0A0A080A0A0A0B0A0A60
:0000000000
```

```
comment This information proprietary to VLSI SYSTEMS INC.;
begin "prmgen"
require "prom.sai" sourcefile;
$512;

define

v2      =[a0],
v5      =[a1],
v3      =[a2],
v4      =[a3],
v1      =[a4],
v0      =[a5],
v10     =[a6],
vodd    =[a7],
hreset  =[a8],

adrs    =[(cvb(v0)*d0
         + cvb(v1)*d1
         + cvb(v2)*d2
         + cvb(v3)*d3
         + cvb(v4)*d4
         + cvb(v5)*d5
         + cvb(v10)*d10)],
vsync   =[(vodd ∧ (1030≤adrs<1036) ∨ ¬vodd ∧ (1031≤adrs<1037))],
vblank  =[(adrs ≥ 1024)],
vreset  =[(hreset ∧ v0
          ∨ vodd ∧ (adrs = 1076)
          ∨ ¬vodd ∧ (adrs = 1078))],
vint    =[(adrs = 1024)];

prombegin

prom(0,d0,      vreset);
prom(0,d1,      vint);
prom(0,d2,      vblank);
prom(0,d3,      ¬vsync);

promend;
writeprom("g1",0);
end;
```

```
:10000000080808080808080808080808080808080870
:10001000080808080808080808080808080808080860
:10002000080808080808080808080808080808080850
:10003000080808080808080808080808080808080840
:100040000E0C0C0C04040C0C0C0C0C0C0C0C0C0CFE
:100050000C0C0C0C040C0C0C0C0C0C0D0C0C0C0CE7
:100060000C0C0C0C040C0C0C0C0C0C0C0C0C0C0CD8
:100070000C040C0C040C0C0C0C0C0C0C0C0C0C0CD0
:10008000080808080808080808080808080808080F0
:10009000080808080808080808080808080808080E0
:1000A000080808080808080808080808080808080D0
:1000B000080808080808080808080808080808080C0
:1000C0000E0C0C0C040C0C0C0C0C0C0D0C0C0C0C75
:1000D0000C040C0C040C0C0C0C0C0C0C0C0C0C0C70
:1000E0000C0C0C0C040C0C0C0C0C0C0C0C0C0C0C58
:1000F0000C040C0C040C0C0C0C0C0C0C0C0C0C0C50
:10010000080808080808080808080808080808080F
:10011000080808080808080808080808080808085F
:10012000090909090909090909090909090909093F
:10013000090909090909090909090909090909092F
:100140000E0C0C0C04040C0C0C0C0C0C0C0C0C0CFD
:100150000C0C0C0C040C0C0C0C0C0C0D0C0C0C0CE6
:100160000D0D0D0D050D0D0D0D0D0D0D0D0D0D0DC7
:100170000D050D0D050D0D0D0D0D0D0D0D0D0D0DBF
:10018000080808080808080808080808080808080EF
:10019000080808080808080808080808080808080DF
:1001A000090909090909090909090909090909090BF
:1001B000090909090909090909090909090909090AF
:1001C0000E0C0C0C040C0C0C0C0C0C0D0C0C0C0C74
:1001D0000C040C0C040C0C0C0C0C0C0C0C0C0C0C6F
:1001E0000D0D0D0D050D0D0D0D0D0D0D0D0D0D0D47
:1001F0000D050D0D050D0D0D0D0D0D0D0D0D0D0D3F
:0000000000
```

```
comment This information proprietary to VLSI SYSTEMS INC.;
begin "prmgen"
require "prom.sai" sourcefile;
$512;

define

x0      =[a0],
x1      =[a1],
x2      =[a2],
x3      =[a3],
w0      =[a4],
w1      =[a5],
w2      =[a6],
w3      =[a7],
cas1    =[a8],

x       =[(cvb(x0)*d0 + cvb(x1)*d1 + cvb(x2)*d2 + cvb(x3)*d3)],
w       =[(cvb(w0)*d0 + cvb(w1)*d1 + cvb(w2)*d2 + cvb(w3)*d3)],
width   =[(if w then w else 16)],
en(i)   =[(        (x ≤ (15-i) < (x + width))
          ∨ cas1 ∧ (x ≤ (15-i+16) < (x + width)))];

prombegin

prom(0,d0,      ¬en(14));
prom(0,d1,      ¬en(10));
prom(0,d2,      ¬en(6));
prom(0,d3,      ¬en(2));
prom(0,d4,      ¬en(0));
prom(0,d5,      ¬en(4));
prom(0,d6,      ¬en(8));
prom(0,d7,      ¬en(12));

prom(1,d0,      ¬en(15));
prom(1,d1,      ¬en(11));
prom(1,d2,      ¬en(7));
prom(1,d3,      ¬en(3));
prom(1,d4,      ¬en(1));
prom(1,d5,      ¬en(5));
prom(1,d6,      ¬en(9));
prom(1,d7,      ¬en(13));

promend;

writeprom("g20",0);
writeprom("g21",1);
end;
```

```
:10000000000001018181838 3C3C3C7C7E7E7EFEF26
:10001000FFFEFF7FFFFDFFBFFFFBFFDFFFF7FFEFEF
:10002000FEFE7F7FFDFDBFBFFBFBDFDFF7F7EFEFDE
:10003000FE7E7F7DFDBDBFBBFBDBDFD7F7E7EFEFCC
:100040007E7E7D7DBDBDBBBDDBDBD7D7E7E7EFEFBA
:100050007E7C7D3DBDB9BB9BDBD3D7C7E7E7EFEF28
:100060007C7C3D3DB9B99B9BD3D3C7C7E7E7EFEF96
:100070007C3C3D39B9999B93D3C3C7C7E7E7EFEF02
:100080003C3C393999999393C3C3C7C7E7E7EFEF6E
:100090003C38391999919383C3C3C7C7E7E7EFEF9A
:1000A0003838191991918383C3C3C7C7E7E7EFEFC6
:1000B000381819119181838 3C3C3C7C7E7E7EFEFEE
:1000C00018181111818383C3C3C7C7E7E7EFEF16
:1000D00018101101818383C3C3C7C7E7E7EFEF1E
:1000E00010100101818383C3C3C7C7E7E7EFEF26
:1000F00010000101818383C3C3C7C7E7E7EFEF26
:1001000000000000000000000000000000000EF
:10011000FFFEFF7FFFFDFFBFFFFBFFDFFFF7FFEFEE
:10012000FEFE7F7FFDFDBFBFFBFBDFDFF7F7EFEFDD
:10013000FE7E7F7DFDBDBFBBFBDBDFD7F7E7EFEECC
:100140007E7E7D7DBDBDBBBBDBDBD7D7E7E7EEEEBB
:100150007E7C7D3DBDB9BB9BDBD3D7C7E7E6EE6EAA
:100160007C7C3D3DB9B99B9BD3D3C7C7E6E66E6E99
:100170007C3C3D39B9999B93D3C3C7C6E6666E6C88
:100180003C3C393999999393C3C3C6C666666C6C77
:100190003C38391999919383C3C2C64666646C2C66
:1001A0003838191991918383C2C2464664642C2C55
:1001B000381819119181838 2C2424644642 42C2844
:1001C00018181111818182824242444424242828 33
:1001D00018101101818082024240440424202808 22
:1001E00010100101808002024040040420200808 11
:1001F00010000100800002004000040020000800 00
:0000000000
```

```
:10000000000010181818383C3C3C7C7E7E7EFEFFF27
:10001000FEFF7FFFFDFFBFFFFBFFDFFFF7FFEFFFEF
:10002000FE7F7FFDFDBFBFFBFBDFDFF7F7EFEFFFDD
:100030007E7F7DFDBDBFBBFBDBDFD7F7E7EFEFFFCB
:100040007E7D7DBDBDBBBBDBDBD7D7E7E7EFEFFF39
:100050007C7D3DBDB9BB9BDBD3D7C7E7E7EFEFFFA7
:100060007C3D3DB9B99B9BD3D3C7C7E7E7EFEFFF13
:100070003C3D39B9999B93D3C3C7C7E7E7EFEFFF7F
:100080003C393999999393C3C3C7C7E7E7EFEFFFAB
:1000900038391999919383C3C3C7C7E7E7EFEFFFD7
:1000A0003819199191838 3C3C7C7E7E7EFEFFFFF
:1000B0001819119181838 3C3C7C7E7E7EFEFFF27
:1000C0001811118181838 3C3C7C7E7E7EFEFFF2F
:1000D0001011018181838 3C3C7C7E7E7EFEFFF37
:1000E0001001018181838 3C3C7C7E7E7EFEFFF37
:1000F0000001018181838 3C3C7C7E7E7EFEFFF37
:10010000000000000000000000000000000000EF
:10011000FEFF7FFFFDFFBFFFFBFFDFFFF7FFEFFFEE
:10012000FE7F7FFDFDBFBFFBFBDFDFF7F7EFEFFEDD
:100130007E7F7DFDBDBFBBFBDBDFD7F7E7EFEEFECC
:100140007E7D7DBDBDBBBBDBDBD7D7E7E7EEEE7EBB
:100150007C7D3DBDB9BB9BDBD3D7C7E7E6EE6E7EAA
:100160007C3D3DB9B99B9BD3D3C7C7E6E66E6E7C99
:100170003C3D39B9999B93D3C3C7C6E6666E6C7C88
:100180003C393999999393C3C3C6C666666C6C3C77
:1001900038391999919383C3C2C64666646C2C3C66
:1001A0003819199191838 3C2C2464664642C2C3855
:1001B0001819119181838 2C242464464242C283844
:1001C0001811118181828242424444242428281833
:1001D0001011018180820242404404242028081822
:1001E0001001018080020240400404202008081011
:1001F0000001008000020040000400200008001000
:0000000000
```

```
comment This information proprietary to VLSI SYSTEMS INC.;
begin "prmgen"
require "prom.sai" sourcefile;
$512;

define

xs2     =[a0],
x3      =[a1],
x2      =[a2],
xs3     =[a3],
x1      =[a4],
xs1     =[a5],
readop_i=[a6],
xs0     =[a7],
x0      =[a8],

readop  =[(¬readop_i)],
xs      =[(cvb(xs0)*d0 + cvb(xs1)*d1 + cvb(xs2)*d2 + cvb(xs3)*d3)],
xd      =[(cvb(x0)*d0 + cvb(x1)*d1 + cvb(x2)*d2 + cvb(x3)*d3)],
shift   =[(if readop then (xs - 0) else (xs - xd))];

prombegin

prom(0,d0,      shift LAND d1);
prom(0,d1,      shift LAND d0);
prom(0,d2,      shift LAND d3);
prom(0,d3,      shift LAND d2);

promend;
writeprom("g4",0);
end;
```

Changes to SUN Graphics Boasd REV-C-811215
to upgrade it to REV-D

1) Cut Shorted trace from U402 pin 14 to 12.
Replace by wire.

2) Cut out Pad. between U601 pin 1 and U701-16
and Capacitor C601-2 and C701-1.
This pad is shorted to Ground.
Replace by wire.

3) Cut J3 pin 3-4 ~~and~~.
Inset Jumpe J3 pin 1-2.

4) Cut Trace above U305 pin 9.
on solder side. Reconnect trace above
U305 pin 9 to U106 pin 12.

5) Cut U605(1) trace.
Connect U605(1) to U302(2).

6) Cut U902(19) and U902(6).
Swap the wires connected to these pins.

High-Performance Raster Graphics for Microcomputer Systems

Andreas Bechtolsheim and Forest Baskett
Computer Systems Laboratory
Stanford University
Stanford, California 94305

A frame buffer architecture is presented that reduces the overhead of frame buffer updating by three means. First, the bit-map memory is (x,y) addressable, whereby a string of pixels can be accessed in parallel. Second, the pixel-change operation is performed by hardware in a single read-modify-write cycle. Third, multiple objects in the frame buffer are addressable simultaneously by a set of address registers. The remaining task of generating (x,y) addresses and providing new data can be managed rapidly by current microprocessors or DMA-devices.

With a modest expenditure of hardware, this architecture eliminates all the bit-shifting, bit-masking, and bit-manipulation conventionally associated with frame buffer graphics, while retaining the full generality of user-programmable control. The particular implementation described allows raster manipulation at full bit-map memory bandwidth. It can paint a 16×16 pixel character into the frame buffer in 16 microseconds and can modify a 1024×1024 pixel raster in 64 milliseconds.

Keywords: Raster-Scan Graphics, Frame Buffer, Bit-Map, Raster Operation, Functional Memory

This work has been supported in part by DARPA-MDA-903-79-C-0680, by Stanford Artificial Intelligence Laboratory, and by Stanford Linear Accelerator Center.

1. Introduction

The raster-scan graphics system described in this paper is part of a workstation being designed at Stanford University to support projects in design automation (VLSI) and advanced text processing (TEX). In brief, the workstations contains a powerful 16-bit microprocessor, a substantial main memory, and a local network interface for accessing centralized file servers or remote large-scale computing resources. Each station also supports one or more high-resolution displays.

Using a commercial 16-bit microprocessor as the main processor of the workstation, it is difficult to achieve adequate frame buffer manipulation speed. The microcomputer processing bandwidth is simply insufficient to deal with the number of bits in a high-resolution frame buffer which is organized as a conventional memory.

Our approach to this problem is to organize the frame buffer in a different way. This architecture treats frame buffer updates as a a function with five parameters:

(X, Y, Width, Operation, Data),

whereby X and Y are the address of a rectangle of size $Width \times 1$, to which the *Data* is applied with the specified *Operation* (see Figure 1).

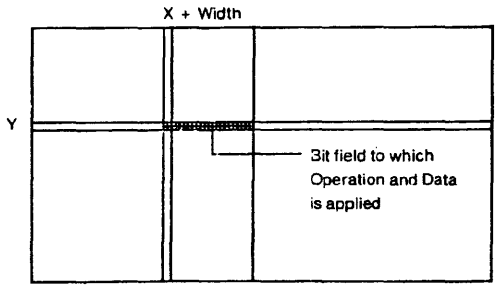


Figure 1. Functional Frame Buffer

In our implementation, *width* can range from 1 to 16 bits and the *operation* is a bit-wise boolean function. These five parameters are maintained in separate registers. X and Y are loaded by referencing

windows in the main processor's address space; *Width*, *Operation*, and *Data* are loaded from the data bus. In a single memory reference operation, the main processor then can provide a new address <x> or <y>, access a string of size <width> at location <x,y>, and perform the previously selected <operation> on the frame buffer with the new <data> provided.

This *functional* frame buffer architecture performs in hardware all bit-shifting, bit-masking, and bit-manipulation usually associated with frame buffers graphics. The only remaining task for the processor (or a DMA controller) is to generate the <x,y> addresses and transfer data to and from the frame buffer. Using the frame buffer function as a primitive, operations on rasters of arbitrary height and width can be implemented easily, such as RasterOP [1].

In our implementation, the frame buffer is a 64k by 16 bit (1024×1024) dual-port memory with sufficient bandwidth to allow one frame buffer modification (read-modify-write) cycle every microsecond. Any processor that can provide a new address and data at that rate fully utilizes the available frame buffer memory bandwidth. This means that:

--- painting a 16×16 pixel character takes 16 microseconds;
--- vectors are drawn at a rate of 1 pixel per microsecond;
--- a 1024X768 raster can be modified in 48 milliseconds;
--- scrolling a 1024×768 rectangle in any direction takes 96 milliseconds.

The speed of these operations eliminates special-purpose hardware for scrolling, vector generation, and cursor motion. Instead, a functional frame buffer architecture allows generalized manipulation of raster images in a truly interactive fashion.

2. The Problem: Bandwidth

A frame buffer display system typically consists of an application program, a frame buffer, and a video monitor (Figure 2). The frame buffer contains the image in a raster (bit-pattern) representation. Two processes work on the frame buffer, one updating the frame buffer to change the image it represents, the other refreshing the video monitor.

The bandwidth of the refresh process is proportional to the number of pixels displayed, the bits per pixel, and the refresh rate, whereas the bandwidth of the update process depends on the response-time requirements of the application. As frame buffer sizes grow because of decreasing memory costs, the bandwidth requirements will increase proportionally.

For example, a high-resolution monochrome monitor with a display area of 1024 by 768 one-bit-pixels and 60 Hz non-interlaced refresh demands a bandwidth of approximately 64 Mbit/sec; the same bandwidth is necessary for a standard video-monitor with a display area of 640 by 480, four bits per pixel, and interlaced 60 Hz refresh. The frame buffer memory has to accomodate both processes. If the bit-map memory has an overall bandwidth of, say, 80 Mbit/sec, then the refresh process leaves 16 Mbit/sec for update purposes.

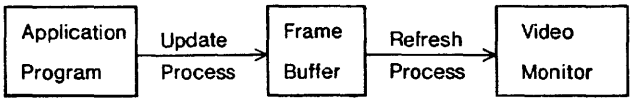


Figure 2. Model of a Frame Buffer Graphics System

The ratio of refresh over update bandwidth indicates the number of display (refresh) times necessary to modify an entire frame buffer; in our case, four refresh times or 66 milliseconds. As this ratio gets larger, the interactiveness of the display decreases, limiting the usefulness of the display. Also, a slower update uses up a larger portion of the overall system resources, depleting the processing cycles remaining for the application program and other tasks.

Ideally, one would like to update the frame buffer at the full available bit-map memory bandwidth. However, it is difficult to provide such a high update rate. Each frame buffer operation is accompanied by the overhead of generating the frame buffer address, performing the raster operation, masking bits that shall remain unchanged, and loop control. The effective bandwidth of a processor performing all of these operations thus needs to be considerably higher than the update bandwidth alone, and far exceeds the bandwidth of currently available microcomputers. The problem thus is how to achieve fast frame buffer manipulation with a processor that is limited in speed.

3. Conventional Frame Buffer Graphics Architectures

Two frame buffer architectures can be distinguished: Those in which the frame buffer is part of main memory and those where it isn't. In both cases, the frame buffer is typically a dual port memory with one port dedicated for refresh to unload the refresh bandwidth from the memory bus.

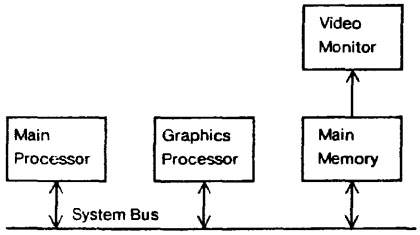


Figure 3.1. Frame Buffer in Memory

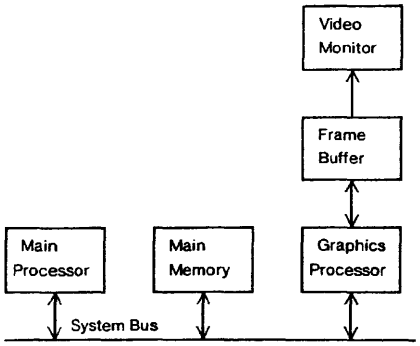


Figure 3.2. Frame Buffer Isolated

The *frame buffer in main memory* architecture (Figure 3.1) makes the frame buffer directly accessible to the main processor. However, it is almost always inefficient to manipulate the frame buffer with the main processor, except with processors that have special instructions for frame buffer manipulation, such as *BitBlt* in the Alto computer [1]. Thus one usually needs a peripheral "graphics processor" for high-speed frame buffer manipulation, which receives commands from the main processor. Because there is only a single memory bus, contention between the main processor and the graphics processor can result. Typically a significant portion of the system resources must be dedicated for frame buffer manipulation in this architecture.

The *frame buffer not in main memory* architecture isolates the frame buffer from the system by placing the frame buffer under exclusive control of a dedicated graphics processor (Figure 3.2). The graphics processor provides a standard set of frame buffer manipulation functions, receiving commands and data from the main processor.

An independent graphics subsystem has a number of advantages. First, with a sufficient rich set of graphics primitives, demands on system resources are greatly reduced. Second, it is straightforward to meet given update bandwidth requirements with a specially-designed controller. Third, a self-contained graphics subsystem can be interfaced to a variety of host computers. For all of these reasons, most commercial high-performance raster-scan systems have adopted this architecture.

A difficulty introduced by the memory partitioning is the limited memory space in the graphics subsystem. Almost all frame buffer applications require memory space in addition to the visible bit-map, to store character sets, graphical objects, and so forth. If main memory were used for this purpose, data has to be moved across the graphics processor interface, putting load back on the main processor system. This problem can partly be solved by making the frame buffer larger than the visible display area, and to use the excess, invisible part as a software controlled cache for graphical objects such as the currently used character fonts. With the cache, communication bandwidth between host and graphics processor is reduced to the remaining miss-rate.

In the context of a low-cost microcomputer system, both architectures have the disadvantage to require a graphics processor, typically a bit-slice, microprogrammable machine. Another, more subtle, problem introduced by a graphics processor is the constraint of predefined frame buffer operations. It will be difficult, if not impossible, for users to extend the graphics processor firmware. This is less a problem for well-defined applications such as vector drawing, but it limits the exploration of novel frame buffer operations (for example area fill, greyscale backgrounds, and brushes), as discussed in Newman and Sproull [1].

4. The Functional Frame Buffer Architecture

An alternative approach to speed up frame buffer updating is to provide a more appropriate representation for raster manipulation at the hardware level.

4.1. Pixel Addressing

If the bit-map memory is organized just as other memory into physical memory words, the task of mapping the pixel-addressing into physical words has to be done in software or firmware, a constant overhead for every frame buffer access. This overhead can be avoided if the frame buffer is (x,y) addressable. Since the desired frame buffer manipulation bandwidth can only be reached by working on multiple pixels in parallel, we shall access as many bits as the data-paths in the system can transfer, and have a facility to specify fewer bits if so desired. In our implementation, a 1×16 rectangle can be accessed at an arbitrary (x,y) location. An alternative solution, implemented by Sutherland and Sproull, is to access 8×8 rectangles [3].

4.2. Pixel Operation

For a one-bit per pixel frame buffer, a small set of boolean operations suffices for the actual frame buffer bit-change operation: Bit-Move, Bit-Or, Bit-And, and Bit-XOR, whereby the new data can be optionally inverted [1,2]. Executing this operation locally to the frame-buffer in a single read-modify-write memory cycle gains at least a factor of two in bandwidth over doing so in the main processor with separate read-write cycles. The source data is either supplied from the main processor or is accessed in an earlier frame buffer memory cycle.

4.3. Control

Control of frame buffer manipulation is retained by the main processor in a completely user-programmable fashion. When the raster operation and the width field have been set, the only remaining task is to provide (x,y) addresses and to transfer data to or from the frame buffer. Making the (x,y) address registers independent windows in the processors address space, rectangular graphical objects can be accessed using autoincrementing addressing modes or DMA devices.

5. Implementation

We have implemented a functional frame buffer architecture with a frame buffer size of 1024×1024 Bits or 128 kBytes. The aspect ratio can be reconfigured for a variety of visual display sizes such as 1170×896 or 1024×768. In the latter case, invisible frame buffer space can be use for font storage. Memory planes can be stacked to provide up to eight bits per pixel.

5.1. Memory Organization

The 1024×1024 bits of memory are stored in 64 16k dynamic RAMs. This memory supports two organizations: for refresh purposes it consists of 16k 64-bit physical memory words which are aligned along horizontal scan lines. For update purposes, each 64-bit physical memory word is further divided into four 16-bit logical memory words, since the data paths are only 16 Bits wide (Figure 5.1.a).

For refresh cycles, all 64 RAM chips are read out in parallel into a 64 Bit buffer, whose content is shifted out to the video monitor (Figure 5.1.b). The timing of video refresh cycles depends on video bandwidth: high-resolution non-interlaced monitors require one refresh cycle every microsecond. Note that dynamic RAM refresh is done automatically by the video refresh.

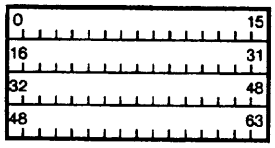


Figure 5.1.a
Memory Organization

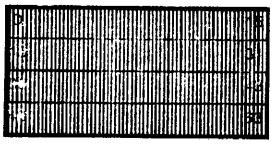


Figure 5.1.b
Refresh Cycle

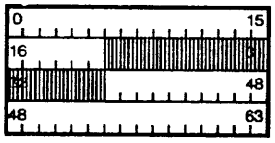


Figure 5.1.c
Intraword Access

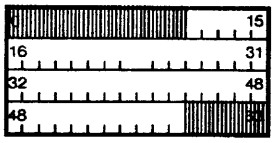


Figure 5.1.d
Interword Access

5.2. Update Cycles

Update cycles are read or read-modify-write cycles that alternate with refresh cycles. For bit-string addressing, 16 consecutive bits must be accessed starting on any bit boundary. Memory accesses can thus cross logical and physical word boundaries. The data is then aligned with a 16-bit shifter to appear normalized at the processor interface.

To access 16 bits within a physical word (Figure 5.1.c), the appropriate memory chips are selected individually by controlling the row-address-strobe of each chip a function of the bit-address. All RAM chips receive the subsequent column-address-strobe, but only those selected with RAS will be enabled. The data-out from all logical words is ORed together before it is put on the data bus. If an update crosses a physical word boundary (Figure 5.1.d), the memory chips in the next physical memory word must receive an address one higher than the other RAM chips. This is implemented by dividing the memory in two banks with separate address drivers (Figure 5.2). An adder increments the address for the lower bank by one if the access starts in the higher bank.

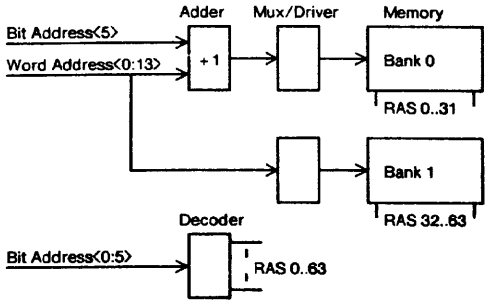


Figure 5.2. Memory Addressing

5.3. Function Unit

The function unit performs the actual bit-change operation in a single read-modify-write cycle. The bit-change operation is applied bit-wise to the source data and the old destination value to yield a new destination value. The source data can be either supplied by the main processor or by the frame buffer from a previous read cycle. The set of functions are:

Clear	Dst ← 0
Copy	Dst ← Src
Paint	Dst ← Dst OR Src
Erase	Dst ← Dst AND NOT Src
Invert	Dst ← Dst XOR Src
Clear/	Dst ← 1
Copy/	Dst ← NOT Src
Paint/	Dst ← Dst OR NOT Src
Erase/	Dst ← Dst AND Src
Invert/	Dst ← Dst XOR NOT Src

The function unit is implemented as a programmable logic array (partitioned into four chips). The function has one more input, MASK, that inhibits the modification of the corresponding destination bits, causing these bits to be rewritten with their old value. This is necessary if we desire to change fewer than 16 Bits at one time, for example when drawing vectors. The MASK is computed from the value of the WIDTH register and the destination address. The relation of the function unit to the rest of the graphic system is shown in Figure 5.3.

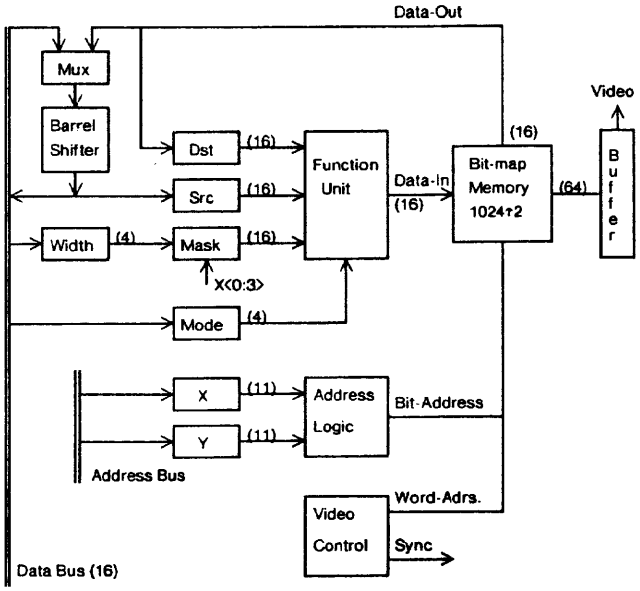


Figure 5.3. Data Paths in the Functional Frame Buffer

5.4. The Graphics Controller Interface

The processor communicates with the graphics controller through a number of registers:

Data	Data to and from the frame buffer
X	X coordinate
Y	Y coordinate
Z	Frame buffer plane
Width	Width of destination object
OP	Mode of function unit
Control	Data flow control

The X and Y registers are windows in the processor's address space, loaded as a sideeffect of reading or writing frame buffer data within these windows. All other registers are loaded explicitely from the data bus, since they are usually changed infrequently.

The processor uses *read* and *write* commands to initiate frame buffer read cycles and read-modify-write cycles. Whether the processor actually supplies or receives any data in such an operation depends on the setting of the *Control* register.

The independent registers have the advantage that unchanged parameters need not be retransmitted, thereby reducing communication bandwidth between processor and graphics controller. On the other side, the registers must be changed for accessing and operating on different graphical objects. This problem is solved with a set of registers which is large enough to hold the working set required for frame buffer operations.

Most frame buffer operations, scrolling for example, involve two objects: a destination and a source. More complicated operations can involve three objects, for example when the source is first manipulated with a grey-scale pattern. Even more sets are useful to cache state information for objects that are refererenced often. For example, a cursor typically requires a cursor symbol object, a save area object, and a current location object. For these reasons the design provides 16 complete register sets.

The details of the graphics controller interface will be usually invisible to application programs, which will interface to higher-level **graphics packages.**

6. Conclusions

High-resolution frame buffers require significant processing bandwidth for high-speed manipulation. Most current bit-map graphic systems organize the frame buffer just as ordinary memory. A functional frame buffer architecture was described that decomposes a frame buffer operation into pixel-access, pixel-operation, and control. By implementing the access and operation function in hardware, one can greatly reduce the computation needed for frame buffer manipulation. A frame buffer was implemented that allows to access a 1×16 rectangle at any (x,y) location and manipulate it with a preselected boolean function. In conjunction with a fast microprocessor or a DMA-controller, this frame buffer architecture allows raster manipulation at 32 Mbit/sec.

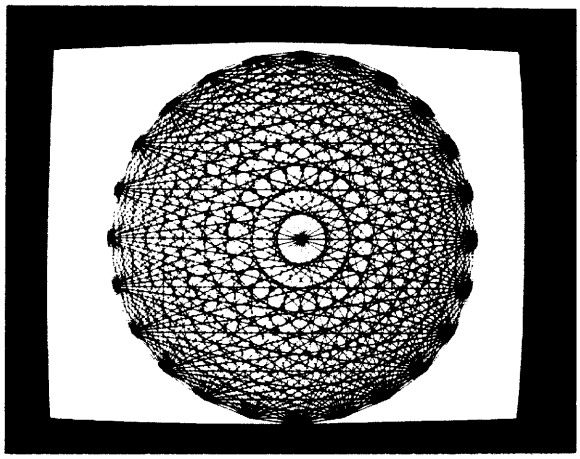


Figure 6.1. A sample 1024 by 768 image displayed on an interlaced monitor.

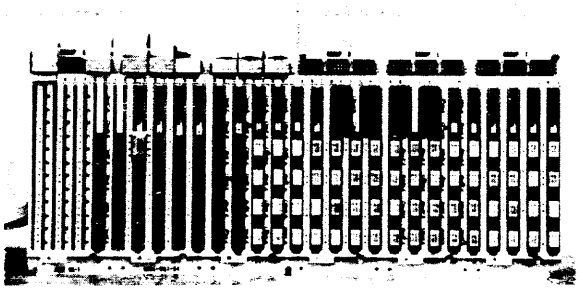


Figure 6.2. A prototype of the functional frame buffer.
The large packages on the top center are the function unit PLAs.

7. References

[1] W.M. Newman and R.F. Sproull,
Principles of Interactive Computer Graphics,
second edition, McGraw Hill, 1979.

[2] C.P. Thacker, et al.,
"Alto: A personal Computer",
in D. Siewiorek, C.G. Bell and A. Newell,
Computer Structures: Readings and Examples,
second edition, McGraw Hill, 1980.

[3] I.E. Sutherland, R.F. Sproull, S. Gupta, A. Thompson,
personal communication.

[4] R.F. Sproull
"Raster Graphics for Interactive Programming Environments",
Xerox PARC Report CSL-79-6, June 1979.