A Single-Chip Multiprocessor For Multimedia: The MVP

Karl Guttag, Robert J. Gove, and Jerry R. Van Aken
Texas Instruments

We defined the multimedia video processor (MVP) to accelerate applications with heavy image and graphics processing requirements. Here we give an overview of the architecture.

Currently under development is the digital technology to capture, store, enhance, transmit, and process still images and full-motion video. With literally hundreds of billions of dollars worth of images needing handling each year, the digital processing of images will be the largest growth market in electronics products over the next decade.

Today's graphical user interfaces represent the consolidation of the separate text and graphics systems of the 1970s and early 1980s into a unified bitmapped environment. Similarly, the computer industry will see the merging of graphics, imaging, video, and audio functions into a fully integrated multimedia environment in the 1990s.

Supporting a complete multimedia environment will require a processor architecture that can support over 2 billion operations per second. To deliver this capability with the latest CMOS technology clearly requires parallel processing techniques. Researchers have proposed a number of parallel processing techniques for image and graphics processing,[1-4] but most architectures are narrowly directed at a few classes of algorithms.

0272-17-16/92/1100-0053$03.00 © 1992 IEEE

Our architecture incorporates a variety of parallel processing techniques to deliver very high performance to a wide range of imaging and graphics applications.

We refer to our architecture as the multimedia video processor, or MVP. The MVP combines, on a single semiconductor chip, multiple fully programmable processors with multiple data streams connected to shared RAMs through a crossbar network. Each of the independent processors can execute many operations in parallel every cycle. The architecture is scalable and supports different numbers of processors to meet the cost and performance requirements of different markets. We have demonstrated through software simulations that this architecture can support a diversity of image processing, graphics, and audio applications.

Target environment

In developing the MVP, we considered the range of office needs we could satisfy by making orders of magnitude more processing power available on a desktop. While much of the public discussion involving multimedia has dealt with image compression, compression represents only a necessary first step—one that reduces the cost of storing and transmitting images. Looking beyond compression, some of the biggest benefits will come from the ability to enhance, manipulate, and recognize objects once we have captured images in digital form.

Our architectural analysis focused on the following three groups of applications: document image processing (including recognition), image generation (graphics), and compression (image and audio transmission or storage). While image processing and graphics involve the most processing-intensive applications, audio enhancement, echo cancellation, compression and decompression, generation, and recognition can also be very demanding.

Figure 1 shows our concept of a system that integrates video teleconferencing, document processing, and audio into a single environment. We expect to see these applications delivered on a variety of platforms, including PCs, workstations, and X terminals. In addition to the desktop environment, this technology will redefine other office products, such as digital copiers, hard-copy units, video-conferencing systems, transaction processing, and security systems.

Video compression requirements

International standards—essential for growth in image-processor markets—are making earlier, proprietary standards obsolete. Both vendors and users benefit from standards that support the exchange of information among systems made by different manufacturers.

The list below briefly describes the three most prominent internationally proposed standards for image compression and decompression:

• JPEG (Joint Photographic Experts Group): Originally targeted at the storage and retrieval of high-quality still images at any resolution, JPEG is also used for video image compression. However, it requires higher data rates than the full-motion video standards described below.

• MPEG (Motion Picture Experts Group): Originally targeted toward video playback at CD-ROM rates (1.2 megabits per second). Not satisfied with the image quality of the original MPEG standard, developers of digital VCRs and digital cable movie distribution systems are working toward enhanced versions. Dubbed MPEG++ and MPEG 2, they support higher data rates.

• Px64: International standard for video teleconferencing (two-way video and audio) using data rates that can be any multiple of 64 Kbits per second (hence the name). The standard encompasses a variety of system tasks, including image compression (CCITT H.261), data formatting, and audio. Image quality and frame rates vary based on the data rate. It can be used to send and store video over any network.

Each of the above standards uses a variety of techniques to compress images. They all use discrete cosine transforms (DCTs), which are integer multiplication-intensive. They take advantage of entropy coding

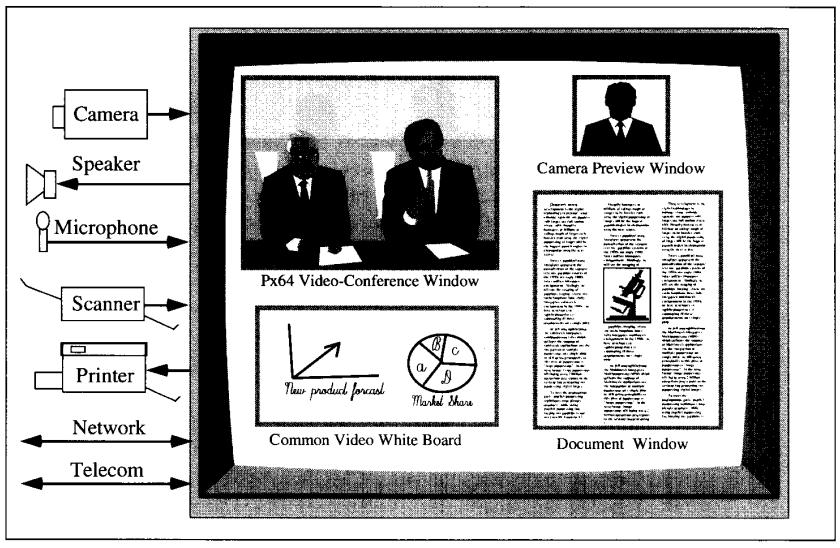


Figure 1. An integrated media environment.

(using variations on Huffman coding), which involves bit-field manipulation and table lookup operations. The video standards MPEG and Px64 both perform motion estimations that require massive numbers of integer operations. The full-motion standards also require audio compression and decompression.

While these standards define the data formats and suggest coding techniques, they permit a large degree of latitude in their implementation. The full-motion standards incorporate decision-making processes that selectively apply a variety of compression techniques in an attempt to achieve the best image quality for a given data rate. Developers can use both pre- and post-decompression processing (not defined by the standards) to reduce artifacts introduced by the compression process. The need to dynamically choose among a variety of compression techniques suggests that designers ought to base the system on a programmable processor.

Table 1 shows that, for a sample implementation of Px64, roughly 1.2 billion RISC-like operations per second are required to execute the key elements of Px64's H.261 compression and decompression at 30 frames per second. The almost 1.2 billion total operations include over 40 million multiplies, each counted as a single operation. The total does not include any pre- or post-processing, audio processing, data transmission formatting, or other system functions. Note that several of the functions in the table require around 100 million RISC instructions per second, which is roughly the speed of today's fastest RISC microprocessors.

Table 1. Typical RISC processing requirements for CCITT H.261 (Px64) video compression and decompression.

Function	RISC-Like MOPS
Motion estimation - block matching	608 (51.0%)
Coding mode decisions	40 (3.4%)
Loop filtering (encode and decode)	110 (9.2%)
Pixel difference	18 (1.5%)
DCT (encode)	74 (6.2%)
Inverse DCT (encode and decode)	192 (16.1%)
Threshold/quantization/zig-zag scan	50 (4.2%)
Bit stream encode	17 (1.4%)
Reconstruction (encode and decode)	62 (16.1%)
Bit stream decode and inverse quantization	22 (1.8%)
Total	1,193 MOPS

General image processing

Much has been written about general digital image processing,[5,6] and standards groups are now looking at issues beyond image compression. Recent work has focused on the interchange and processing of digital images. The proposed XIE (X Imaging Extension) standard,[7] for instance, defines a standard mechanism for transmitting images and also for manipulating those images with a set of operations reproducible across a variety of systems.

Image processing algorithms typically require massive numbers of arithmetic operations. These algorithms include convolution, warping, histograms, halftoning, color-space conversions, median filtering, fast Fourier transforms (FFTs), and morphology. Each of these requires on the order of 10 to 100 operations per pixel. Dealing with color further adds to the processing work load. With images frequently containing more than 1 million pixels each, the processing task can be formidable.

While the computational demands of image processing are quite high, the data bandwidth requirements can be just as demanding. Image processing algorithms can require literally billions of bytes per second of data bandwidth. Over a given period of time, these algorithms tend to work within localized regions of data (for example, rectangular patches within the image) and make repeated accesses to the same locations. For this reason, a single-chip processor can achieve the required data bandwidth by loading data into on-chip memory, from which it can be processed at high speeds. Also, most of these algorithms work their way through external image memory in a predictable pattern that makes it possible to load the data on chip before a processor needs to process it. Loading data onto the chip and saving results to external memory can still require the transfer of over 100 million bytes per second.

Desktop document image processing

One area of multimedia that has received relatively little attention is the single most pervasive medium for communication in the office: paper. In many an office today, the paper just piles up for lack of a good way to deal with it. For a number of practical reasons, paper is likely to remain a favored medium for both reading and publishing for the foreseeable future. Predictions of computers reducing paper in the office will not be realized until computers can input paper documents at least as fast as they can generate them.

Meanwhile, the technology is at hand to help office workers manage paper documents much more effectively. Desktop document image processing (DDIP) systems will soon be widely available to process both paper and electronic documents. Because these documents often contain both text and embedded pictures and graphics, the system must

1. distinguish between the regions of the document that contain text and nontext,
2. perform optical character recognition on fonts of any size, style, and orientation, and
3. compress the nontext images.

Once the system has recognized the text, it can parse the text to aid in sorting, filing, and alerting the user to key topics based on an individually defined profile. In effect, these systems will act as information processing assistants.

Optical character recognition (OCR) of printed text requires an assortment of processing functions, from low-level image conditioning to high-level functions such as detailed feature extraction, matching, scoring, and decision making. The current trend in OCR involves detecting features in both frequency and spatial domains, then comparing results to improve accuracy. The pervasive use of decision-making processes favors multiple-instruction, multiple-data-stream (MIMD) architectures that can readily handle branches in program flow.

At 10-page-per-minute rates (roughly the speed at which office laser printers can generate documents) and scanning resolutions of 300 to 600 dots per inch, the system must process from 1.5 to greater than 6 million pixels per second. Individual functions like edge operations, convolutions, and FFTs require from 20 to more than 100 operations per pixel, and a series of these functions are required to perform OCR. With multiple algorithms employed to improve accuracy, the processing requirements can easily exceed 1,000 millions of operations per second (MOPS), or 1 GOPS.

Basic graphics processing requirements

Image processing is destined to become an integral part of workstations and PCs, but it will be performed within a graphical environment. Higher resolution systems with 24 bits or more per pixel will become commonplace on the desktop. The speed and quality of desktop graphics will benefit from the raw performance in pixel manipulation that image processors will provide.

Graphical user interfaces (GUIs) such as the X Window System and Microsoft Windows use bit-aligned block transfers (bitblts) extensively for text and window manipulation. In addition to basic bitblts, graphics applications require primitives such as the following:

- text with color expansion
- filled polygons
- lines
- curves (splines and Bezier)
- antialiased lines
- Gouraud- and Phong-shaded surfaces

Compared with image processing operations, bitblts[8,9] are simple bit-aligned moves that include minimal (often only Boolean) processing of pixel values. Both bitmapped character fonts and cached stroke fonts usually are stored at 1 bit per pixel and are "expanded" (see the sidebar "Lessons learned from the 340 family") into color as they are drawn. Color expansion can be time-consuming unless the hardware supports it directly.

From our previous experience in developing the 340/TIGA graphics architecture,[14,9] we knew that a processor requires special hardware such as barrel shifters, color expanders, video RAM block-write control, and splittable arithmetic logic units in order to efficiently support graphics primitives.

In the 340, however, instructions to perform bitblts[8,9] and draw lines were executed in a sequence of states controlled by on-chip microcode ROM. This meant that software programs could only use the graphics hardware in the finite number of ways supported by the microcode. From a chip development perspective, given all the combinations of operations the 340 supports, writing and debugging the microcode was a significant challenge. With each graphics standard having slightly different requirements, and given the limitless number of image processing algorithms, we sought to develop an architecture that would allow virtually any multicycle operation to be efficiently implemented as a sequence of single-cycle instructions.

The MVP's parallel processors have many architectural features to support pixel processing. To avoid problems and limitations of the microcoded approach, the parallel processors have single-cycle instructions (that is, no microcode) and give software direct control of the hardware. Making the hardware directly accessible allows programmers to write software to efficiently execute multicycle operations as a sequence of single-cycle instructions. One challenge was to devise how we could use sequences of the parallel processor instructions as building blocks to perform multicycle operations.

The expander hardware provides an example of how we made the parallel processor's hardware directly accessible to the programmer. While the 34010 had "expander" hardware, it was dedicated to converting 1-bit-per-pixel bitmaps into two colors.[14] This "color expansion" is commonly used to draw text, usually stored at 1 bit per pixel, into a color display buffer. The parallel processor architecture generalized the expander to simply take 1-bit values from the multiple flags register and replicate them to fill a 32-bit word that is then routed to the ALU. This lets us translate between 1-bit and color representations of pixels.

In the example of Figure A, a parallel processor expands the color of four pixels from 1 to 8 bits each. At the top of the figure, the four least-significant bits of the multiple flags register have been loaded with four adjacent 1-bit pixels from a bitmap. The parallel processor's expander logic forms a 32-bit merge mask by replicating the value of each of the four 1-bit pixels in the multiple flags register to 8 bits. The "0 color" register in the figure has been loaded with four 8-bit pixels that represent the color to which 0s in the source bitmap are to be expanded. Similarly, the "1 color" register contains four 8-bit pixels that represent the color to which 1s in the source bitmap are to be expanded. The programmer specifies these two color registers, along with the merge mask, as the inputs to the three-operand ALU. The programmer also has the ALU perform a three-

operand Boolean operation that selects between the bits contained in the two color registers based on the value of the corresponding bits in the merge mask. In the result shown at the bottom of the figure, the ALU sets the 8-bit pixels corresponding to 1s in the original bitmap to the 1-color value; it sets pixels corresponding to 0s to the 0-color value. The parallel processor executes the entire color-expand process shown in Figure A in a single machine cycle.

The implementation of the max (maximum) raster-op illustrates the usefulness of the parallel processor's ability to rapidly transform an image back and forth between 1-bit and *n*-bit representations of pixels. By definition, the max operation compares corresponding source and destination pixels and replaces the destination pixel with the larger of the two values.

Figure B shows how the parallel processor performs the max operation on four 8-bit pixels in only two machine cycles. At the top of the figure, four adjacent 8-bit pixels from the destination have been loaded into one 32-bit register and four source pixels have been loaded into another. During the first cycle, the ALU is split into four 8-bit segments. The source word is subtracted from the destination, and the carry-out bits from the four ALU segments, each of which indicates whether a particular source pixel is greater than the corresponding destination pixel, are saved in the multiple flags register. In the second cycle, the parallel processor's expander logic forms a 32-bit merge mask by replicating the value of each of the four carry-out bits in the multiple flags register to 8 bits. The programmer specifies source and destination pixel values, along with the merge mask, as the inputs to the three-operand ALU. The programmer also configures the ALU to perform a three-operand Boolean operation that selects between the bits contained in the source and destination pixel values based on the value of the corresponding bits in the merge mask. In the result shown at the bottom of the figure, each of the four 8-bit pixels represents the larger value of the corresponding source and destination pixels.

In contrast, the 340 architecture, when performing the max function, stores the carries out of the ALU segments in latches accessible only by the microcode. This prevents the programmer from using the hardware in applications not explicitly supported by the 340's microcoded instruction set. The parallel processor's multiple flags register makes this intermediate information available to the programmer.

The parallel processors can synthesize a vast number of different graphics operations by combining the various ALU operations with the various routings of data to the ALU, the expander, and other hardware. In addition to color expansion and max, these operations include min, add or subtract with saturation (clamping to a maximum or minimum value), transparency on source or destination, and *z*-buffering. And while we originally intended the expander to accelerate graphics operations, we have found that its usefulness extends as well to image processing and other applications. For example, the expander hardware is used in the key inner loops of algorithms that perform motion estimation for the Px64 and MPEG video compression standards.

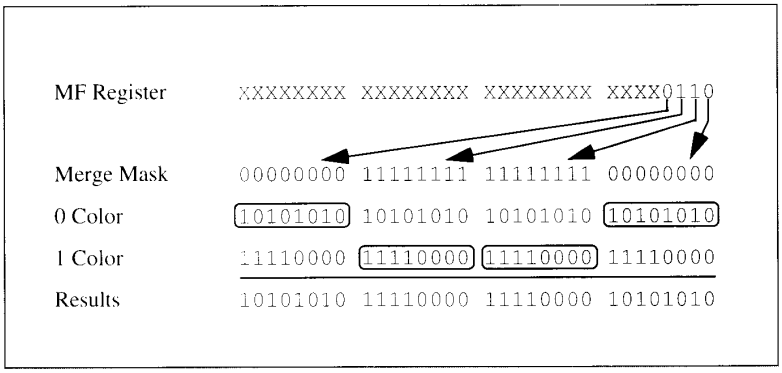


Figure A. A parallel processor color-expand example.

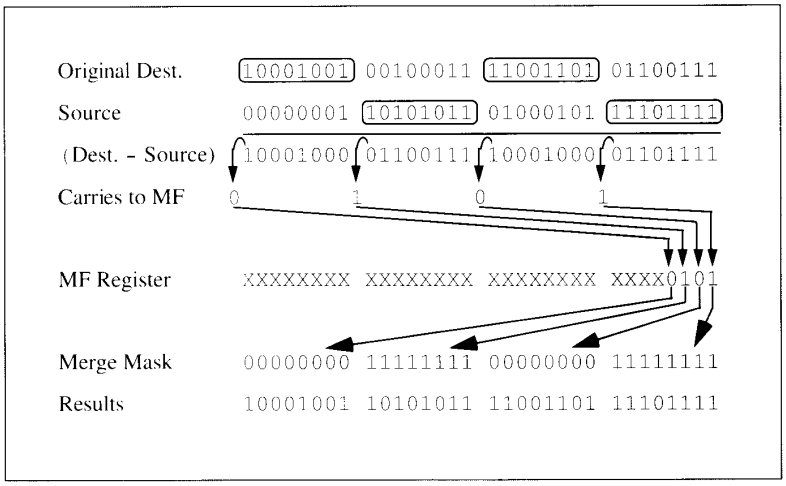


Figure B. A parallel processor max raster-op example.

With more powerful processors, many of the simple, single-color primitives such as lines (for example, Bresenham) and fills are limited only by memory bandwidth. The more sophisticated drawing functions, including antialiased lines and shaded polygons, have computational requirements similar to image processing functions—they involve high-speed multiplies and accumulations on pixel values.

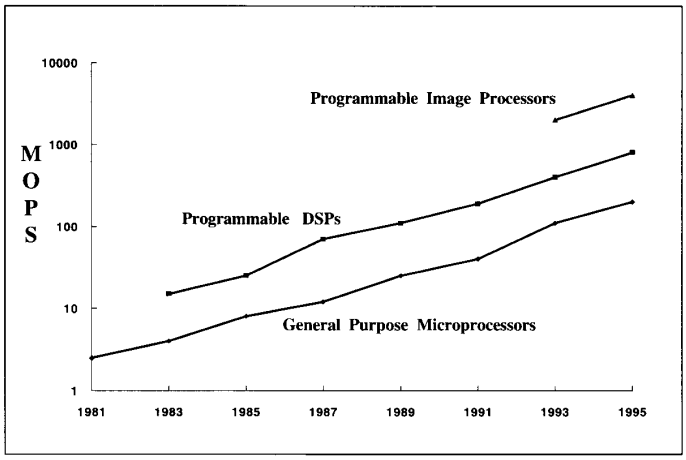


Figure 2. Trends in programmable processor performance.

Floating-point requirements

Well-known algorithms for constructing both 3D graphics and 2D geometric shapes (such as font outlines) require extensive amounts of floating-point calculations. Graphics standards such as PHIGS (Programmers' Hierarchical Interactive Graphics System) and GL (Graphics Language) can be floating-point intensive. Some image processing applications, such as medical imaging, can require higher precision integer and floating-point calculations to maintain accuracy. Audio algorithms can also require floating-point calculations, as is the case for the latest audio compression standard, G.728, for Px64.

The calculation of an inner product—the key primitive in a homogeneous 3D transform—involves accumulating the results of four multiplies. The frequency-domain transforms used in image processing have kernels (butterflies) that consist of only one or two multiplies plus two additions and/or subtractions. The key to performing this type of application efficiently is to reduce the overhead required to set up floating-point operations.

After looking at a variety of graphics and imaging algorithms, we concluded that a single floating-point unit in parallel with one to eight integer processors would give a good balance of floating-point and integer processing capabilities.

Roots of the architecture

We drove the definition of the MVP in the top-down direction based on the algorithms it had to support, and in the bottom-up direction based on new semiconductor technology. Our goal was to define an architecture that could improve the performance of the target applications by at least an order of magnitude. We measured our efforts against general-purpose reduced instruction set computers fabricated with the same generation of semiconductor technology that we planned to use.

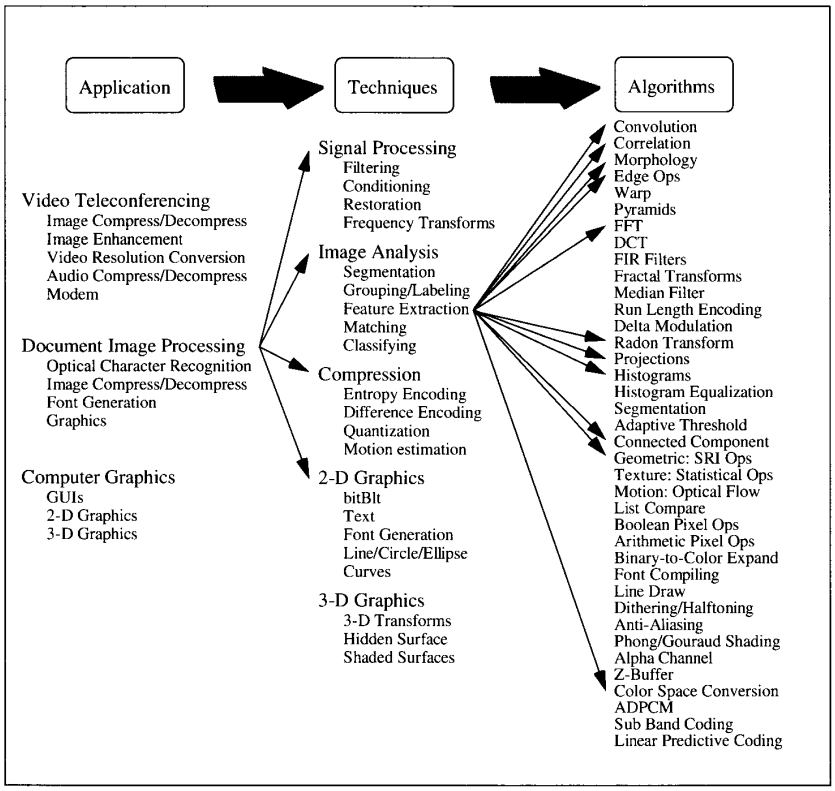


Figure 3. Algorithm analysis flow diagram.

Flexible architecture versus dedicated hardware

Imaging applications require processing techniques that range from multiplication-intensive filtering and frequency domain transforms, to massive numbers of simple arithmetic operations on pixel data, to bit-field extraction and table-lookup operations. Most image processing applications require adaptive algorithms that select among a variety of techniques according to characteristics detected in the image. Because digital image processing is still in its infancy, new algorithms will continue to emerge.

While providing special hardware to accelerate the most demanding tasks is important, overspecialization also has disadvantages. If a processor performs any of its required functions poorly, the resulting bottleneck might degrade overall performance to the point where it cancels any gains resulting from acceleration of the other functions. In the MVP, we incorporated acceleration features, but in a way that avoided dedicating the hardware to a narrow set of functions. We defined the instruction set to give the programmer direct access to hardware functions without framing them within the constraints of specific algorithms. For example, while we spent considerable effort on enhancing the discrete cosine transform (DCT) performance of the MVP's processors, we did not want to have dedicated DCT units for the following reasons:

1. With parallel hardware multipliers, ALUs, and address units, the parallel processors' performance was such that less than 20 percent of the processing time would be required for DCTs when used on so-called DCT-based compression methods. Speeding up DCTs further would have diminishing effects on overall performance. (For example, doubling DCT speed would speed up the whole application by less than 10 percent!)

2. The same hardware used to perform DCTs can be used for a variety of other multiplication-intensive algorithms, including convolution filters, finite-impulse-response (FIR) filters, optical character recognition (OCR), color-space transforms, graphics, and modems.

3. We could obtain high processing efficiency if the application could select between different DCT algorithms (Chen, Lee, Veterali, and so on) to make speed and precision trade-offs.

4. While compression standards today use DCTs heavily, we expect to see better approaches that may not be DCT-based in the future.

Figure 2 shows the trends in programmable processor performance, in terms of millions of operations per second. General-purpose RISC processors typically perform only one operation per instruction. Developers reduce machine cycle time by using faster semiconductor technology and sometimes obtain further speedups by executing multiple instructions per cycle through a combination of pipelining and superscalar techniques. But these techniques rapidly reach a point of diminishing returns—beyond two instructions per cycle. Digital signal processors (DSPs) have demonstrated that providing multipliers and other hardware to perform many operations in parallel enables them to deliver much higher levels of performance to targeted applications than can general-purpose processors.

In defining the MVP, we analyzed[10] imaging, graphics, and audio processing applications. Figure 3 shows our process for analyzing the requirements of target applications. Each of these applications can involve many different processing techniques, which we grouped under the headings of signal (image and audio) processing, image analysis, compression (data, image, and audio), 2D graphics, and 3D graphics. Each technique can, in turn, employ a number of different algorithms. The list of algorithms shown in Figure 3 is hardly exhaustive, and you can implement each algorithm in many different ways.

The arrows in Figure 3 show one example path from document image processing (DIP). Capturing documents, recognizing text, and reproducing documents requires a number of different techniques. Taking just one of these techniques—feature extraction used in text recognition—leads to a large number of algorithms. The programmer then has to decide which algorithms to use and determine the best way to implement them.

To fulfill the target applications' performance requirements, image processors must be able to execute even more operations in parallel than can existing DSPs. To achieve the massive parallelism required, we employed a battery of techniques, including multiple parallel processors, split arithmetic logic units (ALUs), parallel multipliers, very large instruction words (VLIWs), and special hardware to accelerate critical imaging and graphics operations.

The MVP was influenced by Texas Instruments' practical experience in graphics (TMS340 family[11] and video RAMs[12]) and digital signal processing (TMS320 family). We learned from the TMS340 and TMS320 processor families that a programmable architecture adapts best to emerging applications. (Refer to the sidebar "Flexible architecture versus dedicated hardware.") By combining TI's ongoing research into image processing with our experience in application-oriented processors, we defined an architecture that can provide high performance over a wide range of office and business tasks, yet adapt to evolving imaging and graphics standards.

We refined the architecture by functionally simulating it with a series of imaging and graphics algorithms. We tuned the instruction set and modified the architecture of the processors as our testing of key algorithms uncovered bottlenecks. We benchmarked both inherently serial functions, such as the JPEG's modified Huffman decompression, and

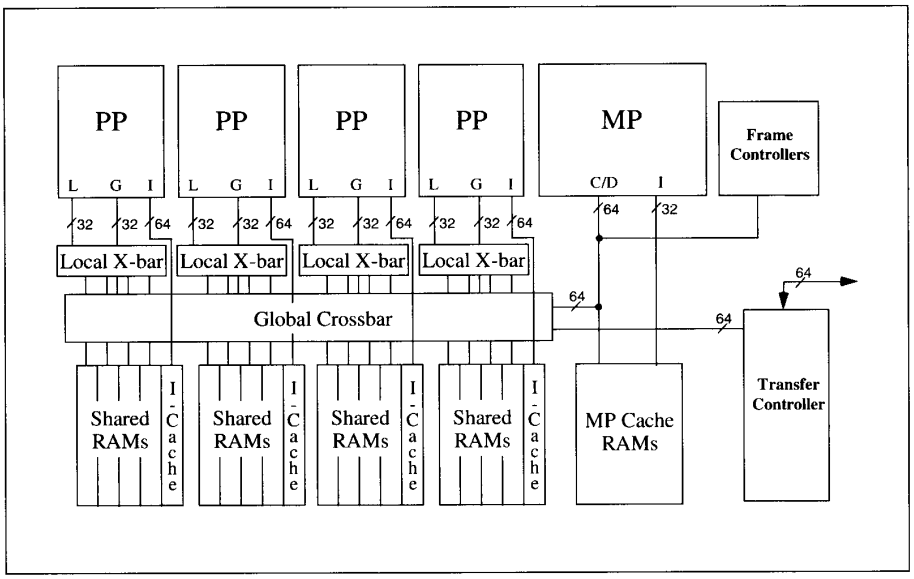


Figure 4. High-level block diagram of the architecture with four parallel processors (PPs) and a master processor (MP).

inherently parallel operations, such as Px64's motion-estimation algorithm.

Architecture for multimedia

Figure 4 shows a block diagram of the major functional blocks of the MVP. The key elements include

1. The master processor (MP): A general-purpose RISC processor with integral floating-point unit.
2. One or more parallel processors (PP): While the diagram shows four parallel processors, the architecture readily supports from one to eight. We designed the parallel processors to handle digital signal processing, pixel processing, and other massively parallel integer or fixed-point processing tasks. Each parallel processor can make two parallel data accesses into the on-chip RAM per machine cycle.
3. Transfer controller: This controller manages the hardware interface between the on- and off-chip memories. It supports both automatic cache servicing and processor-directed (by either the master processor or parallel processors) multidimensional direct memory access (DMA) transfers. We designed the transfer controller to interface with a wide range of dynamic RAM, video RAM, and static RAM memory devices.
4. Shared RAMs: The on-chip data memory for the parallel processors is contained in a number of individual on-chip RAM modules, each accessible in parallel over a crossbar switching network.
5. Local and global crossbars: The on-chip crossbar switching network allows the processors to access each RAM module independently and in parallel with accesses of the other RAMs. The crossbars support cycle-by-cycle connection between the processors and the various shared RAMs on the chip. The parallel processors each have two data memory ports, the master processor has one data memory port, and the transfer controller has one data memory port, all of which can access shared RAM in parallel. The master processor, transfer controller, and each of the parallel processors can access any portion of the shared RAM via the global crossbar. Additionally, each parallel processor has a local crossbar through which it simultaneously can access a group of RAM modules local to that processor.
6. Cache RAMs: The master processor has both instruction and data caches, and each parallel processor has its own instruction cache. The cache controllers reside within each processor.
7. Dual frame-buffer controllers: These support programmable video timing to control both display and capture.

Master processor

The master processor (see Figure 5) is a general-purpose RISC processor with an integral IEEE-compatible floating-point unit (FPU). The processor has a 32-bit instruction word and can load or store 8-, 16-, 32-, and 64-bit data sizes. It has thirty-one 32-bit general-purpose registers, with register 0 hardwired to a constant value of 0.

Like most RISC processors, the master processor has orthogonal three-operand data-path operations, load or store memory operations, and one-cycle delayed branches with optional annul. Most instructions execute in a single cycle.

The master processor also has features less typical of RISC processors. The register file is common to both the floating-point and integer operations. Scoreboard logic keeps track of the registers that will receive the results of loads and floating-point operations, automatically preventing use of these regis-

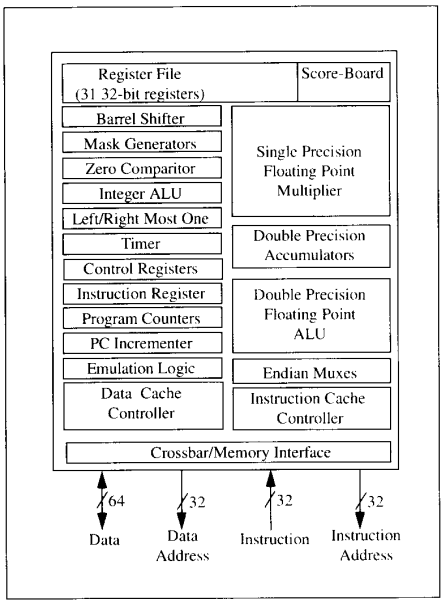


Figure 5. Master processor block diagram.

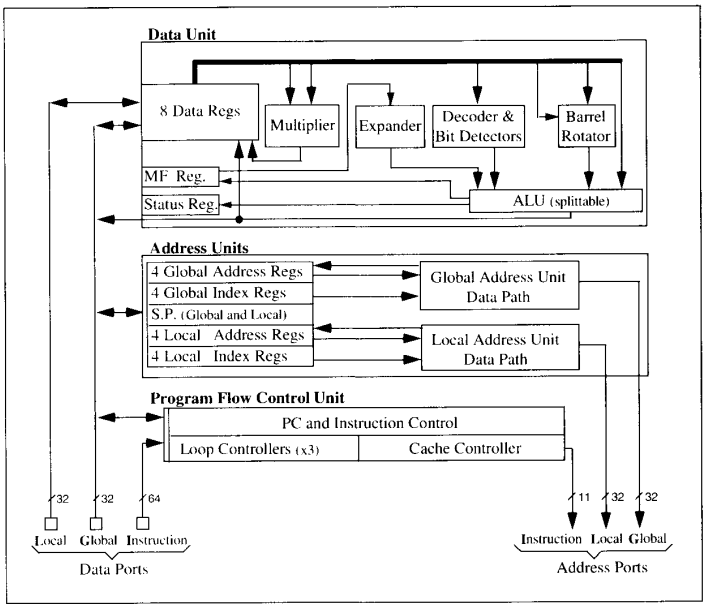


Figure 6. Parallel processor block.

ters until they have been updated. The addressing modes support the optional updating of the base-address register with the result of the address computation.

As its name implies, we intended the master processor to be the main supervisor and distributor of tasks within the chip. With its RISC architecture, it is designed to support efficient compilation of programs written in high-level languages. The master processor services external interrupts, although, depending on the application's requirements, the master processor might in turn initiate or interrupt tasks on the parallel processors. The master processor is responsible for communicating with external processors. It can handle many nonrepetitive tasks and simple interrupts so that the parallel processors can execute more efficiently.

Because of its FPU, the master processor is also the preferred processor on the chip for performing high-precision and floating-point math. Beyond system control and other more general-purpose tasks, the master processor will likely process audio signals (which generally require higher precision but fewer operations) and 3D graphics transformations. In applications such as medical imaging, the master processor can perform high-precision frequency-domain operations.

Floating-point considerations

In addition to the floating-point operations found in other RISC instruction sets, we added a set of special instructions targeted at 3D graphics and imaging to the master processor. The master processor's floating-point unit can initiate independent multiply and ALU operations every cycle.

Typical of fast FPUs, the master processor's FPU is pipelined. Pipelining requires special attention to the data flow in order to efficiently execute the small inner loops associated with 3D transforms and imaging operations.

In addition to a set of floating-point instructions similar to those found in other RISC architectures, we defined a special set of floating-point instructions designed to work together without causing pipeline delays. These instructions support initiating in each successive cycle a new floating-point multiply, add or subtract, a load or store, and an increment of the address pointer.

Parallel processors

In the parallel processor architecture, we merged our experience with digital signal processors, graphics processors, and parallel processing systems. While DSPs have historically excelled at multiply-accumulate-intensive processing, they have not done as well in performing bit and pixel manipulations (necessary for image compression and graphics). Graphics processors perform basic pixel movement and bit manipulation well, but they have not done well at DSP and other math-intensive operations. We designed a single parallel processor to be more powerful at general integer DSP or bit and pixel manipulation than existing single-chip processors.

Figure 6 shows a block diagram of a parallel processor and its four major functional units: the data unit, two address units, and the program flow control unit. These parallel execution units let each parallel processor perform many operations in each cycle. The parallel processor's architecture supports the massive processing associated with frequency-domain transforms (for example, discrete cosine transforms), correlation, filters, and the pixel manipulation required by imaging and graphics. To specify all these parallel operations,

the parallel processor uses a very large instruction word of 64 bits.

Because it is a fully programmable processor, you can program the parallel processor in a high-level language such as C, as well as in assembly code. Just as with the newer DSP architectures, we expect the bulk of code to be written in C, with the key inner loops and other time-critical operations written in assembly code. When the ultimate in performance is required, a programmer can specify the equivalent of 3 to 15 RISC instructions in one parallel processor 64-bit opcode.

Parallel processor data unit

The parallel processor data unit has eight data registers, a status register, and a special multiple flags register. The data registers are "octal ported" so that up to eight different reads or writes take place in a single cycle. The status register contains ALU-result status bits that can be used with conditional instructions. The multiple flags register captures the multiple status results (flags) from split-ALU operations and provides the bits that are input to the expander (see below) for use in ALU operations. In addition to their special access modes, the programmer can access the multiple flags and status registers in the same manner as any other register.

The expander takes 1, 2, or 4 bits in the multiple flags register and replicates them 32, 16, or 8 times to fill out a 32-bit word that is routed to the ALU (see the sidebar "Lessons learned from the 340 family"). Conceptually, each status bit loaded into the multiple flags register from an n-bit segment of the split ALU represents a transformation from n bits to 1 bit, and the expander performs an inverse transformation of each multiple flags bit to n bits. We have found this hardware helpful in performing a number of graphics and imaging operations.

The barrel rotator can rotate an incoming 32-bit quantity by 0 to 31 bits. The decoder generates masks from 0 to 32 bits in length that are used with the barrel rotator to generate signed or unsigned right or left shift operations. (An n-bit mask has the value $2^n - 1$.) You can use the barrel rotator and decoder to perform field extraction and bitblts.

The bit detectors perform leftmost- and rightmost-bit detection. These functions often aid in decoding entropy-compressed (for example, Huffman-encoded) data streams.

The 32-bit ALU performs addition, subtraction, and Boolean operations. The ALU can support the 256 three-operand Booleans that are supported by graphics environments such as MS Windows. Status logic associated with the ALU sends carry-out, zero, sign, and overflow bits to the status register. The ALU can be split into two 16-bit or four 8-bit ALUs during arithmetic operations. During split-ALU operations, a zero or carry-out bit is available from each ALU segment and can be saved in the multiple flags register.

The multiplier performs 16×16 multiplies and produces 32-bit results in a single cycle. Special rounding hardware on the multiplier's output maintains the various precision levels required by the compression standards.

Parallel processor's two address units

The parallel processor has two nearly identical address units. Each address unit can independently perform a load from, or store to, the on-chip shared RAM. Each processor has associated with it a number of RAMs considered local to that processor.

During each machine cycle, the parallel processor's two address units can independently access the on-chip RAMs in parallel. One of the address units can access the parallel processor's local RAMs via its own local crossbar while the other unit simultaneously accesses any of the shared RAMs via the global crossbar. In the unlikely event that both address units attempt to access nonlocal RAMs in the same cycle, hardware detects the conflict and the instruction takes two cycles to complete the two accesses.

Each of the two address units contains four address registers and four index registers, and they share a single stack-pointer register. These registers not only facilitate address calculations, but also serve as source or destination registers both for the data unit's ALU operations and for memory loads and stores.

An address unit's 32-bit data paths can add or subtract an index register or immediate value to or from an address register. The programmer can optionally update the address register with the results of address calculations. To support pre- or post-indexing, the address output to memory can either be the result of the address computation or the original contents of the address register.

Program flow controller

The parallel processor's PFC is responsible for program counter increments, branch-and-loop control, interrupt-context switching, instruction-cache control, and instruction decoding.

The PFC contains a program-counter register, two program-counter-history registers, and loop-control registers. Like all user-visible parallel processor registers, these registers can be accessed by both the data unit and the address unit. Similar to the PDP-11,[13] a program branch occurs whenever the program counter either is the result of a data-unit operation or is loaded by an address-unit operation. We designed the program-counter-history registers for use as return pointers from jumps, interrupts, and emulation-controlled changes to program flow.

The parallel processor supports conditional execution of data-unit and address-unit operations. Status-register bits set by a previous instruction determine whether results are saved from data-unit operations and transfers are performed by address-unit operations. Conditional execution of an instruction is often more efficient than conditionally branching around one instruction. In keeping with treating the program counter the same as other registers, the parallel processor has no explicit conditional-jump instructions. The architecture performs conditional jumps by specifying an operation for which the program counter is a conditional destination.

DSP architectures have already demonstrated the effectiveness of including a single hardware unit to support zero-overhead control of program loops. In analyzing the algorithms for imaging and graphics, we determined that a number of key algorithms could make good use of up to three sets of loop-control units. Each loop control unit can also support zero-overhead branches or returns. You can use zero-overhead branches to support branching from, and returning to, a tight loop.

The need to branch out and back within a tight loop was driven by the massive number of different pixel processing operations—including Booleans, arithmetics (with saturation), max, min, transparency, and color expand—that bitblts can perform. We wanted programmers to be able to write the software to set up the inner loop without regard to the type or number of cycles required for the raster-op. At the same time, we knew we could not afford to add branch-delay penalties to simple raster-ops with inner loops of one or two cycles per word.

Crossbar and on-chip memory

Two of our first concerns in defining the architecture were how to keep the processors from having to wait for data and how the processors would communicate with each other. Our solution was to have some of the on-chip memory dedicated to instruction and data caches, and to have a large number of on-chip shared RAMs connected to the processors through a crossbar switching network. The crossbars consist of address decoders, buses, and multiplexers that let the parallel processors, master processor, and transfer controller connect to any of the shared RAMs on the chip.

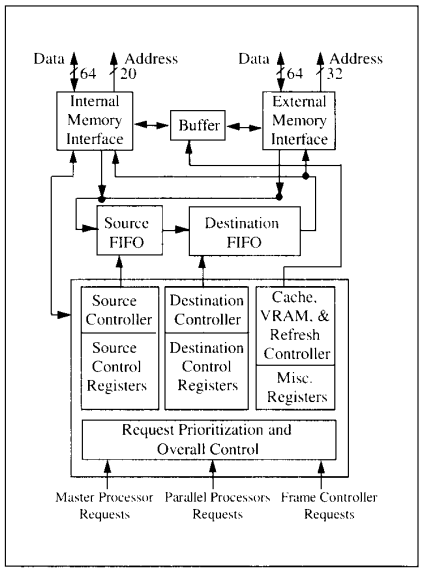


Figure 7. Transfer controller block diagram.

Many of the imaging and graphics algorithms, such as convolution, morphology, DCT, FFT, and antialiasing, require multiple accesses within a group of pixels. Pulling these groups on chip once and keeping intermediate results on chip greatly reduces the number of accesses to external memory. The transfer controller can fetch multidimensional groups of data before a processor needs them. It can also save old results while the processor begins to work on the next block of prefetched data.

Rather than supply a few large RAM modules, the architecture uses many smaller wide-word (accessible up to 64 bits wide) modules that collectively provide tremendous memory bandwidth. The following accesses can occur in parallel in a single cycle:

- Each parallel processor can make two independent 32-bit data accesses and a 64-bit instruction fetch.
- The transfer controller can make a 64-bit DMA-like transfer to save old results or load in new data or instructions for the processors.
- The master processor can make one 64-bit data access and a 32-bit instruction fetch.

Because these RAMs can be shared via the crossbars, memory can be allocated among the processors as needed. Shared memory eliminates the need to move results from one processor to another, which preserves on-chip bandwidth.

The crossbar connections that tie the parallel processors, master processor, and transfer controller to the shared RAMs are dynamically configured on each successive cycle. If more than one processor contends for the same RAM module during the same cycle, a round-robin arbitration method allows one processor at a time to access that memory. Memory contention is rare because of the number of on-chip memories and because algorithms can be structured to reduce contention.

The crossbar structure supports a variety of different parallel processing models. The processors can share results on a cycle-by-cycle basis (fine-grain parallelism) or can save up a number of results (course-grain parallelism), then have another processor use those results without having to physically move the data.

The crossbar switch provides a classic example of an architectural concept that becomes practical in terms of size and speed only with the integration of multiple processors onto a single chip. With the four-parallel-processors implementation, for example, the crossbar will have well over 500 data, address, and control lines that must switch in a few nanoseconds.

The shared memory structure also lets us scale the architecture by adding or subtracting parallel processors. As we add more parallel processors to the chip to obtain the desired performance level, we extend the crossbar and add shared memory.

Transfer controller and memory interface

The transfer controller, shown in Figure 7, prioritizes, schedules, and transfers data between on- and off-chip memories. These transfers can be initiated automatically to perform cache servicing, dynamic RAM refresh, and video-RAM serial-register transfers. Any of the processors can, un-

der software control, initiate block transfers of data between memory spaces.

Both the source and destination arrays of the block transfers can have up to three dimensions (for example, the x and y dimensions of a rectangular area of an image plus the stride from one rectangular area to the next). The transfer controller contains source and destination FIFOs to support byte-aligned transfers and to permit the efficient use of high-speed dynamic-RAM and video-RAM column-access modes.

Having the transfer controller support independent transfers of multidimensional byte-aligned data allows the various processors (master and parallel) to focus their processing power on manipulating data, rather than on collecting or distributing it.

The memory interface supports a wide range of external memory systems, composed of high-performance dynamic RAM, video RAM, and static RAM or a combination thereof. The transfer controller can generate all the necessary control signals for dynamic RAMs and supports the pipelined memory accesses of some of the newer memories.

Dual frame-buffer controllers

The MVP includes two identical and independent programmable display controllers. You can program each controller to generate the synchronization signals to control a display, or to lock to externally generated video signals for video capture. Each controller can automatically post requests to the transfer controller for video RAM serial transfer cycles.

Summary

We defined the multimedia video processor to be a fully programmable multiprocessor architecture for multimedia applications. On a single-chip architecture, we combined elements of RISC, floating point, advanced DSPs, graphics processors, display and acquisition control, RAM, and external memory control. The MVP brings a formidable amount of processing power to a variety of video, image, audio, and other signal processing applications. The architecture supports expanding the number of processors on a single chip to deliver in excess of 2 billion operations per second for multimedia applications. ❑

Acknowledgments

The authors thank co-architects Nick Ing-Simmons, Keith Balmer, Derek Roskell, and Ian Robertson; Chris Read, Jeremiah Golston, Syd Poland, and Eric Hansen, who helped refine the architecture; emulation architects Doug Deao and Gary Swoboda; Paul Fuqua for the architectural simulator; the program manager, Walt Bonneau; and the entire team in Houston, Dallas, and Bedford for their contributions.

References

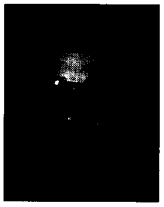
1. P.M. Dew, R.A. Earnshaw, and T.R. Heywood, eds., *Parallel Processing for Computer Vision and Display*, Addison-Wesley, Reading, Mass., 1989.
2. T. Fountain, *Processor Arrays: Architecture and Applications*, Academic Press, Orlando, Fla., 1987.
3. L. Snyder et al., eds., *Algorithmically Specialized Parallel Computers*, Academic Press, Orlando, Fla., 1985.
4. D. Tabak, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
5. R. Gonzales and P. Wintz, *Digital Image Processing*, Addison-Wesley, Reading, Mass., 1987.
6. B. Pratt, *Digital Image Processing*, 2nd edition, John Wiley and Sons, New York, 1992.
7. ISO/IEC, "Information Technology—Computer Graphics and Image Processing—Image Processing Interchange—Functional Specification," International Standards Organization, March 27, 1992.
8. D.H. Ingalls, "Smalltalk Graphics Kernel," *Byte*, Vol. 6, No 8, Aug. 1981, pp. 168-194.
9. J.D. Foley et al., *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley, Reading, Mass., 1990, pp. 806-807.
10. R.J. Gove, "Architectures for Single-Chip Image Computing," *SPIE Conf. Image Processing and Interchange*, San Jose, Calif., Feb. 12, 1992.
11. M. Asal et al., "Texas Instruments 34010 Graphics System Processor," *IEEE CG&A*, Oct. 1986, Vol. 6, No. 10, pp. 24-39.
12. R. Pinkham, M. Novak, and K. Guttag, "Video RAM Excels at Fast Graphics," *Electronic Design*, Vol. 31, No. 17, Aug. 18, 1983, pp. 161-182.
13. LSI-11, PDP11/03 processor handbook, Digital Equipment Corp., 1975.
14. K.M. Guttag, J.R. Van Aken, and M. Asal, "Requirements of a VLSI Graphics Processor," *IEEE CG&A*, Vol. 6, No. 1, Jan. 1986, pp. 32-47.

Karl Guttag is a TI Fellow with Texas Instruments. Since 1982, he has been responsible for graphics strategy at TI. He defined the 34010 and 34020 graphics processors, the first commercial video RAM, and various graphics peripherals. His earlier efforts included design work on the 9995 and 99000 16-bit microprocessors and the 9918 family of consumer video chips (the orginal "Sprite" chips). Currently, he is involved in defining advanced architectures for imaging and graphics, including processors, memories, and peripherals.

Guttag received his BSEE from Bradley University and his MSEE from the University of Michigan. He was elected to the TI technical ladder position of fellow in 1988. He is a member of IEEE and ACM.

Robert J. Gove is a senior member of technical staff and architecture manager in the Digital Imaging Venture Project at Texas Instruments. He has worked in research and development of imaging technologies, including infrared sensing, CCD imaging, image processing, computer vision, parallel processing, and HDTV. Gove was lead imaging architect in the development of the image processor described in this article. His interests include algorithm and architecture development for multimedia applications.

Gove received a BSEE from the University of Washington and both an MSEE and PhDEE from Southern Methodist University. He is a member of IEEE and SPIE.



Jerry R. Van Aken is a senior member, technical staff in the application-specific processors division of Texas Instruments. During his 12 years at TI he has been involved in the definition, simulation, and logic design of VLSI microprocessor system components, and has developed graphics libraries and real-time executive software.

Van Aken received his BSEE in 1974, his MSEE in 1975, and his PhD in electrical engineering in 1979 from the University of Washington in Seattle. He is a member of IEEE and ACM.

Contact the authors at Texas Instruments, PO Box 1443, MS 712, Houston, TX 77251.