THE AMIGA'S CUSTOM GRAPHICS CHIPS

CONDUCTED BY PHILLIP ROBINSON

A conversation with Jay Miner, the chips' designer

COMMODORE'S NEW AMIGA *micro-computer contains a custom* NMOS *(negative-channel metal-oxide semiconductor) chip set that provides many powerful graphics functions. The* Amiga *preview in the* August *issue of* BYTE *("The Amiga Personal Computer" by Gregg* Williams, *Jon Edwards, and Phillip* Robinson, *page* 83) *briefly described those chips. Later, we went back for more details and talked to* Bill *Kolb, Amiga's director of hardware engineering, and Jay* Miner, *the vice president of product development and the designer of the chips. Miner also designed the graphics hardware for the* Atari VCS (2600), 400, *and* 800 *personal computers.*

Although the Amiga *team set out to build a general-purpose microcomputer, Kolb states firmly, "We not only wanted graphics, we wanted enough power to do real animated graphics—where you're not just moving one sprite around on the screen. We wanted to take the next major step, and* VLSI *[very-large-scale integration] was the only way to be that aggressive and keep the cost within reason."*

The Amiga *was originally Miner's idea for the world's most powerful game machine. But as other people joined the team, that conception changed, and features, capabilities, and more* ROM *(read-only memory) were added to the system.*

Block diagrams of the three chips and of the Amiga's *overall architecture accompany this interview (see figures* 1, 2, 3, *and* 4).

ORIGINS

BYTE: *What are the names of the three chips?*

Kolb: Agnus, Denise, and Paula. All DMA (direct memory access) channels reside in Agnus. Agnus is sort of a shortening of address generator. Denise handles most of the video output. Paula's two main functions are sound and the various I/O [input/output] functions. Logically, it's one big chip. For instance, both Denise and Paula are dependent on Agnus for their addresses, but they just weren't feasible as one chip. So they were split up functionally, but it looks like a giant control block to an assembly-language programmer.

BYTE: *When did this design start?*

Miner: It really started with the beginning of the company. In the early days there was more emphasis on the video game than there was on the personal computer. The cost targets were for a much lower-priced machine. We were thinking in terms of $300 or less at the beginning.

At that time we planned to use the 68000 chip, and we didn't expect to have much memory or a built-in disk. The low-cost game machine might not even have a keyboard, but it would have high resolution, a 68000 chip, and superior graphics. Then as time went on, it grew and grew. The individual chips grew, too. The software people talked us into putting in things like hardware line-draw and hardware area-fill.

BYTE: *So even at the beginning you were picturing custom chips for the graphics?*

Miner: Oh, yes. I did the chip set that was in the Atari 400 and 800 and in the original Atari VCS machine. I had a good appreciation for the power of custom graphics chips. We didn't have nearly the extent of the circuitry that's on the chips now. We had visions of a fairly crude form of blitter, nothing

(continued)

Phillip Robinson is a senior technical editor for BYTE. *He can be contacted at* McGraw-Hill, 1000 *Elwell Court, Palo Alto,* CA 94303.

as sophisticated as the three-input, generalized blitter we have now.

THE BIMMER

BYTE: *Tell us about the bliiter.*

Miner: I like to call it a *bimmer* because that stands for bit-mapped image manipulator. The term *blitter* is left over from literature referring to block transfer. This machine does block transfer, but it does much more because it has three inputs, and those three inputs can be combined in many different ways.

The logic operations that can be performed are complete. If you think of three variables, you can perform 256 logic operations on them. The bimmer is intended to be a non-real-time machine that transforms images from one location to another or back. Its main distinguishing features are the logic functions on all four inputs and the capability of barrel shifting, so you can move an image on any pixel boundaries. Only two of the sources have barrel-shift capability. Also, the bimmer can do "modulos," which means that if you're looking at a large image in bit-mapped memory, the bimmer can operate on a small portion of that image. When it gets to the end of that small portion, you add a modulo to the address to jump it to the next line. That's true of the entire display circuitry; all of the bit planes have the same feature, so you can display a small image out of a larger image.

BYTE: *We're pretty familiar with the Atari* 800 *and the chips there. To do horizontal and vertical scrolling you had to reset the pointer to a new byte.*

Miner: You could only move one byte before you had to reinitialize things in memory. Here you can do the same thing, but you can also be displaying a portion of an actually larger memory. You've got both ways you can go here. The engine that puts the bit planes up on the screen also has modulo capability, so the address at the end of one line doesn't necessarily have to be one less than the address of the beginning of the next line. It can be many less. Simply by changing the beginning pointer of the entire screen, you can move the image through memory. You give it a starting address, which is the address of the point at the top left of the screen. You give it a length and a modulo value.

BYTE: *That would be the whole width?*

Miner: Right. Well, it would be the difference between what you're showing

(continued)

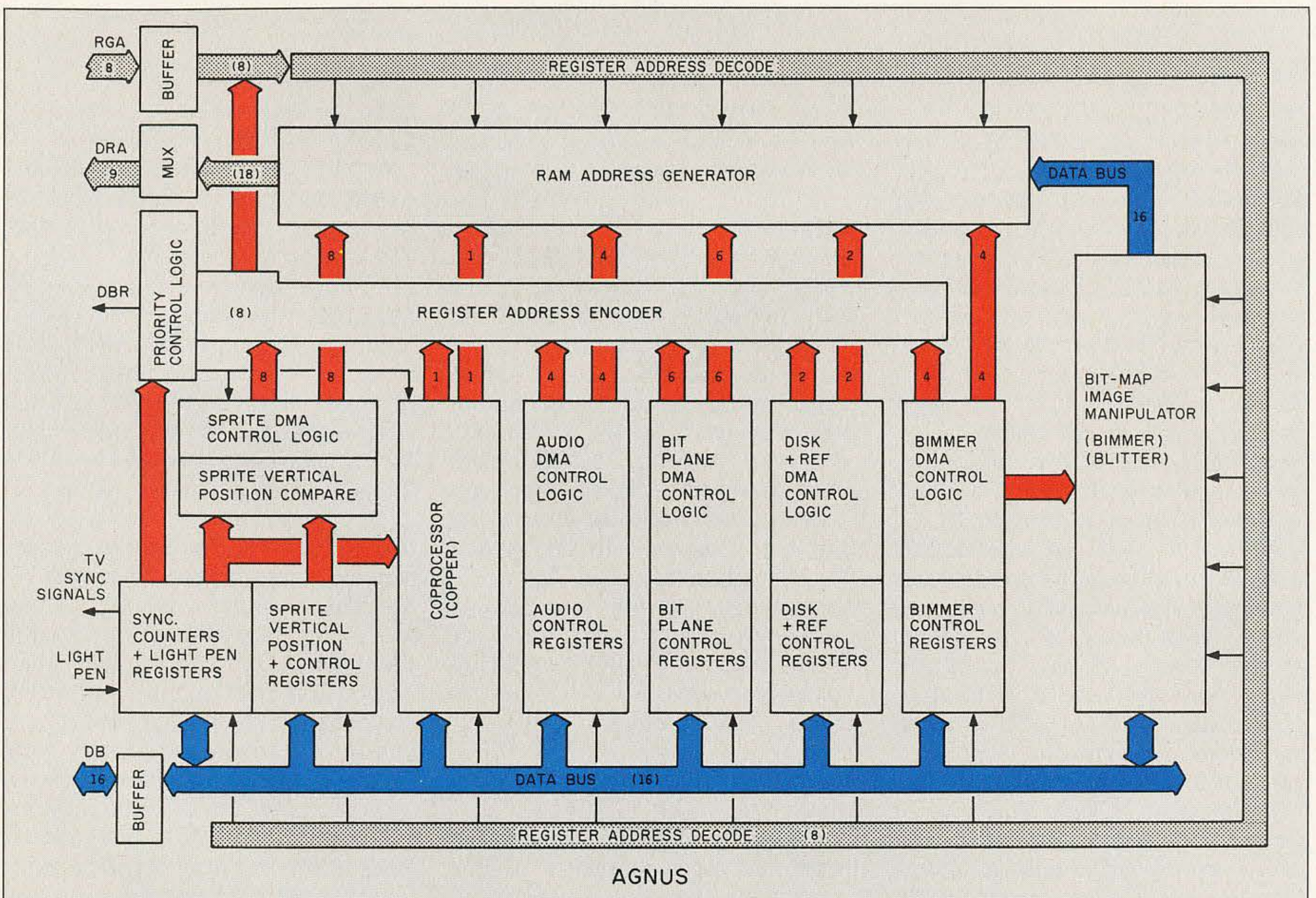

Figure 1: A *block diagram of the Agnus chip.*

and the whole width. There's the capability of six bit planes in this thing. The bit planes can be grouped into two playfields, and each playfield has its own modulo and its own horizontal-scroll register, the same type of horizontal scrolling as in the Atari 800. We thought several times about giving each bit plane its own modulo, but I couldn't think of any display that would really make good use of that, and the extra hardware didn't seem worth it. All the pointers, modulos, backups, and the 18-bit adder that makes them work—by doing both the incrementing and the modulo jumps—are on Agnus, as is all the control logic that sets the priority for which one of those DMA channels gets on the bus at which time.

Agnus Priority-Control Logic—DMA

Miner: The line coming from Agnus's priority-control-logic block should really be labeled DBR, which stands for data-bus request. But it's really not a request, it's a demand, because Agnus always has control.

BYTE: *How do you determine who has priority?*

Miner: The whole priority structure is really interesting. There are a lot of things that have individual time slots that occur, for example, during horizontal blanking. All of the sprite data transfer takes place during horizontal blanking, and it's assigned definite time slots. Each sprite has its own time slot, so it can't interfere with the transfer of the other sprites.

BYTE: *So after each horizontal sync, there are set chunks of time?*

Miner: There are set chunks of time for these data transfers, which include the sprites; the audio, which has some time slots there [four audio channels]; the disk, the refresh—all of these things are assigned. And the display itself, of course, is out here in the display time, so in a sense it also has fixed time slots; it can never compete with these other things. This is all highest, top-level priority. They don't compete with each other because they're always independent. You could have them all at the same priority level without worrying about it. And the stuff at the top priority is the display stuff and data transfer that goes in fixed time slots during horizontal blanking.

The three other things that we have to worry about are the coprocessor, the bimmer, and the main CPU [central processing unit]. That's the way it goes, in that order. The coprocessor is the next most important. It's a real-time coprocessor that's used frequently as a real-time engine to synchronize with the beam for various things like display and audio. It gets any cycles that are empty that it can use, but in order to not make it a hog, it's an every-other-cycle machine at the most. It's looking for empty cycles. If it finds one, it will use it, but it won't use two in a row.

BYTE: *In the top priority, there are no empty spaces?*

Miner: Oh, there are some empty spaces, especially in low resolution. At low resolution, the display portion has empty spaces, another key feature

(continued)

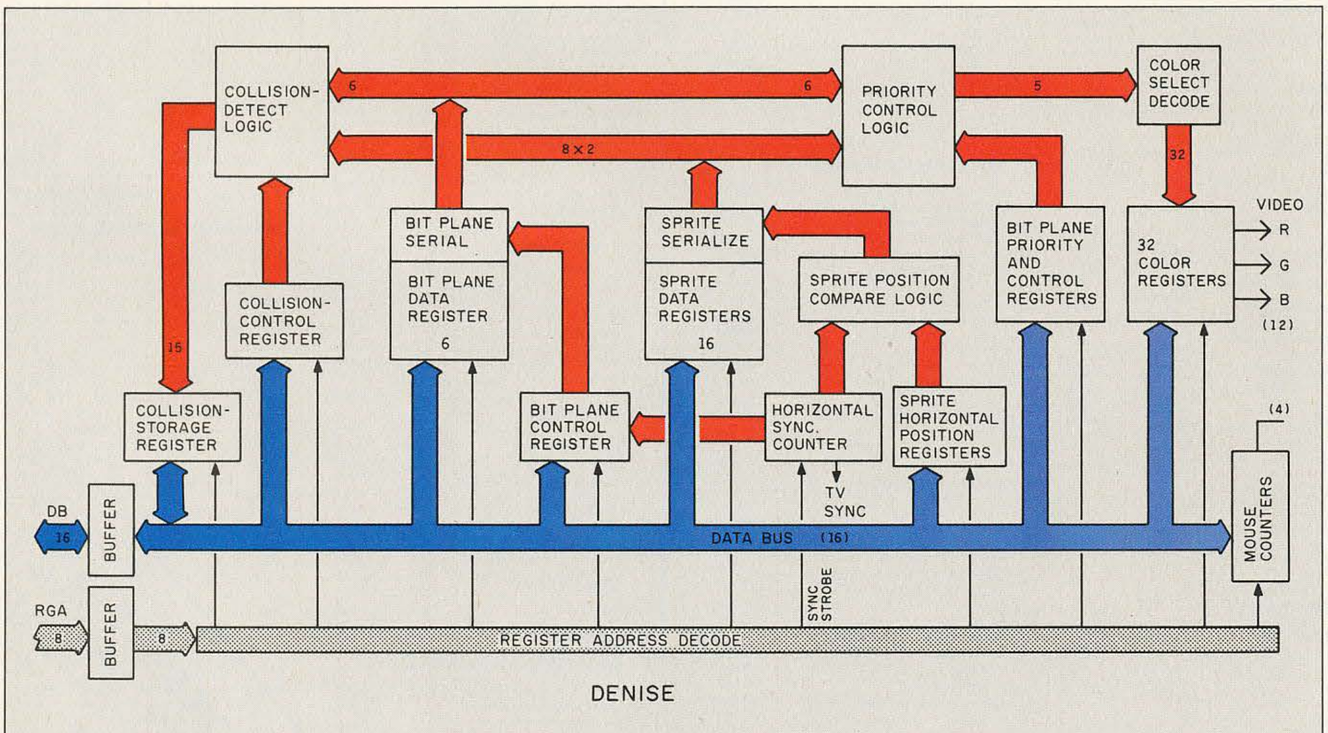

Figure 2: *A block diagram of the Denise chip.*

of the machine. Our normal resolution, what we consider normal low resolution—320 dots across the screen—leaves 50 percent empty slots during the display.

BYTE: *It puts out a dot and then it has a little time?*

Miner: No, putting out dots is continuous. I'm talking about memory fetches to support those dots, which have every other cycle empty.

BYTE: *And there's enough time to switch over and let somebody else use that memory cycle and then switch back?*

Miner: Oh, yes. An empty cycle is up for grabs, always. And during horizontal blanking, to maintain that concept, we have every other time slot assigned to a sprite or an audio refresh. That means that during horizontal blanking, 50 percent of the memory cycles are empty. So looking at the whole time, approximately 50 percent of the cycles are empty. The reason we did this is because the 68000 CPU can use the bus efficiently only 50 percent of the time.

BYTE: *Why is that?*

Miner: Because of the way it's made internally. It has to fetch an instruction, which is a memory cycle, decode that instruction, and do some operation like storing data. The way it's set up, the length of time—the number of clock ticks it takes to decode the instruction—is almost equal to the length of time it spends on the bus. So at the lowest resolution—320 dots across the screen—we match the processor, and the processor thinks it's got an empty bus because it interleaves right in between those display cycles. This is something that is unique to this machine. And the processor is as happy as a clam; it thinks it's got full bus access.

If we go from 320 dots up to 640 dots, then that fills in the display time. But what I just said is still true during horizontal blanking. The microprocessor has the bus all the time that Agnus lets it. The coprocessor is an every-other-cycle machine. It goes along using time slots as it can, based on those rules.

The bimmer, however, is a real hog. If a time slot is available, the bimmer will use it, especially when it's in what's called the nasty mode. Now *there* is a mode where you can tell the bimmer, "Hey, don't be so nasty, don't take so many cycles, leave some for the main microprocessor in case there's an interrupt." Because if the bimmer gets to operating heavily, it's operating on a large area of screen in a non-real-time way; churning memory up, it can hog a lot of bus cycles. Of course, it's the right arm of the main microprocessor, so it's doing things that the main micro would have to do otherwise, in terms of graphics manipulation. Still, if you want to be at all responsive to interrupts—and in a multitasking machine like this, you have to be responsive to interrupts—you've got to have a mode where the

(continued)

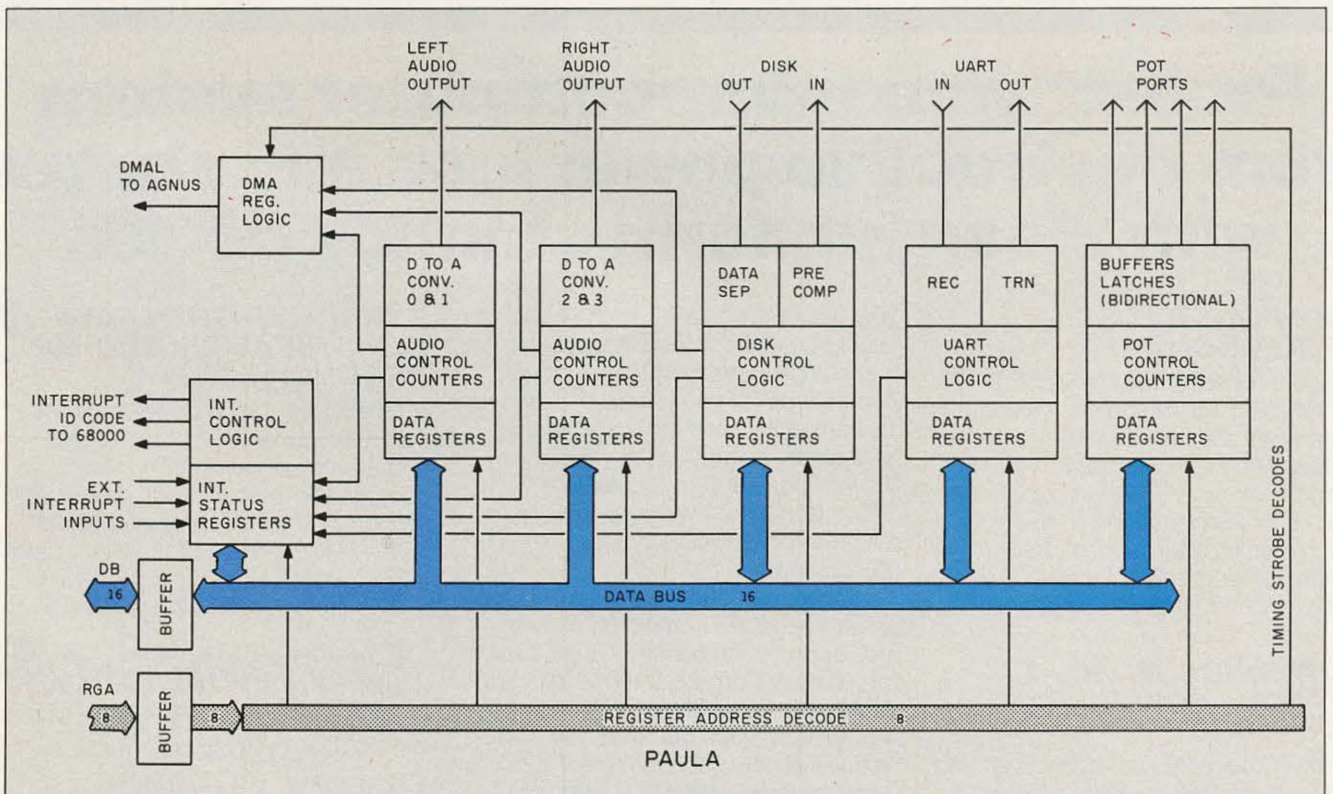

Figure 3: *A block diagram of the Paula chip.*

bimmer once in a while takes a pause and says, "Okay, microprocessor, here, I'll give you a couple of time slots."

SPRITES

BYTE: *We're curious about the sprite hardware.*

Miner: Unlike the Atari 400 and 800, we do not have "players"—sprites with a vertical bit map. The idea was to save vertical registers and vertical comparisons in hardware and put them into the software by requiring the programmer to reposition the image in the vertical bit map.

BYTE: *That was a direct outgrowth of the way you did the players in the Atari 2600, where they were only one line long and one bit wide. You had to redefine them on the fly for each video line. A logical extension would be to make that two-dimensional.*

Miner: It wasn't that so much. You see, the 2600 had no bit map at all except in ROM, where there were some bit maps of sprite images. All of the stuff was created on the fly. I was really tight on register space—the design rules were *big*—we were trying to save as much hardware as we could and put functions into software. So we came up with the idea that if we didn't have any vertical comparators at all, and no vertical position registers, then what we would end up with is a sprite that is not a real sprite in both directions, but only in the horizontal direction. In the vertical direction it's a bit map. That was the concept that we patented in the Atari machine. We don't have that here at all. We've got a general-purpose sprite both horizontally and vertically—a classical sprite concept with a vertical start position and a vertical stop position.

BYTE: *The sprite is 16 bits wide, but it can have any height?*

Miner: It can be any height because it can have any start and stop position, but its height is not related to the bit-map image like it is in the Atari 400 and 800. Its height is related to the line count given by a start-control register and a stop-control register. There are eight sprite engines. Each one is 2 bits deep (which allows for four colors per sprite) and 16 bits wide at low resolution (at the 320-dot horizontal rate).

BYTE: *What choices do you have for the four colors? Is there a separate color table for each sprite?*

Miner: There's a separate color table, but not for each sprite. There's some sharing that has to go on. We've got only 32 color registers, and those have to be shared between the playfields and the sprites. Sometimes the playfield uses 16 of the colors and the sprites have the other 16. Sometimes some of them are shared, depending on how many colors you're trying to show on the playfield. Those eight sprites can be combined to make four sprites that are 4 bits deep with 16 colors each. The total sprite bit resolution, however, does not go down to as fine as the high-resolution playfield.

BYTE: *How do you get more than eight sprites?*

Miner: You can reuse the sprite engines any time you want.

BYTE: *You mean you just have to reset them between frames?*

Miner: You reset them horizontally or vertically. Once you finish using one

(continued)

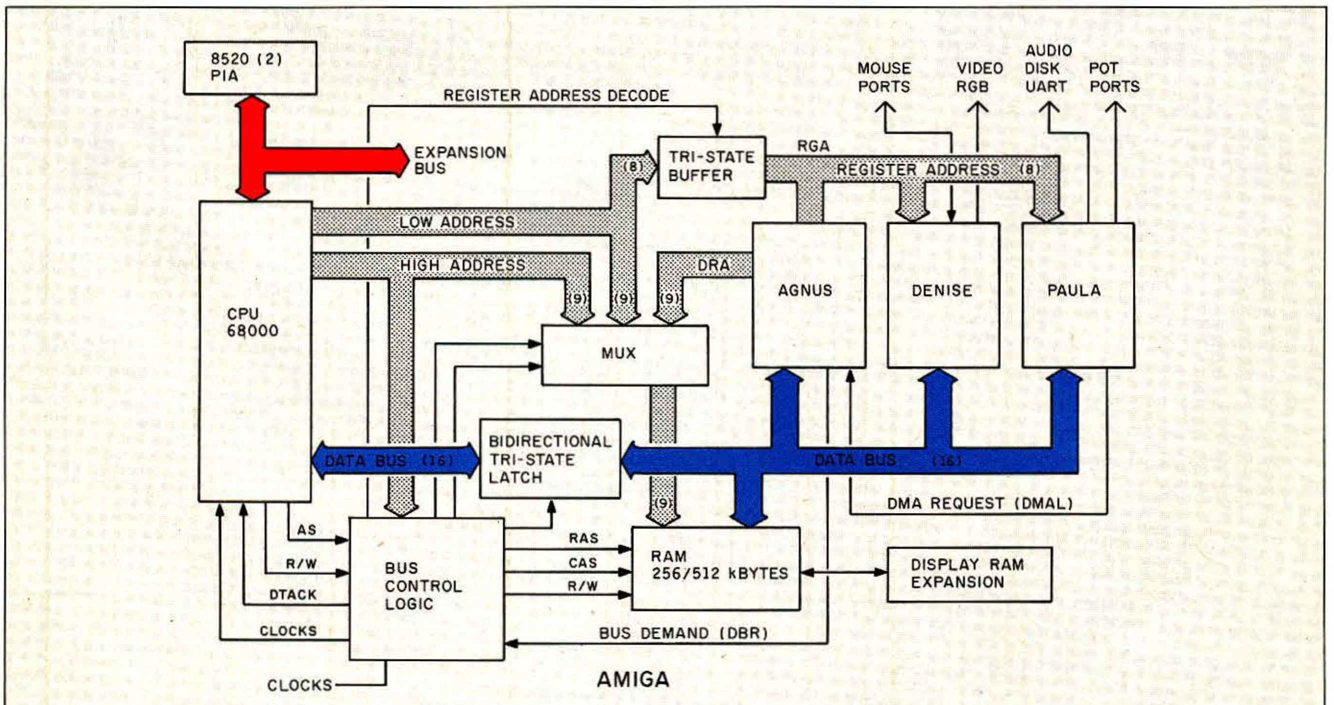

Figure 4: *A block diagram of the Amiga's overall architecture.*

Inquiry 344 →

vertically, you have to wait one line time before you can use the same engine again vertically. You can use the same engine over and over again on a horizontal line if you can get enough microprocessor or coprocessor time to go in there and rewrite the registers.

Digital RGB, Analog RGB, and NTSC

BYTE: *When you put out* RGB *[red-green-blue] data, how does it come out?*
Miner: It comes out as 4 bits, 4 bits, and 4 bits.

BYTE: *And that's how* RGB *monitors normally take their information?*
Miner: Off chip it goes into a ladder. There are three groups of 4 bits coming right out of the 32 color registers, and then there's a four-resistor ladder on each one of those that converts it into three analog values. That's what goes to the monitors.

BYTE: *Then the values of that analog data—which you've changed from the digital data—determine how strong each of the* RGB *guns is when it's firing at a particular point?*
Miner: Yes, on the so-called analog RGB. There are two kinds of RGB: digital and analog. This is important to stress because IBM talks about 16 colors, but what IBM really means is two shades of eight colors, and those two shades are always the same color. There's no way to change them. That's what's called digital RGB, or RGBi. It's got red on and off, it's got green on and off, it's got blue on and off, and it's got an intensity level that determines brightness or darkness for each one of those. It's a four-wire control, but it's completely digital. We put that out too, in order to be compatible, but we also put out the analog RGB, which has 4 bits, 4 bits, and 4 bits, into ladders, so you get 16 values of red, 16 values of green, and 16 values of blue. It's equivalent to 2^{12} total colors and luminances.

BYTE: *So on the analog output, you could have any number of bits that you wanted? You could put out 10 bits on each line?*
Miner: Yes, if you had big enough registers. In fact, that's probably one of the things we'll be expanding in the future chip set.

BYTE: *Why did you choose 4 bits in the first place?*
Miner: Originally, this wasn't going to be RGB; it was going to follow the NTSC [National Television System Committee] standard. NTSC works on intensity, hue, and saturation. Color and luminance. YIQ is what they call it. The *Y* is the intensity, and the *I* and the *Q* define a vector that determines the saturation. Having 4 bits of each was about the best we could tackle in terms of having on-chip ladders that would take the 4 bits for each one of these and convert them into an actual phase angle.

BYTE: *So that ladder is on Denise?*
Miner: No, it was when we had the YIQ; when we were emphasizing NTSC, we had a ladder on board. Then we deemphasized it. We found a Motorola chip that did a good job of converting RGB into NTSC. We needed the extra room on the Denise chip for extra resolution on the color registers, so we dropped the YIQ NTSC completely. But we've still stuck with the 4, 4, 4 bits. Also, you've got a real pin limitation on a chip like this. We tried to keep the chip simple and low-cost to manufacture, and on-chip ladders take up a lot of area. They're notoriously inaccurate, and you can buy 1 percent resistors external for a penny apiece.

BYTE: *Is there still* NTSC *output from the Amiga?*
Miner: In the box there is, yes. But not in the chips.

Bit Planes

BYTE: *Explain the bit planes to me a little better. You've got ones and zeros in memory and you overlay them; you look at a group of them simultaneously to determine what the color is of something that's actually on the screen. Do you have an address for the beginning of each bit-plane area?*
Miner: Yes. The concept of bit planes is very deeply ingrained in this architecture. There are really two conflicting display concepts here. One is pixel addressing; the other is bit-plane addressing. We've chosen bit-plane. One reason for our decision is that we wanted to do a very efficient area-fill.

BYTE: *You mean filling in a particular zone on the screen?*
Miner: Yes, and that's done quite well and efficiently with bit-plane addressing on a single bit-plane basis. We wanted to have a lot of variety in the number of bit planes that you can specify. We wanted to have our two separate playfields—each one with a controllable number of bit planes in it. We didn't want to waste a lot of data transfers if we had fewer bit planes than others did. So we decided not to transfer data on a pixel basis, which wastes a lot of transfer time if you don't use all of your pixels or all the bits within a pixel. Even if you don't use them all, you still have to address them, and it still takes a memory cycle. When you're bit-plane-oriented, if you've got only two bit planes instead of eight, since you're moving data out of a single bit plane only, it doesn't matter because you're using all the bits that come across. That was really why it came about: to increase the efficiency of data transfers and the sprite transfers for different numbers of bit planes and different organizations.

BYTE: *When the bimmer is operating on its three sources and sending to its destination, is it operating on pieces of bit planes?*
Miner: It's always operating on only one bit plane at a time. If you want to do a picture with multiple bit planes, you just do the same routine and point it to where that other bit plane is located.

BYTE: *But it can't take a chunk and move it from bit plane number 1 over to bit plane number 2?*
Miner: It could, sure. But that isn't normally the way it's done. Usually you define the bit planes, and you operate on them as though they were images one behind the other.

(continued)

BYTE: *This screen that you have that's looking at a section of the large image is actually looking at the bit planes stacked on top of each other?*
Miner: No, the bit planes are never really stacked.

BYTE: *Well, in memory then, because memory is just stretched out.*
Miner: Memory is contiguous, right. So the bit planes are really located separately in memory, but since they're fetched by the bit-plane DMA channel, a word at a time from each bit plane, they're placed into these holding registers in Denise. Then when bit plane number 1 comes along, they know they've all been filled, so you simultaneously convert them all from parallel to serial and start squirting them out. While they're squirting out, the parallel's being reloaded to get ready for the next squirt-out. As they come out, you're looking at them as though they were a pixel, at a single instant in time.

BYTE: *How does barrel shifting fit in?*
Miner: The bimmer's barrel-shift capability lets you move images on pixel boundaries. If it weren't for the barrel shifter, the bit-plane concept wouldn't work at all. When you're doing pixel addressing, since each pixel has its own address, to move stuff by one pixel all you have to do is increment the address by 1. There's no problem in moving stuff—using pixel addressing—on arbitrary pixel boundaries. But when you're using bit-plane techniques like we are, where each word represents a whole bunch of pixels from one bit plane, then to move that image within a word, within a single pixel boundary, you've got to shift it by an arbitrary number from 0 to 15.

BYTE: *Even across words?*
Miner: Yes. The barrel shifter allows you to do that here. As the data is transferred from source to destination, you can move it by an arbitrary number of pixels.

SCROLLING

BYTE: *Could you explain the scrolling process?*
Miner: The bit planes need the horizontal-sync-counter output bits because they have to fetch over and over again across the line. Also, they need to do scrolling. The bit planes have a delay capability called horizontal scrolling built into them. This hardware scrolling actually delays the fetching of data so that it shows later on the screen. To do that, it's got to have a counter that causes 0 to 15 bits of delay.

What shows on the screen is the size of the screen display. The picture in memory can be quite a bit larger than that, and it can have multiple bit planes. There are two ways to change

(continued)

Faster CAD Input

The GTCO DIGI-PAD is a fast tracing device, a function the mouse can't perform at all. It's an absolute screen pointing device for direct cursor control. It can also provide direct, simple menu selection. The GTCO DIGI-PAD is a digitizer tablet in sizes including 12"x 12", 11"x 17", 20"x 20", 24"x 36", 36"x 48" and 42"x 60".

The DIGI-PAD is easily interfaced to PCs and is compatible with most PC/CAD software, such as AutoCAD™ and CADPLAN™.

The digitizer surpasses all other input devices for tracing and pointing and menuing. GTCO digitizers use patented electromagnetic technology for years of silent, maintenance-free operation.

Ask your dealer about the GTCO DIGI-PAD.

® DIGI-PAD and Micro DIGI-PAD are registered trademarks of GTCO Corporation.
™AutoCAD is a trademark of Autodesk Inc.
™CADPLAN is a trademark of Personal CAD Systems, Inc.

With the GTCO DIGI-PAD®

Micro DIGI-PAD® for small applications.

DIGI-PAD for large digitizing tasks.

GTCO Corporation
1055 First Street • Rockville, MD 20850
(301) 279-9550

Inquiry 171

what shows on the screen. One is to horizontally scroll smoothly 0 to 15 bits within a word. The other is to change the pointer a whole word value. So you can relocate the thing just by changing the pointer. If you come to the edge of the big picture, then you've got to do something in the software—block moves and so on.

COLLISION DETECTION

BYTE: *What about the information feeding over to the bit-plane controls and the whole interaction of bit planes and sprites? Collision detection has nothing to do with what shows; it just tells you when something has happened, right?*

Miner: Exactly. Collision detection is looking in real time at the simultaneous occurrence of objects. Sprites are on the 16 lines out of the sprite-serialize block, and bit planes are on the six lines out of the bit-plane-serialize block. Any simultaneous, real-time occurrence of more than one object at the collision-detection logic will be detected and stored in a latch in the collision-storage register. The program or the programmer can read this back out any time.

BYTE: *How do you know when there's an object here if there's always some sort of data on the line? If this line is low, then do you assume that it's not data?*

Miner: Right. Zero is always nothing. Zero is transparency.

BYTE: *But collision is more complicated than just "There are two things here."*

Miner: Collision control is quite complex. We've got an ability in this machine that I've never seen in any other machine before. Take a four-bit-plane playfield. Here's a sprite coming along. It can collide with that playfield by virtue of hitting any of those planes. This whole architecture is bit-plane-oriented rather than pixel-oriented. I can collide with any bit plane or mask any bit plane from the collision. Or I can actually invert the polarity of the bit plane with which I'll collide. The collision-control register decides which bit planes get looked at by the collision monitor and with what polarity. You can be very picky about what kinds of playfield the sprites collide with. By using all bit planes and getting the right polarity, you could have a collision with any individual color.

BYTE: *With 128 virtual sprites as a possibility and the various bit planes, it seems like you'd have an enormous number of things for the collision-control register to keep track of.*

Miner: Well, the collision-control register doesn't keep track of those virtual sprites. It only keeps track of real sprite-engine collisions. For real sprites, you use the collision-control register every vertical-blank time, and if a sprite collided during the previous frame, then you know that a collision occurred. ■

USE US...
TO SAVE A BUNDLE!!

We save educational, business and institutional buyers hundreds or even thousands of dollars on their computer supply needs. HOW ABOUT YOU?

We deal only in first quality, guaranteed products — all in stock for immediate shipment!!

BULK DISKETTES

JANUS FROM .. SS/DD 69¢ EACH 100 LOT DS/DD .. 79¢ EACH 100 LOT

VERBATIM FROM .. SS/DD 85¢ EACH 100 LOT DS/DD . $1.05 EACH 100 LOT

SENTINEL FROM .. SS/DD 75¢ EACH 100 LOT DS/DD .. 85¢ EACH 100 LOT

OTHER MAJOR MANUFACTURERS AVAILABLE, CALL...

SENTINEL COLOR DISK
SS/DD .... 89¢ EACH 100 LOT
DS/DD ... 99¢ EACH 100 LOT

3-1/2" DISKS for Macintosh & HP .... $1.89 EACH 100 LOT

Verbatim DATA CASSETTES ...... T300 $4.99 EA R300 $4.49 EA

DISK STORAGE BOXES
(many to choose from)
including our DX85A — Holds 100 disks
AT JUST .... $12.95 EA

PRINT WHEELS
Diablo .... low as $5.45
Qume .... low as 5.45
Wang .... low as 6.45
Royal .... low as 14.50
Nakajima .... low as 13.95
IBM Displaywriter .... low as 15.95
Ricoh .... low as 24.95
Silver Reed .... low as 12.95
ALL TYPES & STYLES IN STOCK.

PRINT RIBBONS
PRICES PER DOZEN RIBBONS
Epson MX 80 .... low as $3.49
Epson MX 100 .... low as 4.99
Diablo .... low as 3.49
Imagewriter .... low as 3.99
ALL ABOVE AVAILABLE IN COLOR AS WELL! WE STOCK RIBBONS FOR ALL MAJOR BRAND PRINTERS!!! CALL TODAY!!

CALL FOR VOLUME DISCOUNTS!!!

780 Trimble Road, Suite 608, San Jose, California 95131
800/437-0900 • 800/435-9700 in CA
P.O.'s accepted from Institutional, Educational & Government Accounts. Dealer and Distributor Calls are Welcome.

C.O.D. ADD $2.00
SUPPLY ORDERS $25.00 MIN.

Inquiry 340