6.0 Conclusions

The SX chip provides the cost effective pixel processing capability to a traditional workstation. Its integrated architecture speeds up the memory access necessary for graphics and imaging applications. The integer vector processor architecture of the SX pixel processor provides more efficient ALU operations than can be accomplished with the CPU. The feature of being a programmable pixel processor will also allow the SX chip to easily accommodate new algorithms and new applications.

Acknowledgments

We would like to thank Tim Van Hook and Vicki Woolf for their work in designing the SX architecture. Thanks also the Andreas Bechtolsheim for allowing the project to happen.

The authors would also like thank Conan Mishler for his invaluable and transcendental efforts in making the chip, and Jasvinder Nijjar for his efforts in keeping them running.

References

1. Akeley, Kurt, Tom Jermlouk. “High performance Polygon Rendering”, Computer Graphics 22,4 (August 1988) pp 239-246.

2. Borenstein, Nathaniel. “Multimedia Electronic Mail: Will the Dream Become a Reality”, Communications of ACM 34, 4 (April 1991) pp. 117-119.

3. Cockroft, Greg and leo Hourvitz. “NeXTstep: Putting JPEG to Multiple Uses”, Communications of the ACM 34, 4 (April 1991) pp. 45-116.

4. England, Nick. “A Graphics System Architecture for Interactive Applications Specific Display Functions”, IEEE Computer Graphics and Applications 6, 1 (January 1986) pp. 60-70.

5. Kelly, Michael, Stephanie Winner, Kirk Gould. “A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm”, Computer Graphics 26, 2 (July 1992) pp 241-248.

6. McCormack, Joel, Bob McNamara, Lindsay Cage. “A Smart Frame Buffer”, Hot Chips III, (August 1991) pp 7.1-7.7.

7. Postscript Language Reference Manual, second edition. Adobe Systems Incorporated, Addison-Wesley Publishing Company, Inc. Chapter 7, pp. 325-342.

8. Weber, John. XIE, “A Proposed Standard Extension to the X11 Window System”, Proposal made to the MIT X consortium, 1991.

5.0 Hardware Performance

The below operations were preformed on prototype hardware. The results reflect guaranteed never to exceed hardware SX performance. All software overhead has been eliminated and only the SX instruction stream is measured. All the tests were performed in true color(RGB) and to Vram. Performance represents the number of operation per second while ns/pixel is the total cost of the operation in terms of nanosecond per pixel.

Operation	Performance	Ns/Pixel
Random 10 pixel RGB 2-D Vectors	528K vectors/sec	188 ns/pixel
Horizontal 10 pixel RGB 2-D Vectors	697K vectors/sec	142 ns/pixel
Vertical 10 pixel RGB 2-D Vectors	467K vectors/sec	212 ns/pixel
64x64 RGB Rectangular Raster Fills with constant color	12K rectangles/sec	20 ns/pixel
64x64 RGB Rectangular Raster Copies	4.8K rectangles/sec	50 ns/pixel
Random 100 Pixel RGB Gouraud shaded Z-buffred Triangles	21K triangles/sec	45 ns/pixel

Figure 12 - SX Performance Measurements

4.2 Sample Algorithms

This section will describe how to implement some basic pixel operations using the SX. Only the important parts of the algorithms are described; the housekeeping parts such as loops and address calculations are omitted. At the end, the SX performance for these operations is reported.

4.2.1 Fill (Clear, Tile)

The operation is to write a pixel to memory.

```
/* store registers Rfill..Rfill+31 to
   memory at dst */
stld(dst, Rfill, 32)
```

Tile is the about the same speed as fill when the source fits in the registers.

4.2.2 Stipple

The operation is to write a pixel to memory if the corresponding bit in a mask is set.

```
/* store next 32 bits of mask into
   MASK register */
write(MASK, mask)

/* conditional store registers Rstip-
   ple.. to memory at dst */
stld(M, dst, Rstipple, 32)
```

If the pattern is small enough, say 32 x 32 bits, it does not need to be loaded before each store.

4.2.3 Copy

The operation is to copy a pixel from memory to memory.

```
/* load next 32 pixels into Rcopy from
   memory at src */
ldld(src, Rcopy, 32)

/* store from Rcopy to memory at dst
   */
stld(dst, Rcopy, 32)
```

“Backward” copies, needed when copying from a source to a destination on the same raster, with the destination coming after the source, are done simply by stepping the addresses backwards.

4.2.4 Transpose

The operation is to transpose the raster about the diagonal extending from the upper left to the lower right corner.

```
/* read in a block 12 pixels by 8 pix-
   els */
ldld(src, Rblock, 12)
... seven more loads ...

/* gather each row and store into mem-
   ory at dst */
gath(Rblock, 8, Rrow, 8)
stld(dst, Rrow, 8)
gath(Rblock+12, 8, Rrow, 8)
stld(dst, Rrow, 8)
... ten more gath/stld ...
```

The basic idea is to load the 12 x 8 block of pixels, and then gather each column of 8 pixels into 8 sequential registers and then store. Other operations such as flipping about the horizontal axis or rotating by 90 degrees can all be done by changing the starting register and the step size (positive or negative) of the gather operation.

4.2.5 2D Lines

The operation is to draw a 16 pixel long Bresenham line.

```
/* perform a Bresenham iteration and
   calculate the address offsets */
plot(Rbase, Rdelta, Roffset, 16)

/* indexed store to memory -- store
   pixel to dst+offset */
stla(dst, Roffset, Rpixels, 16)
```

The plot instruction calculates the offsets which are used by the stla instruction (store-long-array) to store the pixels.

Memory can be accessed in direct or array mode. In direct mode, consecutive words in physical memory are accessed. In array mode, an offset vector is used for calculating the effective addresses of the memory words to be accessed.

Bytes can be accessed in four formats: normal bytes, quad bytes, channel bytes and packed bytes. Quad bytes are used for processing each channel of a XBGR raster separately. Channel bytes are used for processing one channel of the XBGR raster. Packed bytes are useful for data movement or efficient logical operations on 8 bit data.

name	description
ld[b|s|c|q|l|p]d	load bytes, shorts, every 4th byte, unpacked XBGR, longs or packed bytes to registers from base address
ld[b|s|q|l]a	load bytes, shorts, unpacked XBGR or longs to registers from base address plus offset from vector
st[b|s|c|q|l|p]d	store registers as bytes, shorts, every 4th byte, packed XBGR, longs or packed bytes to base address
st[b|s|q|l]a	store registers as bytes, shorts, packed XBGR or longs to base address plus offset from vector
Memory operation variations	
precision	load and store operations allow signed and unsigned shifts to support 32.0, 24.8, 16.16 and 8.24 fixed point formats
conditional store	store operations include mask-based conditional store, mask-based store select, and store through plane mask
clamp	clamp to 8 or 16 bit signed or unsigned format on store
write	writes a value into a register via the instruction queue

Table 2 - Memory Operations

The SX can load byte and short elements with sign extension, as well as with predetermined left shifts. The SX can store with mask, plane and clamp controls, which enable-masking of elements within a vector, masking of bits within an element, and, for short and byte elements, clamping between predetermined minimum and maximum values.In addition, a store select option allows the data written to be selected from the elements of one of two destination vectors.

4.1.11 Write

The write instruction places a value into a register. This is done through the instruction queue in order to ensure that the register value is synchronized with subsequent queued instructions.

Memory pixel format	Register format	Sign and scaling
8 bit	one pixel to one register	signed or unsigned left shifted 0, 8, 16, or 24 bits
16 bit	one pixel to one register	signed or unsigned left shifted 0, 8, or 16 bits
32 bit	one pixel to one register	as is
8 bit (packed)	4 8-bit pixels to one register	as is
XBGR (4 8-bit pixels)	one pixel to four registers	signed or unsigned left shifted 0, 8, 16, or 24 bits
XBGR	one channel (X, B, G, or R) of one pixel to one register	signed or unsigned left shifted 0, 8, 16, or 24 bits

Table 3 - SX Data Formats

sources zero and as a destination, discards data. The immediate value range is from -64 to 63.

4.1.1 ALU Unary and Binary

These instructions perform basic unary and binary operations. One source vector is necessary; for binary operations, the second operand may be another vector, an immediate operand specified in the instruction, or a scalar. The result of the operation is placed in the destination vector.

4.1.2 Sum and Dot

Sum adds the elements of a source vector and a scalar and stores the result in a scalar. Dot multiplies corresponding elements of two source vectors and stores the sum of the product in a scalar. For dot, the options are the multiply precision (16x16 and 16x32), the rounding, and the scaling (0, 8 or 16 bits).

Dot can be used to implement digital filters.

4.1.3 Saxpy

In this instruction, the elements of the source vector are each multiplied by the value in the SCAM register and added to the elements of a second vector. The result is placed in the destination vector. The options are the multiply precision (16x16 and 16x32), the rounding, and the scaling (0, 8 or 16 bits). Saxpy is useful in many image processing operations such as convolution and color space transformation.

4.1.4 Shift

These instructions perform arithmetic, logical and funnel shifts. Arithmetic and logical shifting is performed by shifting the bits of each element in the source vector by an amount specified by the immediate operand, or by the elements of a vector. Funnel shifting is performed by shifting across source vector elements by an amount specified by the scalar or immediate operand.

Funnel shift can be used to align one bit rasters.

4.1.5 Compare

These instructions compare the elements of a vector to those of another vector or a scalar. The functions (==, <=, <, >=, >) are available for vector comparisons, and (==, <, >) are available for scalar comparisons. The mask register is set as a result of the comparison: a “1” is written to bits corresponding to those positions where the result of the comparison is true, and a “0” otherwise. These operations can be used for thresholding an image, for example.

4.1.6 Scatter and Gather

Scatter places consecutive elements from its source vector in nonconsecutive elements into the destination; the distance between each nonconsecutive element is specified as an immediate value, positive or negative. Gather is similar except it takes nonconsecutive values and places them consecutively in the destination.

These instructions can be used for zooming and transposing images.

4.1.7 ROP

This instruction performs a raster operation (rasterop) between two source vectors into a destination vector. The actual function performed is dictated by the content of the ROP register. Boolean operations between pixels can be performed by the rop instruction.

4.1.8 Plot and Delt

The plot instruction performs Bresenham interpolation [3], and creates a vector of offset values from the base offset, base error, and major and minor offsets and error deltas. The delt instruction adds four delta values to four successive elements in the source vector. For example, plot followed by delt allows computation of pixel addresses and values for a linearly shaded 2-D line which is then drawn by an array mode store.

4.1.9 Selection

This instruction selects between the elements of two source operands to write into the destination vector. Selection is dictated by the bits of the Mask register. Source operands can be vectors or scalars, and selection on a byte-by-byte basis is also allowed.

4.1.10 Load and Store

These instructions insert and extract long and short words and bytes to and from memory. There are many attributes of the load and store operations of the SX. The important ones are described below.

Unfortunately, consistent MBus activity could keep the SXPP from ever accessing memory. To avoid this Pixel Processor memory activity is specially monitored. When an SXPP read or write request appears on GRIF, a counter is started. If the counter exceeds a registered impatience count, the GRIF operation will be executed next. This allows some SXPP memory operations in even the most heavily loaded MBus streams.

Since the SXPP acts on vectors of data, its operations are generally much longer than the cache-line based MBus operations. The VRAM access modes make this difference more marked. The longest SXPP memory request can take over a hundred cycles to complete, as compared with a maximum of twenty cycles for MBus. While an MBus write would simply be placed into the write buffer, and MBus read might have to wait for one of these SXPP operations to complete before it can be acknowledged. To avoid locking up the MBus, read operations can force an abort of an SXPP operation. When the read has completed, the pixel processor's operation is restarted with the next element.

4.0 Instruction Set

The instruction set is divided into two categories, processor instructions which are executed in the ALU's and multipliers, and memory operations which are handled by the Load/Store logic through the GRIF. These are summarized in Table 1 and Table 2 and described in the following section.

name	description
ALU unary and binary operations	
add[v|s|i]	add vector, scalar or immediate to vector
sub[v|s|i]	subtract vector, scalar or immediate from vector
or[v|s]	bitwise or of vector or scalar to vector
and[v|s|i]	bitwise and of vector, scalar or immediate to vector
xor[v|s|i]	bitwise xor of vector, scalar or immediate to vector
abs	absolute value
mul	multiply vectors
Vector to scalar operations	
sum	sum vector to scalar
dot	dot product of two vectors to scalar
Shift operations	
sra[v|i]	shift vector right arithmetic by vector value or immediate
srl[v|i]	shift vector right logical by vector value or immediate
sll[v|i]	shift vector left logical by vector value or immediate
slf[s|i]	funnel shift left logical by scalar or immediate
Compare, Select and ROP operations	
cmp[v|s]	compare (>,≥,≤,<,=) vector with vector or scalar and set bits in mask register (MASK)
sel[v|s|b]	select between elements of two vectors, two scalars or corresponding bytes of two scalars based upon bits in mask register (MASK)
rop[l|b|m]	rasterop of two vectors based on logical (32bit word-wise), bytewise, or bitwise interpretation of mask register bits
Miscellaneous operations	
saxp	multiply of vector and scalar (in SCAM register) with vector offset to vector
scat	scatter vector (adjacent registers) into registers
gath	gather registers into vector (adjacent registers)
delt	add four delta values to 4-component vector
plot	Bresenham interpolation to vector

Table 1 - Processor Operations.

4.1 Description of Instructions

In addition to its data registers, the SXPP contains some special registers. These are: the SCAM register, used to hold the constant for saxp; the MASK and PLANE registers, used by store; and the ROP register used by rop. Also, register zero

the data is read from memory, it is passed out to Mbus.

The primary concern for Mbus operations is the read latency time for burst reads. The performance of CPU's with small caches suffers markedly as this latency increases.

3.2.1.2 The GRaphics InterFace (GRIF)

The GRIF protocol calls for address/data pairs for writes. This allows writes of any pattern. The reads come across as 64 bit addresses.

The primary concern for GRIF operations is throughput on writes. Screen fills and copies, and most drawing is limited by the write rate to the frame buffer.

3.2.2 Memory Types

The DRAM has 8 ECC check bits for every 64 bits of data. If a write of less than 64 bits is needed, a 64 bit read is performed, those 64 bits are checked and composed with the write, and 64 bits are written.

The VRAM memory has both bit write mask and byte write enables. All VRAM reads pull 128 bits of data at a time. The VRAM is also multiply mapped. Any location in VRAM can be accessed (either a read or a write) in one of several modes. The most straightforward is 32 bit chunky mode, where 32 bits pixels are read in, one after the other. There is also 16 bit planar mode, where every other byte is read, and the pack/unpack logic forms the data into a continuous stream. The third mode is 8 bit planar, where every fourth byte is read

These different modes allow software to manipulate the true color frame buffer as an eight bit or sixteen bit display, or for support of pseudo-color displays. The eight bit mode is useful for running eight bit applications in one color channel, or for manipulating the overlay data in an XBGR pixel. The sixteen bit mode can be used in medical applications where pixels are often ten or more bits deep, or for a 16 bit frame buffer with 8 color bits and 8 overlay bits. In that case, the 16 bit mapping can be used to access only the color, or only the overlay section of the display memory.

3.2.3 Memory Arbitration

With both the GRIF and MBus interfaces, the SX Memory Controller has two sources of requests for memory access. If an operation goes unacknowledged on the MBus, none of the MBus devices will be able to operate. This will stop access to the memory, writes to the SXPP, and all I/O. An unacknowledged GRIF operation will only stop the SXPP. Because of this, MBus operations are given higher priority. Whenever there are both types of requests pending, the MBus request is executed next.

.

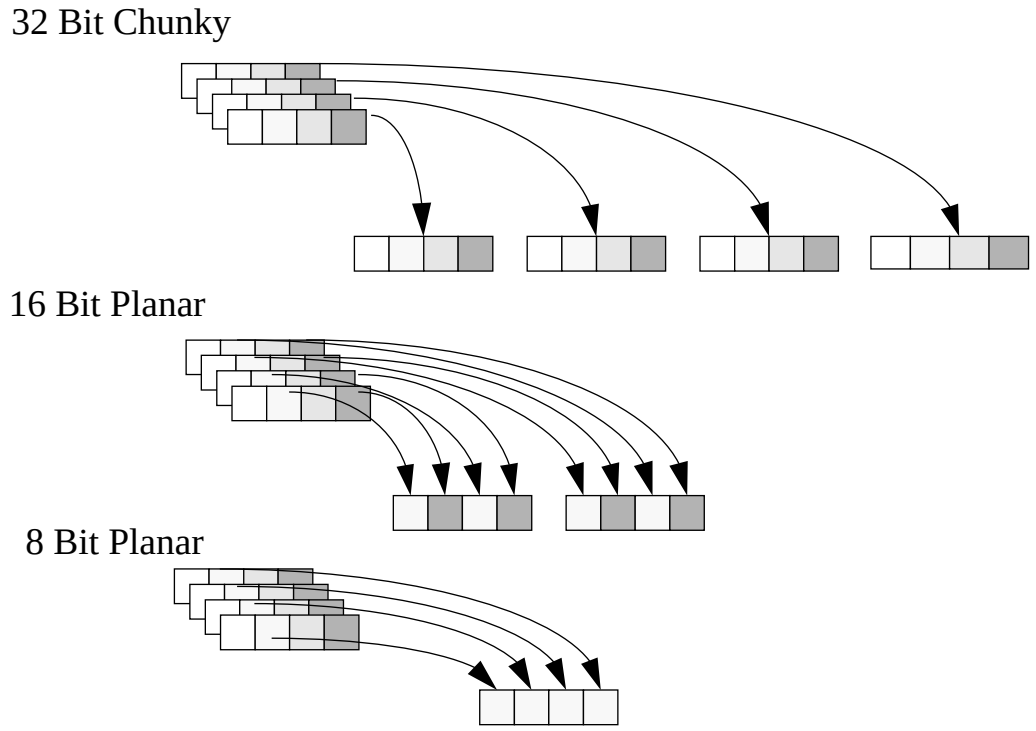

Figure 11 - VRAM Access Modes

(R&R) rather than a normal cycle termination. The R&R tells the Mbus arbitration logic to allow the host processor to resend the instruction after any pending bus requests by other Mbus masters are serviced. This prevents any lengthy SX occupancy of the Mbus which would block an Mbus master's attempted access to either memory, I/O, or some CPU's cache. The R&R is also used by the SXPP to break up the lengthy bus access cycles required for certain rare operations to keep bus occupancy at a minimum. Since the SXPP resides on a multi-processing bus, hardware is included to prevent another processor's accessing the SXPP in the middle of a cycle broken up by an R&R.

The SXPP can be viewed as a load-store machine. SXPP vector instructions can be classified as immediate-register, register-register, or register-memory /memory-register types of operations. Register-register instructions require one 32 bit entry within the IQ. Since Mbus is a 64 bit wide bus, the SXPP allows either one or two of these types of instructions to be sent over the bus and placed into the IQ at one time. The immediate-register instructions use two entries if 32 bits is to be written into an SXPP register or three entries if 64 bits is to be written into two SXPP registers. Register-register and immediate-register instructions are written to a single Mbus address assigned to the SXPP IQ.

Memory access instructions require two IQ entries. A process using the SXPP Memory access instructions require two IQ entries. A process using the SXPP specifies the virtual address of the memory locations to be accessed. The process writes the memory instruction operation code to a region of virtual address space which is entirely mapped to the SXPP instruction queue. This results in a translation of the virtual address to be accessed by the SXPP and transmission of the memory instruction operation code over the Mbus to a 36-bit Mbus address. The lower 32 bits are the physical address to be accessed by the SXPP instruction and the upper 4 bits tell the SXPP to recognize the operation as a write to the SX instruction queue. The result of this is that the SX does not require an MMU and system resources are effectively shared.

Finally, the SXPP contains various control/status registers used to control the flow of instructions, set operation modes, and provide limited error checking and performance monitoring.

3.2 SX Memory Controller

The memory controller needs to not only supply data to the pixel processor, but also provide for the memory requirements of a high speed, multi-processor workstation. There are then two interfaces into the memory controller. One is vector based, and the other cache-line based. There are also two types of memory. One is DRAM with ECC, the other is byte accessible VRAM.

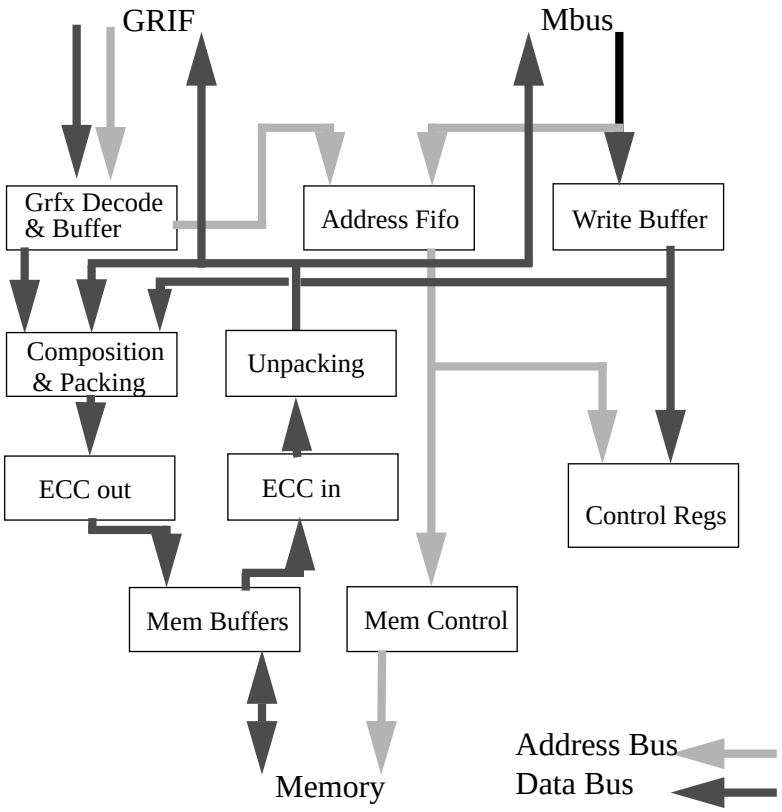

Figure 10 - SX Memory Controller Block Diagram

3.2.1 Bus Interfaces

3.2.1.1 Mbus

When an Mbus write to memory is detected, the address is latched into the Address Fifo, and the data is placed into the Write Buffer. The Mbus then is free to proceed with the next operation, and the data is committed to memory at the next opportunity.

When an Mbus read is detected, the address is again latched into the Address Fifo. All outstanding transactions are completed, while the Mbus waits for the data to come back. As

memory data item comes back from the memory controller. For the SX store instruction, there is never anything written back to the register file as all the data are sourced to the memory controller to be written back to the system memory.

resource and data requirements are extracted and then compared with the current state of the machine in the hazard detect subunit. When all resource and data are available, the pending instruction is launched. Note that for SX processor

A Typical Store Instruction Pipeline in Direct Offset Addressing Mode(16 byte source vector)					
Clock 1	Clock 2	Clock 3	Clock 4	Clock 5	Clock 6
A	B	C	D		
	D	E	F	G	
		D	E	F	G

Events	Concurrent Operations
A	Decode Instruction, Load Base Physical Address, Compute Register File Pointers, Load Vector Count, Load Memory Offset
B	Fetch Data from Register File, Compute Vector Count, Compute End Byte Address Offset
C	Clamp, Format & Steer Data, Prepare next Mask, Check Vector Count, Check Unix Page Crossing, Check Ram Page Crossing, Update Byte Write Mask
D	Compute Register File Pointers, Compute Byte Address Offset
E	Fetch Data from Register File, Compute Effective Physical Address, Compute Vector Count
F	Clamp, Format & Steer Data, Check Unix Page Crossing, Check Ram Page Crossing, Update Byte Write Mask, Prepare Next Mask, Check Vector Count
G	Memory Data Address Valid, Memory Data Valid, Store

Figure 9 - Store Pipeline

An example of the SX store pipeline operation is shown in Figure 9:. Only a single operation is shown, although in practice different operations can be in different pipe stages at the same time. Note that a load goes through a similar process, but does not source data from the register file, and a separate machine writes the data back into the register file.

3.1.4 SXPP Control Unit

The various functional units of the SX are controlled and coordinated by the SXPP Control Unit (SCU). The goal of the SCU is to provide maximum utilization of the hardware resources of the SXPP, while preserving instruction stream integrity. The SCU is responsible for launching instructions, and detecting hazards (structural and data) in the instruction stream. The SCU interfaces with the various blocks of the SX through control lines which transmit commands and monitor the status of each functional block.

The SCU is comprised of three basic subunits: a master state machine, instruction decoder, and hazard detector. Instructions from the FIFO are decoded in the instruction decoder block in two clock cycles. During instruction decode, the

operations all instructions are non-blocking, i.e. once started they do not stall. This is not the case for SX load and store operations in which data is transferred across the asynchronous GRIF interface to memory. A primary function of the hazard detect block is to insure the range of registers required for the pending instruction will be available when the pending operation requires them. Thus the pending operation may be launched prior to completion of the previous instruction even when the source operand registers of the pending instruction are destination registers of the preceding instruction.

3.1.5 SXPP Mbus Interface

A 64 entry instruction queue (IQ) is used to hold instructions issued by a host CPU to the SXPP. An SXPP instruction requires either one, two or three IQ entries depending upon its type. Unless the IQ is full, all instruction writes into the IQ are quickly acknowledged. Since it is possible for the host instruction issue rate to exceed the SXPP instruction execution rate at times, the SXPP indicates the IQ is full by terminating the Mbus cycle with a relinquish and retry

results are written back into the register file.

Time →					
Fetch	A0,A1 B0,B1	A2,A3 B2,B3	A4,A5 B4,B5	A6,A7 B6,B7	A8,A9 B8,B9
Xalu		A1opB1	A3opB3	A5opB5	A7opB7
Yalu		A0opB0	A2opB2	A4opB4	A6opB6
Writeback			S0,S1	S2,S3	S4,S5

Figure 7 - A Typical ALU Instruction Pipeline

3.1.2 The Register File

The register file is the source and destination of the vector elements on which the SX instructions operate. It is, in essence, the central data cache for the graphics and imaging acceleration engine. It is composed of four interleaved banks of 32 entry, dual read ported, 32-bit wide registers. This gives a total of 128 registers.

All the registers can be read and written by instructions using register numbers to specify the registers of interest. The data select logic in front of the register file allows data loaded from memory or a data stream supplied by one of the processing units to be written into the register file. The cross bar at the output of the register file routes the data fetched from the register file to the execution units for processing. This is also where immediate values and scalar values are introduced into the vector stream, by selecting the value instead of one of the streams coming from the registers.

3.1.3 The Load and Store Unit

The load unit executes a set of instructions that handle a variety of data types encompassing the range of pixel types and memory data formats the SX can process. It receives the incoming memory data vector from the graphics data path and then formats and steers each element of the vector into the register file.

Likewise, the store unit executes a set of instructions for a variety of data types. It extracts each vector element from the register file, clamps (if required) formats and steers them into a memory data stream before transporting them to the graphics data path.

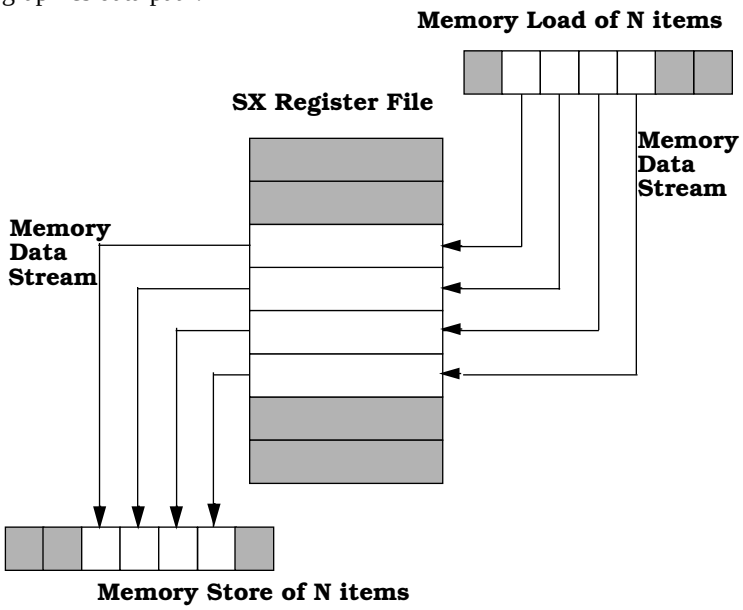

Figure 8 - Load/Store Operations

The functions of the load and store units are illustrated in Figure 8:. The memory data is represented by a stream of data items. For the memory load operation, the stream of memory data items is the source and the register file is the destination where the data items are organized in a vector form. The register file is the source and memory is the destination during memory store operations.

The instruction pipelines for the load and store operations are similar to that of the basic ALU instructions. However, the writeback stage is not necessarily the time that the results are written back to the register file.

For the load instruction, the writeback occurs when the

The two multipliers and their two adders can be used together to do a single 16x32 multiply. Because of the units feeding each other, the throughput is halved, and the latency is increased.

Time →						
Fetch	A0,B0	A1,B1	A2,B2	A3,B3	A4,B4	A5,B5
Xmult		A0.high xB0.low	A1.high xB1.low	A2.high xB2.low	A3.high xB3.low	A4.high xB4.low
Ymult		A0.low xB0.low	A1.low xB1.low	A2.low xB2.low	A3.low xB3.low	A4.low xB4.low
Xadd				P0.high +P0.low	P1.high +P1.low	P2.high +P2.low
Yadd					rnd P0	rnd P1
Writeback						P0

Figure 5 - 16x32 Multiply Pipeline

3.1.1.2 ALU's

The SXPP has two nearly identical ALU's. Each one consists of a shifter, an adder, a comparator and a logical unit. The adder supports add, subtract and absolute value operations. The shifters support logical and arithmetic shifts, as well as funnel shifts which shift entire vectors as one large operand. The comparators compare, and the rop logic does all of the logical operations, such as Selects, AND's, NOT's, and the like. Normally data comes in through the four buses (A and B operands for X and Y ALU's) and XA and XB are operated on together by the X ALU, while YA and YB are used by the Y ALU.

The two ALU's differ in the way they handle certain special instructions. The SUM instruction, which produces a single result, is only done in the XALU. The Bresenham PLOT operation computes the error in one ALU, and the offset in another. Compare instructions built up a mask based on their results which is computed in the X shifter. But the majority of operations are executed by both the X and Y, two results being produced each cycle.

The execution of typical ALU instructions is more straightforward than even the 16x16 multiply. The data comes in, and every cycle two pairs of data are operated upon, and

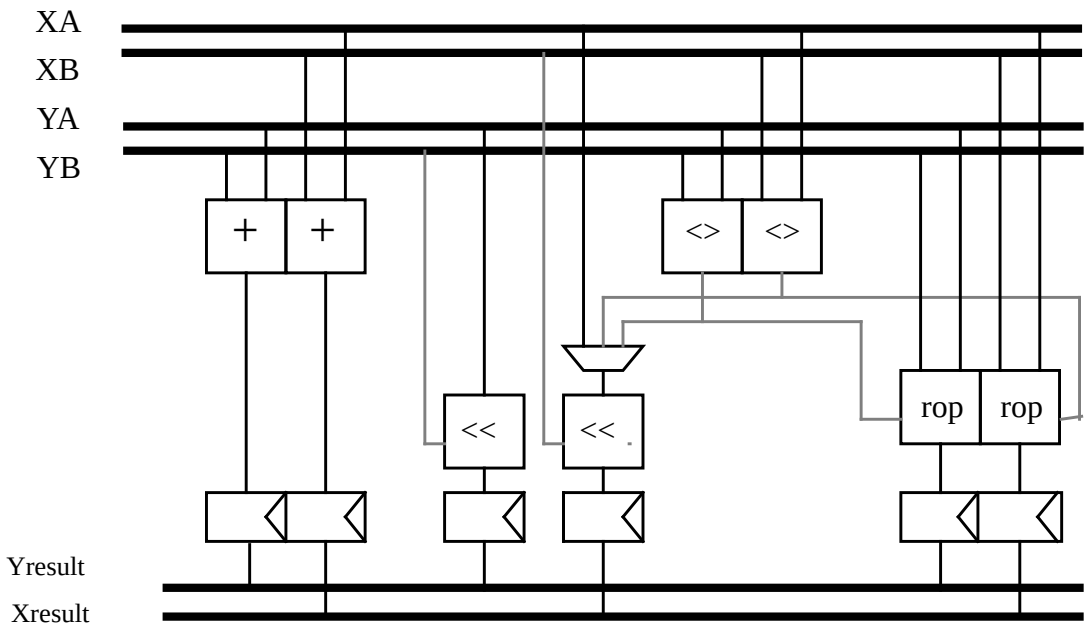

Figure 6 - SXPP ALU's

and add the two vectors together, each performing one add per cycle. The resultant sum vector would be written back into the register file.

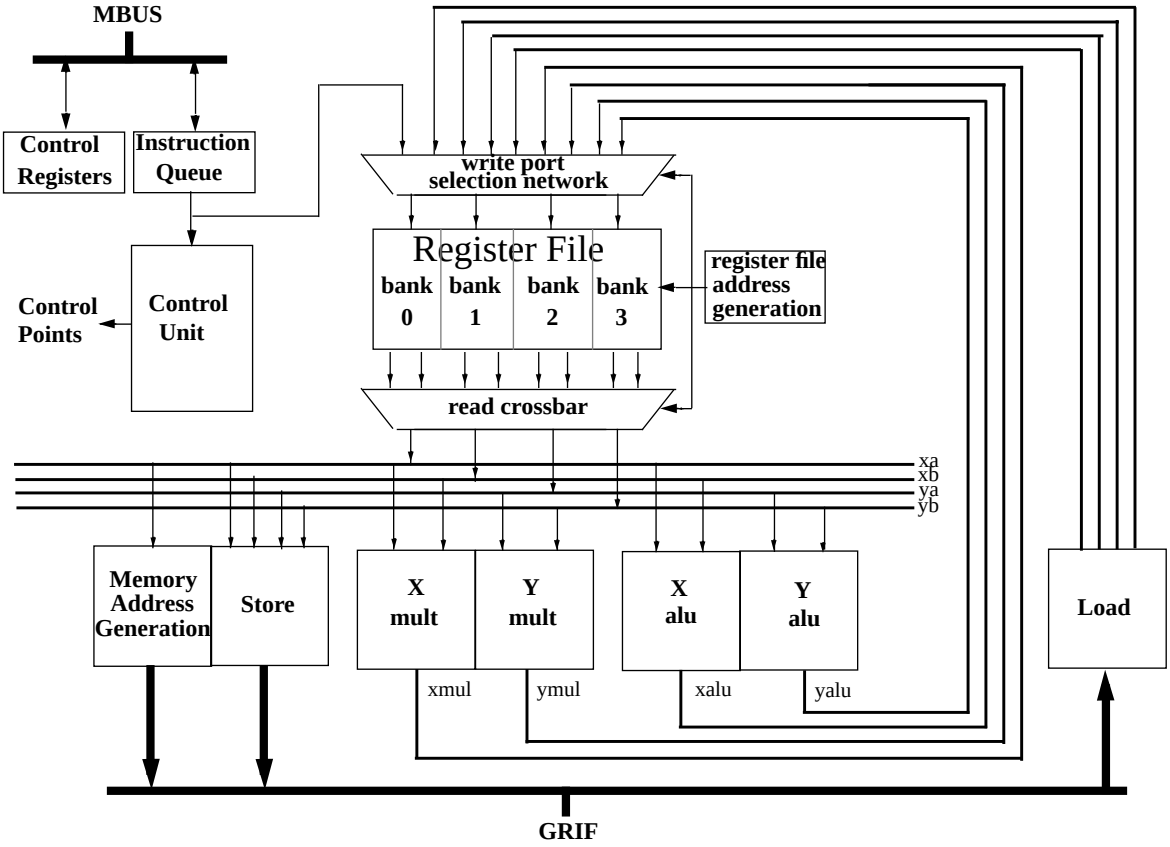

Figure 3 - SX Pixel Processor Block Diagram

3.1.1 Processing Units

There are four processing units, two multipliers and two ALU's. The two multipliers (Xmult and Ymult) act independently for simple instructions, and can be used as a pair for more complex operations. Similarly, the two ALU's (Xalu and Yalu) can act by themselves or in concert, depending on the complexity of the instruction.

3.1.1.1 Multipliers

Each multiplier unit is made up of a 16x16 pipelined multiplier and an associate adder. The combined multiplier/adder allows rounding of results, dot instructions, and saxpys. Note that it takes two cycles to get a result from the pipelined multipliers.

Time →					
Fetch	A0,A1 B0,B1	A2,A3 B2,B3	A4,A5 B4,B5	A6,A7 B6,B7	A8,A9 B8,B9
Xmult		A1xB1	A3xB3	A5xB5	A7xB7
Ymult		A0xB0	A2xB2	A4xB4	A6xB6
Xadd				rnd P1	rnd P3
Yadd				rnd P0	rnd P2
Writeback					P0 & P1

Figure 4 - 16x16 Multiply Pipeline

3.0 SX Architecture

The SX chip is made up of the SXPP and SXMC. Each of them connect to the MBus; the SXPP for instructions, and the SXMC for memory read or write operations. The SXPP makes requests to the SX Memory Controller through the Graphics Interface (GRIF). The high bandwidth of GRIF allows the pixel processor to manipulate vectors of data.

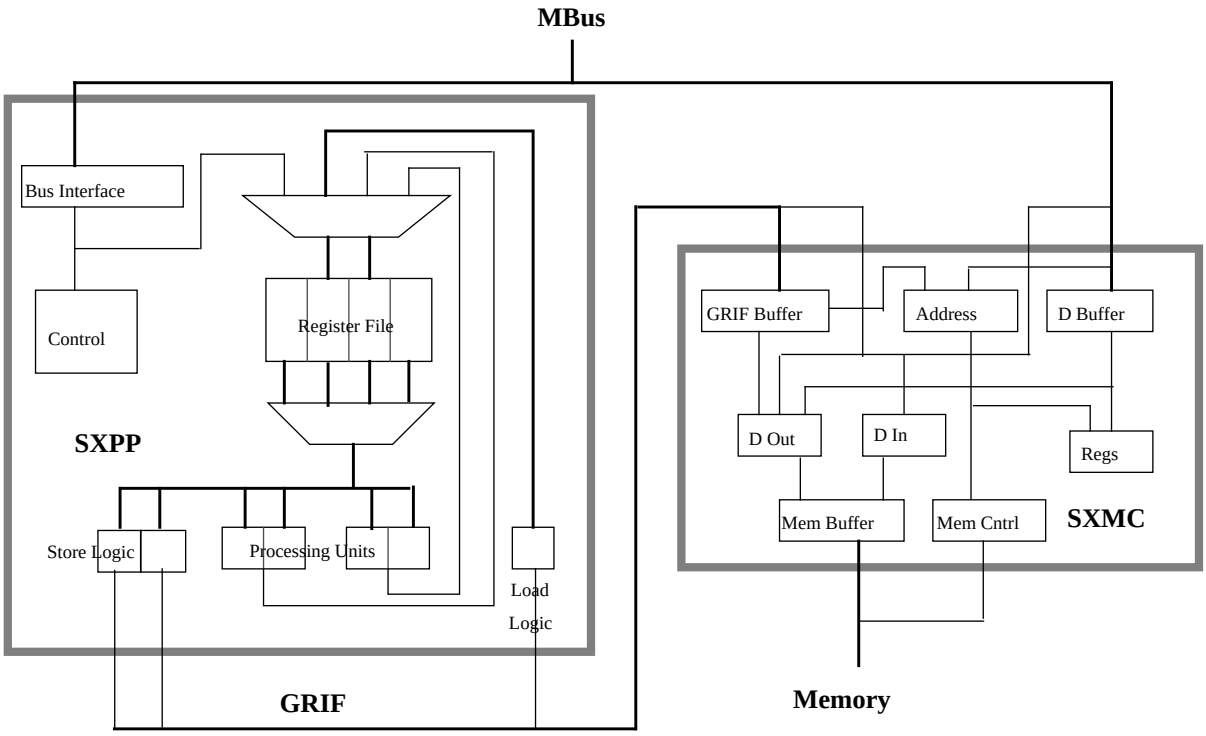

Figure 2 - SX Block Diagram

Missing from the SX are a floating point unit, a dedicated edge walker, a Z-buffer interface, a DSP, or any large function-specific logical block. Floating point is handled by the CPU, edge walking can be done in the SXPP or CPU, an effective and inexpensive Z-buffer can be implemented through main memory, and most graphics, imaging and visualization algorithms can be executed in the SXPP. While it would be faster to have each of these units specifically implemented in hardware, it would be prohibitively expensive. The SX can accelerate a wide range of applications at only the cost of an increase in memory controller die size.

3.1 SX Pixel Processor

The SX Pixel Processor is an integer vector processor. It acts as a coprocessor to a CPU, executing the instructions passed to it on MBus. Combined with a fast memory interface, the SXPP is designed to handle the inner loop code of many algorithms. It can read data in, act on it, and store it to the display memory or main memory. The instructions set covers mathematics, logic and graphics, but has no loop or branch instructions.

There are five major portions of the SX Pixel Processor. They are the Mbus interface which handles bus interface and instruction storage, the SX Control Unit which manages setup and sequencing, the Register File which contains the 128 registers used as source and destination for the operations, the Processing Units which execute all of the arithmetic and logic instructions, and finally the Load/Store block which handles all of the memory read and write requests across the GRIF.

If an arithmetic operation, a vector add for example, were sent to the SX, it would be taken from the MBus, and placed into the instruction queue. When its turn came, it would be passed to the control unit and would be launched once all of the necessary resources were available. The two source vectors would then be read from the register file, and sent out on the data buses. The Xalu and Yalu would take the data in,

2.0 Target System

The target system includes one to four SuperSPARC processors on a 64-bit interprocessor bus (Mbus), and the SX chip which handles accesses to main memory (DRAM) and video memory (VRAM) on a 128-bit memory bus. The memory management units (MMU) are the SPARC Reference MMU contained within each SPARC CPU chip. They perform virtual to physical address translation and provide process protection.

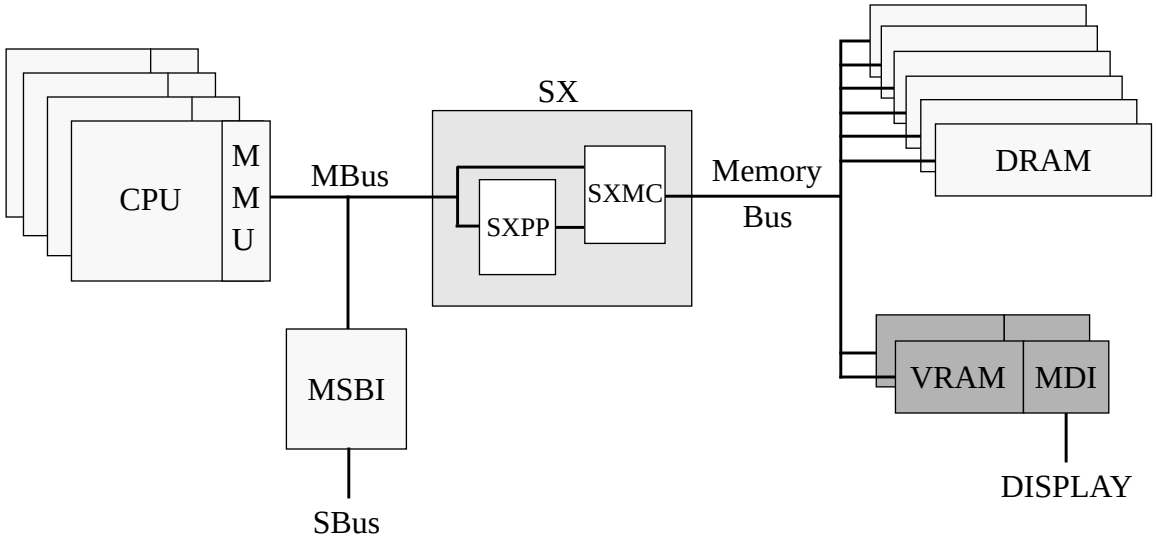

Figure 1 - The Workstation Architecture

The system processors and I/O devices interface to the SX through the Mbus. Mbus is a synchronous 64 bit, multiplexed address-data, circuit switched bus supporting multiple masters. Shared memory multiprocessor support is accomplished with a write-invalidate cache consistency bus protocol. The SXPP and SXMC portions of the SX both have Mbus control logic allowing them to connect to the single Mbus interface of the chip operating semi-autonomously.

The memory interface is 128 bits wide, and uses the RAS/CAS protocol common to most dynamic memory. There are 16 extra bits which are used for ECC on DRAM, or byte write enables on VRAM. While the two types of memory are handled differently by the SXMC, the differences are not exposed to the CPU's or SXPP which can access either type of memory.

The display subsystem of the workstation is based on the Memory Display Interface (MDI) chip. It is a high performance video display controller featuring up to three 24-bit color lookup tables, hardware cursor support, transparent overlay support with blending, and fully programmable monitor timing. A video SIMM containing the MDI, 4MB VRAM, a three channel 10-bit DAC, a pixel clock generator and other miscellaneous support logic provides video signals required for a 76 Hz 1152X900 true color display with 32-bit pixels. The architecture supports up to 4 video SIMMs.

The I/O subsystem of the workstation is based on the Sbus which is connected to the memory bus by the Mbus to Sbus Interface chip (MSBI). The workstation provides four Sbus slots and also comes complete with the same rich set of integral I/O devices: Ethernet, ISDN ports, serial ports, parallel port, digital audio, SCSI, floppy disk, and keyboard/mouse.

A Memory Controller with an Integrated Graphics Processor

John Watkins, Raymond Roth, Michael Hsieh, William Radke,
Donald Hejna, Byung Kim, Richard Tom

Sun Microsystems, Inc
2550 Garcia Ave, Mountain View, CA 94043

Abstract

This paper describes the SX graphics accelerator. It is a programmable processor built into a workstation memory system. The goal of the SX is to achieve performance comparable to that of low end 2D and 3D graphics processors and to surpass low end imaging performance, at the lowest possible cost.

The SX contains an integer vector processor, with an instruction set tailored to the needs of image processing and multimedia as well as 2D and 3D graphics. It can directly access data in both video and main memory, allowing accelerated processing on images up to 512 Mbyte in size. The SX offers a cost-effective method of providing a rich graphics and image processing capability compared to traditional workstation and accelerator approaches.

1.0 Introduction

There has been a trend of demand for the contemporary workstations to provide imaging capabilities in window systems [7 [5] as well as to support the video display service in the underlying transport media such as the e-mail [2]. One solution has been to enhance the graphics and imaging performance of the workstation by adding the special purpose acceleration engines. The added hardware performs some or all of the stages of specific classes of algorithms such as 2D rendering [6], 3D rendering [1][5], image processing [4], or video compression [3]. This solution also adds to the cost, while only helping those features supported by the specialized option.

The first design goal for this project was to provide a workstation which achieves performance comparable to that of machines with low-end 2D and 3D graphics processors, and surpassing low-end imaging machines. The second goal was that this be achieved with only a small incremental cost over an unaccelerated workstation. By implementing an integer vector processor within a traditional error correction code (ECC) memory controller, both of these goals were met.

The SX chip architecture offers a cost effective method of providing a rich graphics and image processing capability in a workstation comparing favorably with those approaches using the add-on acceleration options for specific applications. The SX Pixel Processor (SXPP) has an instruction set which is tailored to the needs of image processing and multimedia applications as well as 2D and 3D graphics. The SX Memory Controller (SXMC) allows the SX Pixel Processor to directly access data in both video and main memory, while still filling the role of system memory interface. This organization enables the SX Pixel Processor to provide accelerated processing of any image which will fit into the 512 Mbyte main memory.

In addition, the inclusion of the SX chip in a workstation offloads the burden of performing the graphics operations with the CPU. One most visible benefit is that the operations accelerated by the SX chip do not allocate cache lines and hence do not interfere with other CPU operations. Also the memory bus is not congested with the transfer of large images to and from the screen.

The following sections describe the workstation which hosts the SX chip, the functional blocks and pipelines of the SX Pixel Processor and SX Memory Controller, the vector instruction set for the SXPP, and the performance of the overall system.