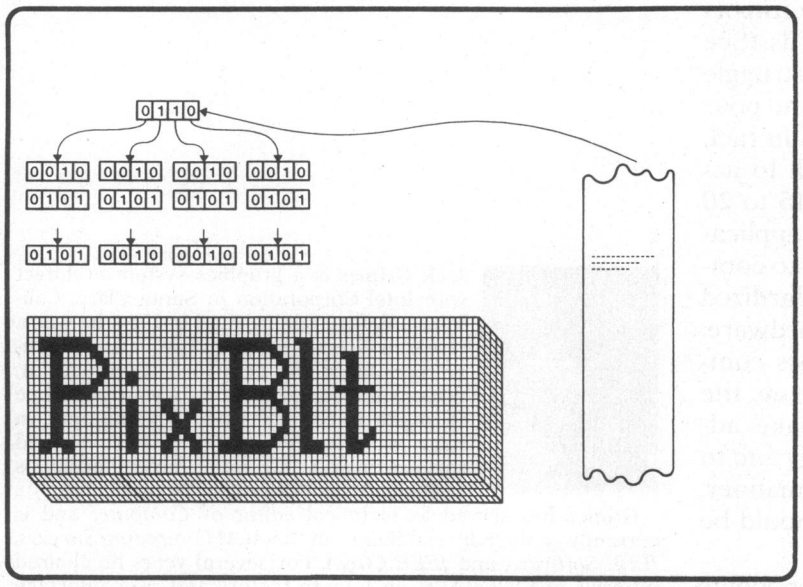
The Texas Instruments 34010 Graphics System Processor

Mike Asal, Graham Short, Tom Preston,
Richard Simpson, Derek Roskell, and Karl Guttag
Texas Instruments



The 34010 Graphics System Processor is a 32-bit graphics microprocessor capable of executing high-level languages. It combines a full general-purpose instruction set with a powerful set of graphics instructions that includes arithmetic as well as Boolean pixblts (pixel block transfers). Because it is completely programmable, the 34010 can be used in many different graphics and nongraphics applications. It was designed to support a wide range of display resolutions and pixel sizes, as well as applications such as page (laser) printers, ink jet printers, data compression, and facsimile transmission.

The 34010 includes such system features as an on-board instruction cache, full interrupt capability, wait and hold functions, and display timing control, as well as test and emulation support. Unique among today's microprocessors, the 34010 addresses all memory down to the bit level with variably sized fields rather than the common byte or word addressing. For example, the 34010 can push a 5-bit quantity onto a stack. This field-processing capability is an integral part of the basic architecture.

The developments leading to the 34010 began at Texas Instruments four years ago. Bitmapped graphics was just starting to find wide use in high-end systems where color was important, and the dominance of vector stroke/scanned high-end graphics systems was diminishing. Terminal and personal computer text displays were almost exclusively handled by hardwired text controllers that generated blocked text typically of 80 columns by 25 rows.

Groups such as those at Xerox PARC[1] were demonstrating that bitmapped graphics could provide better human interfaces and text capabilities as well as more advanced graphics. The improved capability and falling cost of dynamic RAMs indicated that bitmapped graphics would soon be widely used in almost all display systems.

During this period the first VLSI devices aimed at bitmapped graphics were introduced.[2] These chips provided hardwired implementations of a few graphics primitives such as one-pixel-wide line and

0272-1716/86/1000-0024$01.00 © 1986 IEEE

circle drawings, but text was generated by a hardwired block text display mode that was not compatible with the display of bitmapped graphics.

In defining the 34010, Texas Instruments decided that to simply extend the number of hardwired algorithms would be a fundamental mistake for several reasons:

1. The number of functions would be fixed. Not only would hardwiring support relatively few algorithms, but new or different algorithms could not be supported.
2. Even the most basic functions could require many attributes such as line style, width, color, and endpoint shape. Hardwiring would mean selecting the attributes to be supported and those to be left out.
3. Customers' experience in quality graphics absolutely required very fine control over the algorithms. Conceptually, drawing to a bitmap means selecting the nearest "integer" pixel, and thus some rounding error shows up as "jaggies" or aliasing. A quality graphics package might need access over any error terms or other parameters.
4. The format of display lists or commands varies according to user requirements. For example, simple block text might require a simple format, while variably sized text requires more complex display lists. The only way to support any format is to have a programmable processor interpret the commands.

The 34010 designers wanted to remove all arbitrary barriers imposed by hardwired controllers on both text and graphics. With the 34010, graphics algorithms could be added as required by a particular application or graphics standard. General-purpose instructions would be used to interpret display lists, with special graphics instructions used for fast pixel manipulations. A complete general-purpose instruction set that could support high-level-language programming (such as C) was blended with very powerful graphics instructions such as the pixblts.[3]

The 34010 was designed to give this flexibility without sacrificing speed in comparison with the hardwired controllers. Hardware such as an instruction cache, a large register file, a barrel shifter, a field mask and merge, and an independent memory controller was intended for the efficient implementation of any function.

The need for a video RAM

While the increased density and falling cost of dynamic RAM have triggered the change to bitmapped text and graphics, the increasing depth of the RAMs creates a fundamental problem. Most display devices must be constantly refreshed, which requires the entire displayable contents of memory to be read out at a periodic rate. Competing with these display refresh reads, a processor must access the memory to update the display's contents. The denser RAM devices, even with faster access modes, were making it impractical or even impossible to read their contents fast enough to support display refresh, let alone allow a processor accesses for updating.

At the beginning of the 34010 program, the designers recognized this problem and worked with the memory group within Texas Instruments to help define the industry's first multiport video RAM. The video RAM[4,5] incorporates a memory array with a large parallel-load and serial-output shift register function. The memory array can transfer a large number of bits (256 in the original version) from the memory array to the shift register in a single memory cycle time. The shift register supports the display refresh function while leaving the random access array free (except for an occasional memory-to-shift-register transfer) for updating.

The 34010 execution model

The 34010 has a 32-bit internal architecture. The CPU is made up of a 256-byte instruction cache; control ROM (CROM) and control logic; and the data path, which includes two 32-bit ALUs, barrel shifter, mask-merge logic, 31 32-bit user registers, left-most-one detection hardware, and window comparators. In addition to the CPU, the 34010 has CRT control, DRAM interface, a separate host processor interface, and an independent memory processor that pipelines accesses to the memory while arbitrating between the various sources generating requests—all integrated on a single chip.

The 34010 combines some of the best attributes of the so-called RISC (reduced instruction set computer) and CISC (complex instruction set computer) approaches into a single architecture. Each approach has its merits,[1] and the design goal with the 34010 was to properly blend these merits. Consistent with the Berkeley RISC[6] concepts, the 34010 has a large register file, simple direct-instruction decoding, low instruction pipelining for fast jumps, move-to-register instructions that zero-extend or (optionally) sign-extend external operands of different sizes to the full register size of 32-bits, and fast register-to-register operations. However, unlike the Berkeley RISC machine and similar to CISC machines, the 34010 allows for multiple-cycle instructions such as multiplies, divides, and the very complex pixblt instructions; pipelined writing to

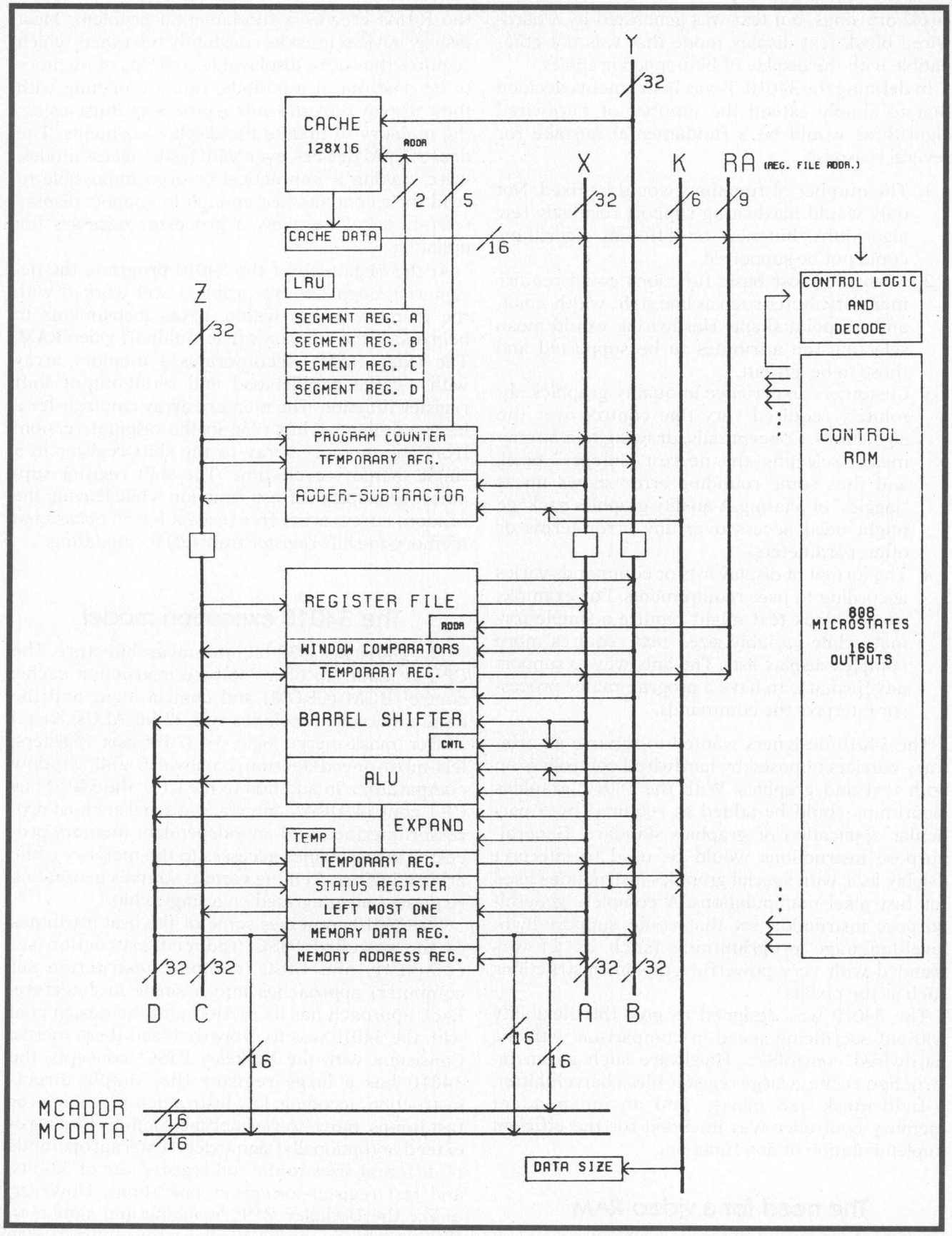


Figure 1. Block diagram of the CPU.

memory; memory-to-memory move instructions; and a wider selection of addressing modes. Additionally, the 34010's instruction cache boosts performance without requiring very fast external memory.

Every memory-to-register, register-to-memory, or memory-to-memory operation is in effect a field extract or field insert. For example, a memory-to-register move operation involves reading a field at a given bit address in memory and either sign-extends or zero-extends the number to 32 bits before it goes into a register. Memory-to-memory move instructions involve reading (extracting) a field at one bit address and storing (inserting) the field at a second bit address. These operations can involve reading, barrel shifting, merging data, and performing read-modify-write operations.

The 31 registers in the register file are organized as two register files, the A and B files, which share the stack pointer. Effectively, the stack pointer can be thought of as both A15 and B15. All of the registers can be used as address or data registers. Of these, only the stack-pointer register is fully dedicated in that it is assumed to point to the stack on which context information is saved in the event of interrupts or subroutine calls. The more complicated graphics functions use some or all of the B file registers as default parameters. When these graphics functions are not being used, these registers can serve any purpose. The A file registers have no predefined function in any instruction and are therefore totally at the user's disposal.

The single-register operand instructions can use any of the 31 registers. With two-register operand instructions, all 31 registers are available as the source operand register, but the destination register must be from the same register file (either A or B) as the source. The move register to register is an exception in that it can be used to move an A file register to a B file register or vice versa. Also, as mentioned earlier, the stack pointer is in both files, so that data from either file can be moved to or from the stack.

In typical applications the large register file can greatly improve performance, as all frequently used variables can be kept in the file for fast access and manipulation. An advantage with the 34010's register file is the ability to support the mixture of high-level-language control and assembly-coded, time-critical functions in separate register files and thus accelerate context switches. The first C compiler used only 13 of the A file registers (more than some other 32-bit CPUs *have*), leaving the B file totally free for assembly code. These registers will often be used by graphics instructions, which can involve many parameters. For example, the complex pixblt uses 15 registers to specify all the possible parameters and to store intermediate variables in the case of an interrupt.

A block diagram of the CPU and its major buses is shown in Figure 1. Three buses run the length of the data path; each can be split into two buses, allowing the data path to operate simultaneously as two independent units. For example, the A-X bus can be split into the A bus and the X bus.

Much of the 34010's power comes from the microcode contained in the CROM, which controls all of the hardware in the CPU. There are 808 microstates and 166 outputs, and nearly half of this very large CROM is devoted to the implementation of the pixblt instructions.

The 256 bytes in the instruction cache are organized as four segments of 32 16-bit words each. Each segment has a corresponding register which contains the base address of the section of memory from which the segment can be loaded. Thus the 34010 can execute code from the cache, which is located in four completely separate areas of memory. The four segments of memory can also be contiguous, allowing fairly lengthy loops to reside entirely in cache. If execution begins in a new area of memory, the new segment replaces the least recently used segment.

These replacements are controlled by the LRU register, which retains a record of the segment least recently used. Segments are further divided into eight subsegments, each with an associated "present flag" to indicate whether the subsegment has actually been loaded. Segments are loaded one subsegment at a time. Thus a segment would be brought into a cache by loading and executing four words, bringing in the next four, and so on.

In general, registers are read out onto buses (A,B,K) at the start of a machine cycle. These buses provide the inputs to the functional units such as the ALU or barrel shifter. Evaluation takes place during mid-cycle. Finally the results of the various operations are loaded onto output buses (C,D), which are then loaded into the appropriate registers.

The next microstate to be executed is determined in parallel with the activity in the data path. With a single-cycle instruction, the CROM address of the next microstate is obtained from the opcode word, which is always read from the cache. If the word indicated by the PC is not currently in cache, then that word, along with the other three words in the subsegment, is loaded. In the case of a multiple-cycle instruction, the next CROM address is contained within the control word of the present microstate. This address is a base address which can be conditioned to allow microbranching to one of several microstates.

To illustrate the internal workings of the CPU, we will take the instruction MOVE RS,*RD+,0 as an

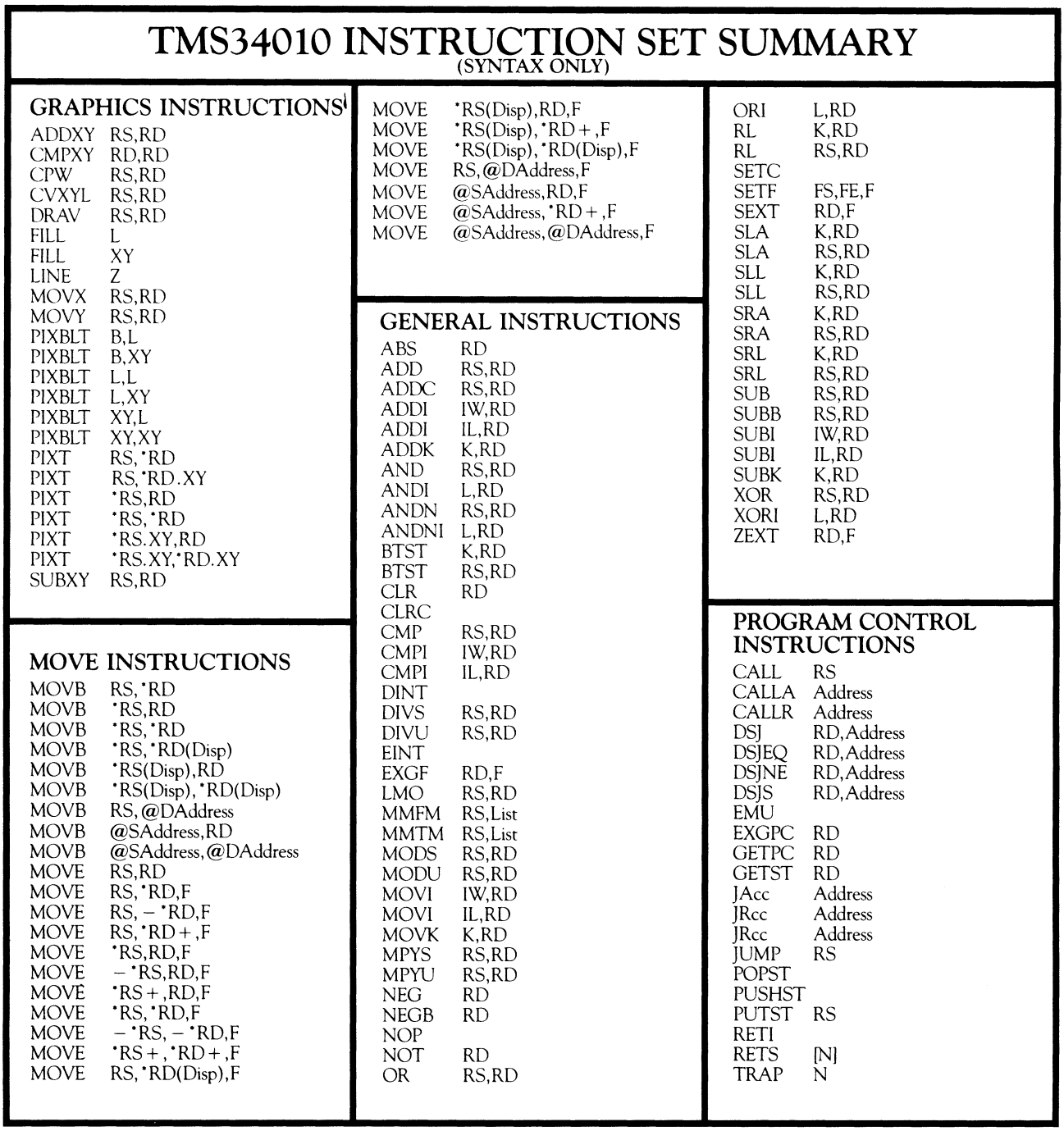


Figure 2. TMS34010 instruction set summary.

example. The syntax for the instruction is as follows: The field to be moved is in the source register from the register file, which is RS. The destination register, RD, is used as a pointer to a memory location and is incremented by the field size after being loaded into the memory address register. The field size is given by either the field size 0 or field size 1 quantity, both of which are contained in the status register. In this case the trailing 0 indicates that field size 0 is to be used. Assuming the instruction is in the cache, only one machine cycle is required for the execution of this instruction. The CPU can execute further instructions from the cache while the memory controller completes the write to memory.

The address in RD is gated onto the A bus, which

is then gated to the memory controller address (MCADDR) bus, if the memory controller is free to start the memory cycle immediately. The A bus is simultaneously loaded into the MA register, which will otherwise provide the address when the memory controller becomes available in a subsequent cycle. Field size 0 is gated from the ST register to the K bus. This value is loaded into the data size register (DS) used by the memory controller to determine the number of bits to be inserted into memory. The A and K buses also provide the inputs to the ALU, which performs an add operation to increment the pointer by the field size.

While these operations are taking place, the data in RS is gated onto the B bus, which is input to the barrel shifter. The four LS bits of the memory address on the A bus determine the alignment of the data within a 16-bit memory word, and hence are used by the barrel shifter as the rotate amount.

At the end of the evaluation phase, the output of the adder-subtractor is gated onto the Z bus, which then loads the PC. The ALU output goes onto the C bus, which is written into RD. Lastly, the BS output gates onto the D bus. Like the address on the A bus, the data on the D bus can be used immediately if the memory cycle has been initiated in that same machine cycle. The MD register is loaded from the D bus and provides the memory controller access to the data in subsequent cycles.

General-purpose instructions

The general-purpose instructions of the 34010 are designed to support a complete programming environment, including the use of high-level languages. The instructions available with the 34010 are listed in Figure 2.

As mentioned earlier, the 34010 supports bit addressing and fields rather than the more common byte addressing. Consequently, all memory operations involve field extraction and/or insertion. The data quantities moved are specified by one of two field sizes, which are programmable and contained in the status register. The field size can be from 1 to 32 bits in length and can begin on arbitrary bit boundaries within a memory word.

When data moves from memory into a register, the field is right justified and either sign- or zero-extended to 32 bits. Register-to-register Boolean, arithmetic, and shift instructions then use the 32-bit data paths to perform 32-bit operations in a single CPU cycle.

The addressing modes for the MOVE instructions reflect the register-based nature of the machine. Register-to-memory, memory-to-register, and memory-to-memory moves are supported for the five types of addressing modes. These are register indirect, register indirect predecrement (by the field size), register indirect postincrement (also by the field size), register indirect with displacement, and absolute addressing. The 34010's large register file and fast register-to-register instructions make it easy to synthesize more complicated addressing modes.

The integer arithmetic instructions include addition, subtraction, add with carry, subtract with borrow, and signed and unsigned multiplication and division. Most simple integer operations and all eight Boolean operations execute at six million per second out of the instruction cache. The signed and unsigned multiplications use 2-bits-at-a-time hardware to give a 64-bit product in roughly 3 microseconds.

The barrel shifter and sign control logic are used to perform rotate left/right, shift left logical, shift right arithmetic, and shift right logical by any amount (specified either in a register or as a constant) from 1 to 32 bits in a single CPU cycle.

The program flow control instructions give the user conditional jumps, subroutine calls and returns, decrement-skip-jump instructions for looping, program counter and register exchange, and 32 software traps. The 34010 is designed to execute conditional jumps quickly, requiring only one cycle if the jump is not taken and two if the jump is taken.

Instructions are also included for status register modification, including interrupt enabling/disabling and setting of the field size and field extension control.

Graphics instructions

In addition to the conventional linear addressing, the 34010 supports an optional *x-y* addressing function for graphics instructions. Most of these instructions provide a choice between *x-y* or linear addressing. For example, the pixblt instruction allows the source and destination array pointers to be in either format. Consequently there are four different versions of the instruction.

x-y addressing

With *x-y* addressing, a single 32-bit value is treated as separate 16-bit *x* and *y* halves. The two 16-bit values are converted automatically—within the instructions that use *x-y* addressing—to a linear address needed by memories. The generation of the linear address is a function of the *x* and *y* values, the *x-y* array's "pitch" (the linear address difference between two vertically adjacent pixels), and a 32-bit offset that allows the *x-y* address to be relative to any point in memory. The conversion process is shown in Figure 3. The clipping of figures and

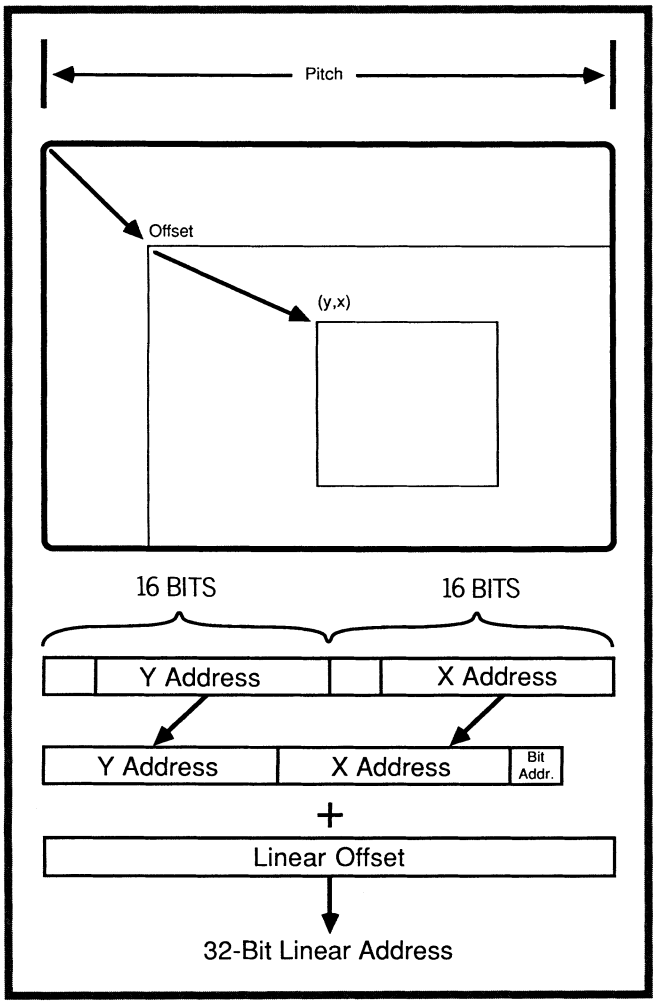


Figure 3. Conversion from *x-y* address to linear address.

arrays to fit within a rectangular region is also supported by *x-y* addressing.

Other instructions for *x-y* address manipulation perform *x-y* addition, subtraction, and comparison. Moves between registers of *x* or *y* halves are also provided.

Pixblt

The pixel block transfer instructions give the user a powerful tool for manipulating two-dimensional arrays of pixels. Pixblt supports the combinations of two rectangular arrays of pixels using any of the 22 Boolean and arithmetic pixel processing operations, plus transparency, plane masking, and window clipping options. In addition to simple filling, the FILL instruction supports a single value operating on a rectangular array of pixels with any of the pixel processing options.

The basic pixblt instruction combines the pixels in the source array with those in the specified destination array. The relative word alignments of the source and destination arrays can be totally arbitrary. Edge conditions are taken care of, and pixels are moved a memory word at a time except at the edges, where read-modify-writes may have to be performed. An added complication for the two-array transfer is the bit alignment of the source and destination arrays, which may be aligned differently with respect to memory word boundaries. Thus the source data must be shifted to align to the destination location before it can be written. This instruction aligns the source and destination arrays automatically and with very little, if any, decrease in performance.

An important feature supported in pixblts is the ability to have different values for the source and destination pitches. For example, an array only 16 bits wide can be stored off screen packed as 16-bit words. When transferred to the bitmap, the source pitch is 16 bits, while the destination pitch is the number of bits in one line of display memory.

Window clipping

A window is specified as a rectangular region that is optionally protected from being overwritten. Probably the most common use is the implementation of a viewport concept. Only the pixels that lie within the window are written to the bitmap.

Four different window clipping options are supported. The "no clipping" option ignores windowing boundaries and does not protect any pixels. The "clip-to-fit" option prohibits pixels outside the window from being written. The "interrupt on window violation" immediately aborts the operation if any point in an operation will lie outside the window. The window violation interrupt is then set and will be taken immediately if it is enabled. The "pick" option determines whether any pixel lies within the window. With the pick option, all pixel drawing is inhibited and the window violation interrupt is issued whenever a pixel would be drawn within the pick region.

For the pixblt instructions with the clip-to-fit option, the destination array is preclipped to the window before any data is transferred. This preclipping avoids the potential waste of large amounts of time computing addresses for pixel transfers that will end up being clipped. Adjustment of the source array is based on adjustment of the destination array.

Pixel processing

Every pixel transferred with the graphics instructions of the 34010 can use pixel processing operations, which combine each source pixel with

Table 1. The 22 pixel-processing options.

PP	Operation				Description
00000			S	→D	Replace destination with source
00001	S	AND	D	→D	AND source with destination
00010	S	AND	D–	→D	AND source with NOT destination
00011			O	→D	Replace destination with zeroes
00100	S	OR	D–	→D	OR source with NOT destination
00101	S	XNOR	D	→D	XNOR source with destination
00110			D–	→D	Negate destination
00111	S	NOR	D	→D	NOR source with destination
01000	S	OR	D	→D	OR source with destination
01001			D	→D	No change in destination*
01010	S	XOR	D	→D	XOR source with destination
01011	S–	AND	D	→D	AND NOT source with destination
01100			1	→D	Replace destination with ones
01101	S–	OR	D	→D	OR NOT source with destination
01110	S	NAND	D	→D	NAND source with destination
01111			S–	→D	Replace destination with NOT source

*Although the destination array is not changed by this operation, memory cycles still occur.

The following six pixel-processing codes perform arithmetic operations on pixels of size 4, 8, and 16 bits.

PP	Operation		Description
10000	D + S	→D	Add source to destination
10001	ADDS(D,S)	→D	Add S to D with saturation
10010	D – S	→D	Subtract source from destination
10011	SUBS(D,S)	→D	Subtract S from D with saturation
10100	MAX(D,S)	→D	Maximum of source and destination
10101	MIN(D,S)	→D	Minimum of source and destination

the corresponding destination pixel before overwriting the destination pixel with the result. Both logical and arithmetic processing are supported. The 16 standard Boolean operations are familiar from their use in bitblts.[7]

Less familiar but very important for pixblts are the arithmetic operations that operate on entire pixels rather than on the individual bits within pixels. For multiple-bits-per-pixel applications, these operations are not only desirable but necessary.[3] Six of these options are supported. The entire list of 22 pixel processing options is shown in Table 1.

Plane mask

The plane mask enables the user to inhibit writes to some or all of the bits within a pixel. For example, when a pixel contains information for a group of planes, the bits associated with a given plane or planes can be protected. Thus text can be modified in one plane while the graphics information in other planes remains unaffected. This mask is applied to pixel data as it is read into the chip as well as to the data to be written externally.

Transparency

Transparency detection can be selected to treat zero-value pixels as transparent. This detection is applied to the result of pixel processing and plane masking. With transparency enabled, a pixel of zero value does not overwrite the destination pixel, leaving the background unchanged. In this way, only the part of a rectangular array that makes up the desired shape (such as a text character) is actually written. The overlaying of text on graphics then becomes a very simple task.

Direction of pixblt

For the basic pixblt instruction, any of the four corners can be selected as the starting corner. This in turn determines the direction in which the transfer takes place. Thus the overlapping of source and destination arrays (which commonly occurs with scrolling) can be handled in a manner that prevents the overwriting of subsequent source data by writes to the current destination location.

Expand

The pixblt with expand option is very useful for transforming a 1-bit-per-pixel shape such as a text font to a color image in the bitmap. Each pixel is expanded from 1 bit to either a "one color" or "zero color," which can be 1, 2, 4, 8, or 16 bits in length. This process is illustrated in Figure 4. After expansion, any of the arithmetic and Boolean operations can be applied, as well as such options as transparency and plane masking.

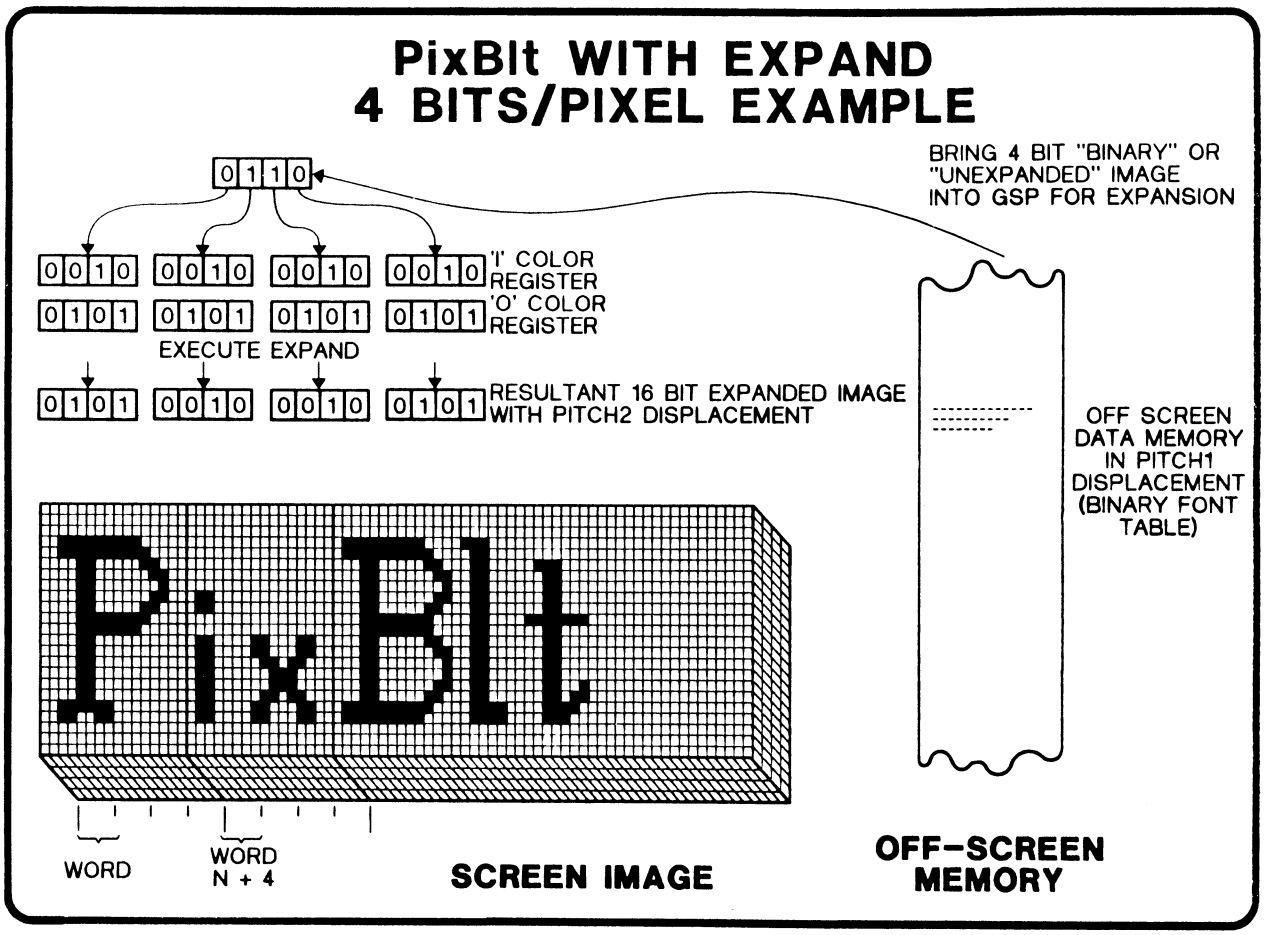


Figure 4. Expansion of the 1-bit-per-pixel representation to a color image.

Single-pixel instructions

There are several instructions for manipulating individual pixels. A set of single-pixel transfer (pixt) instructions supports both *x-y* and linear addressing for either the source or destination pointers. Windowing, pixel processing options, transparency, and plane masking are all supported for this instruction.

The Draw and Advance (DRAV) provides the basis for incrementally drawing figures such as lines and circles. This instruction combines a single-pixel draw (with pixel processing operations) with an addition to both the x and y halves of an *x-y* address pointer.

A simple assembly language implementation of the inner loop of Bresenham's line algorithm[8] using DRAV is shown in Figure 5. The code for the inner loop requires just seven 34010 instructions, but for any given pixel to be drawn, the conditional branching causes only four of the instructions to be executed.

To draw a line from $(y0,x0)$ to $(y1,x1)$, where $a = x1 - x0$, $b = y1 - y0$, and $a \geq b$, the registers used are set up according to Figure 5. The microcoded LINE instruction replaces the code shown in the figure, nearly doubling the performance. This instruction still supports very flexible endpoint and jagged-line control by letting the programmer control the decision variable (or error term). The action taken when the decision variable is zero is also programmable. Windowing, pixel processing, transparency, and plane masking are all available, as with the other pixel transfer instructions.

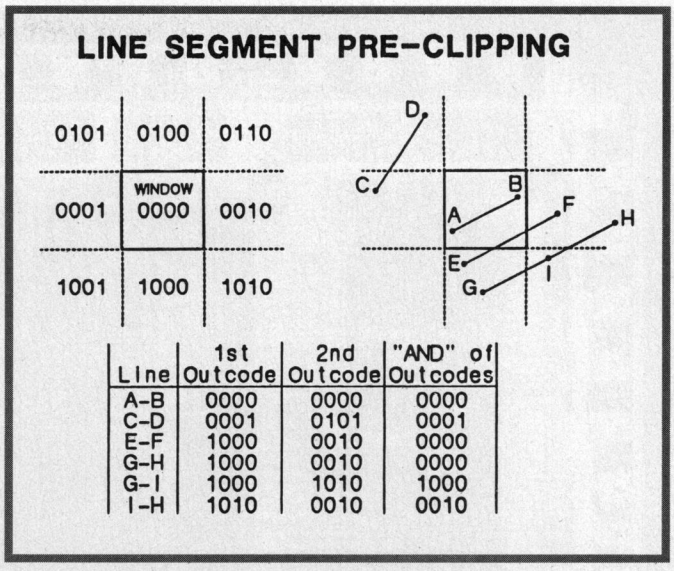
Compare Point to Window

One additional instruction extremely useful in the software implementation of many graphics algorithms is Compare Point to Window (CPW). Executing in a single cycle, CPW compares the x and y values of a given point to the corresponding values of both the window start and window end registers, generating a 4-bit code. This code indicates in which of the nine regions relative to the window the point lies. This concept, illustrated in Figure 6, is best

```
A0 = Destination pointer      (y0,x0)
A1 = Increment1               (1,1)
A2 = Increment2               (0,1)
A3 = Pixel count              a - 1
A4 = Decision variable        d
A5 = Temporary1               2b - a
A6 = Temporary2               2b
B9 = Pixel color1

LOOP:
   JRN DNEG       ; Jump to DNEG if the decision
                  ;    variable d is negative
   DRAV A1,A0     ; Write pixel to location specified
                  ;    by A0 and add the increment
                  ;    value in A1
   ADD A5,A4      ; Add the temporary value in A5
                  ;    to d
   DSJS A3,LOOP   ; Decrement the count and loop
                  ;    until zero
   RETS           ; Finished
DNEG:
   DRAV A2,A0     ; Write pixel to location specified
                  ;    by A0 and add the increment
                  ;    value in A2
   ADD A6,A4      ; Add the temporary value in A6
                  ;    to d
   DSJS A3,LOOP   ; Decrement the count and loop
                  ;    until zero
   RETS           ; Finished
```

Figure 5. Implementation of Bresenham's line algorithm using DRAV.



Line	1st Outcode	2nd Outcode	"AND" of Outcodes
A-B	0000	0000	0000
C-D	0001	0101	0001
E-F	1000	0010	0000
G-H	1000	0010	0000
G-I	1000	1010	1000
I-H	1010	0010	0010

Figure 6. The 4-bit code produced by the CPW instruction is very useful in clipping algorithms.

known for its use in the Cohen-Southerland line-clipping algorithm.[9]

The 34010 design architecture

The 34010 currently is driven by a 50-MHz input clock whose frequency is divided by 8 internally to give a machine state rate that allows execution of more than 6 million instructions per second from the cache. It uses 1.8-μm CMOS technology to integrate 200,000 transistors on the chip with a typical power dissipation of 0.5 watt. The 34010 is housed in a 68-pin plastic package. A die photo is shown in Figure 7.

Local memory control

The 34010 provides a comprehensive set of signals for external memory control. The high frequency of the input clock allows generation of signal timing with sufficient resolution to efficiently interface to DRAMs. Eight control signals and a 16-bit address/data bus allow most combinations of external memory devices to use minimal additional logic. Two signals, DEN and DDOUT, can be used to control bidirectional transceivers where buffering is required. LAL is provided to latch the multiplexed column address so that the triple-multiplexed bus is free to function as a data bus. RAS, CAS, and W are standard DRAM control signals which, together with the 16-bit multiplexed address/data bus, provide a direct DRAM interface. The TR/QE signal can be used as a simple output enable for RAMs as well as the shift register transfer control for video RAMs. A "ready" function is also provided on another pin to cater to slow memories.

The memory controller is a separate autonomous controller interfacing to "local" memory systems. Its main function is to service requests from the CPU, the host interface, and the display refresh and DRAM refresh mechanisms. The memory controllers must be fairly intelligent to handle the bit-addressable, variable-size field requests from the 34010's CPU. For example, in a register-to-memory transfer, the CPU will define the field size (1-32 bits) to be moved and the bit location where the first bit of the field is to be placed. The memory controller must translate the CPU request into one or more memory cycles. Since the memory controller is autonomous, on write cycles the CPU does not have to wait for the write cycle(s) to complete; instead the CPU can execute code while the memory controller writes to memory.

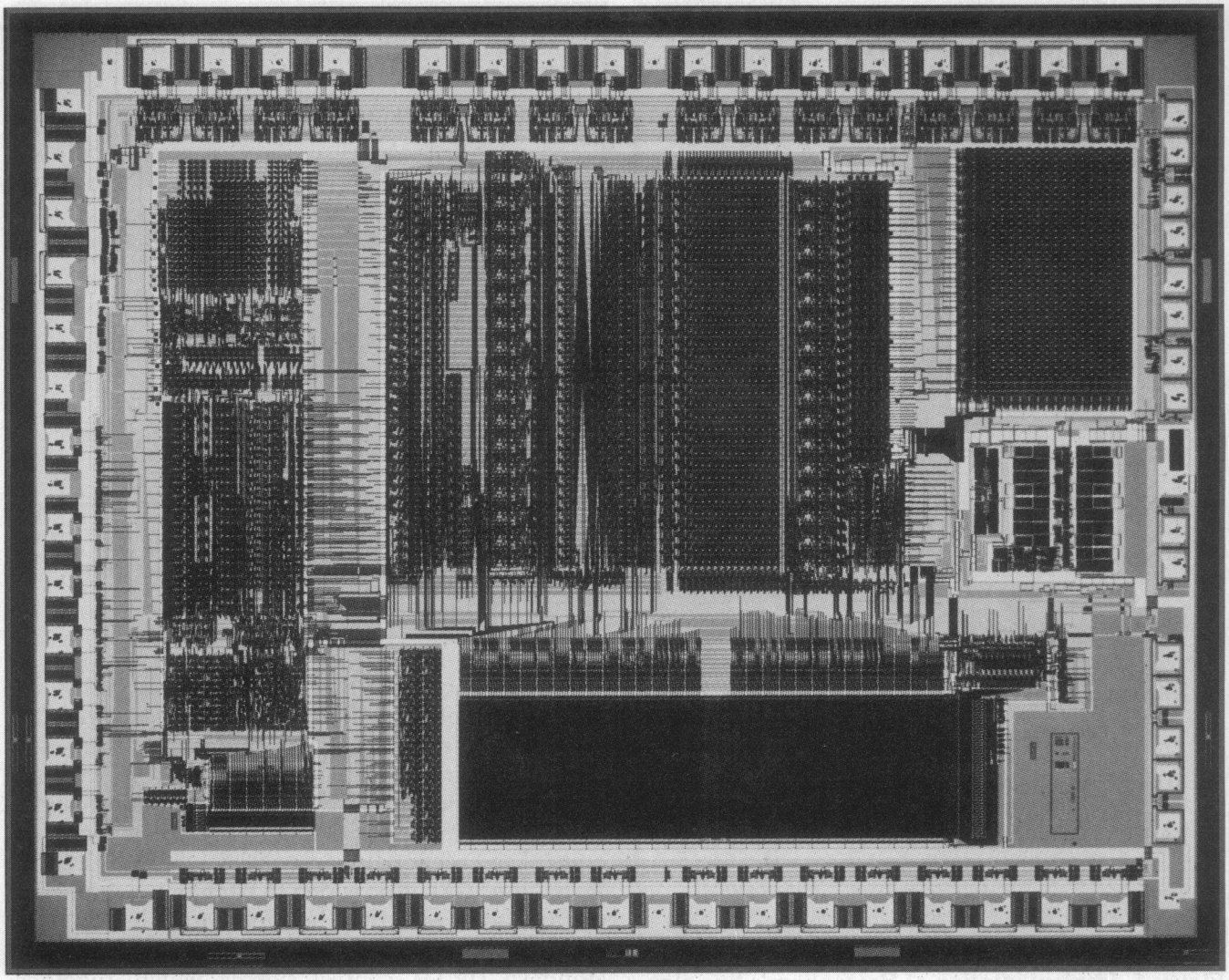


Figure 7. Die photo of the 34010 Graphics System Processor.

Video controller

The video control section of the 34010 provides the sync and blanking signals used in CRT control as well as direct support of video RAMS. The video clock input (VCLK) can be totally asynchronous with respect to the 34010's main CPU processor clock (INCLK), so that the processor and video timing can be independently optimized. Eight 16-bit timing control registers support generation of horizontal and vertical timing waveforms for essentially any display resolution. In internal sync mode the 34010 will generate the sync outputs from VCLK. In external mode the video controller will read either one or both of the sync signals as input(s) and synchronize screen refreshes accordingly. This permits graphics images created by the 34010 to be superimposed on images created externally.

In addition to sync and blank generation, the video controller performs the video-RAM-to-shift-register cycles needed to refresh the screen. Three I/O registers (DPYADR, DPYCTL, and DPYSTRT) allow screen refreshing to be configured to whatever memory configuration is adopted.

DRAM refresh

The 34010 has a separate counter that schedules DRAM refresh cycles. It can be configured to provide either RAS-only refresh with 8 bits of refresh address, or CAS-before-RAS refresh where no address is needed.

Host interface

The 34010 has been designed to maximize system performance and ease of use while minimizing system chip count. The external interface signals and their various groupings are shown in Figure 8.

The 34010 contains a host interface consisting of a 16-bit bidirectional data bus and nine control pins. The control pins are designed to support any of the commercially available microprocessor interfaces with little external logic, and the upper and lower bytes of the 16 data pins can be wired together to support 8-bit microprocessors. The interface can be configured to work either high-byte first or low-byte first (addressing the so-called Little Endian/Big Endian problem faced in processor-to-processor interfaces).

This interface supports a pipelined indirect pointer mechanism for having a host processor access the memory space of the 34010. It can support sequential reading or writing of up to 5M bytes per second. This is one of two ways that a host processor can access the 34010's memory. The other method uses a conventional hold interface, described below.

The host interface consists of a 32-bit address register and 16-bit data and control registers. The address pointer can point to external local memory or to the 28 memory-mapped I/O registers inside the 34010. The interface to the host can be totally asynchronous. The HRDY (host-ready) pin is used to signal a host in case overrunning the pipeline makes it necessary to stretch a cycle. The dedicated interrupt pin can be used by the 34010 to signal the host.

Also provided in the host interface are two 8-bit host control registers. The host control register enables the host to halt the 34010's CPU at the end of the current instruction. (The memory controller keeps running to give the host and refresh control access to the memory.) It also allows the host to generate a nonmaskable interrupt to grab control of the 34010 or generate a maskable interrupt with a 3-bit code for message passing.

Hold

A straightforward hold/hold-acknowledge system provides direct access to the 34010's local memory by another processor. Asserting hold will cause the local memory address/data and control pins to enter a high-impedance state so that external devices can access local RAM directly. The hold-acknowledge output tells the requesting processor that the bus is in a high-impedance state.

Emulation

The 34010 was designed to directly support hardware development systems by building emulation capability into the chip. The RUN/EMU pin, together with an emulation acknowledge output, controls the emulation mode. This allows a development system to dump and load the contents of the 34010 under the control of a code placed on the three pins LINT1, LINT2, and HOLD.

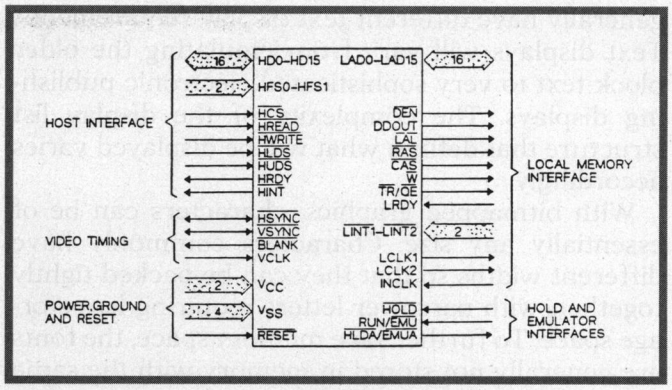


Figure 8. Pinout with interface groupings.

Testability

The increasing demands for complex devices fuel an increasing demand for effective and thorough testing. A special effort was made to incorporate a comprehensive test mechanism. Testability strategies for VLSI circuits aim to increase the controllability and observability of internal storage nodes (or bits). The 34010 was designed so that on any CPU cycle the machine can be stopped, and then every storage node on the device can be dumped and/or loaded. This permits the necessary observation into the machine as well as control in setting up the state of the machine.

Power and interrupts

A total of four separate ground lines and two separate power lines provide for fast-switching outputs with low noise. The reset input performs a hardwired reset function, clearing various registers and performing a trap 0 on release. Two pins in this group are provided for external interrupts from either a host or coprocessor.

Applying the 34010

In defining the 34010, the designers learned about the breadth of graphics functions that should be supported in a complete graphics system. We will offer just a few examples of the types of functions a 34010 can be called on to perform and try to give a little insight into some of the 34010's features that are particularly useful in each application.

Text

Text has been and will remain the most important graphical element for most systems. Applications

generally have different text display requirements. Text displays will vary from emulating the older block text to very sophisticated electronic publishing displays. The complexity of the display list structure that defines what will be displayed varies accordingly.

With bitmapped graphics, characters can be of essentially any size. Characters commonly have different widths so that they can be packed tightly together, with narrower letters requiring less storage space. To further save memory space, the fonts are generally not stored in memory with the same "pitch" as the display memory.

Geometric (or equation-defined) text descriptors give even further flexibility. Geometric text can be scaled or rotated to provide variations that would require an infinite amount of storage if different sizes and orientations for each character had to be stored individually. Theoretically, geometric text could be slow to compute the shape of each letter, but in most text applications the overhead can be very small. Typically the text will consist of only a few sizes, styles, and orientations, and thus once a letter has been translated from equation to a bitmapped image, it can be used repeatedly via block transferring. The 34010 could generate the fonts from the equations and, of course, block transfer the characters onto the screen memory.

The 34010's general-purpose instructions can be used to interpret any display list structure, be it simple or complex. The result of the display list interpretation will be a pointer to the font (or shape) of each letter and a pointer to where the character should appear on the display. A pixblt instruction (often the pixblt with color expand) will then be used to copy the font image onto the display memory.

The pixblt itself can be quite complicated. The binary-to-color expand option supports efficient storage of arbitrary-size fonts at 1 bit per pixel. Any of the Boolean or arithmetic operations can be called on for combining the text with the current image. Transparency detection can be used for handling overlapping color characters. The pixblts can automatically clip each character to fit within a rectangular window. The screen image might be x-y addressed with one memory pitch, while the character would be linearly addressed with a different memory pitch. Any plane or group of planes can be masked (or locked). And all of the above-mentioned operations can be performed by a single 34010 pixblt instruction.

Line, circle, ellipse, and general figure drawing

Most figure drawing with a 34010 will be performed using "incremental algorithms," which have been widely described.[10–12] The fast general-purpose instructions and special instructions that allow independent addition/subtraction of the x and y halves of a 32-bit register are used to generate essentially any function.

Incremental algorithms generally have one or more decision variables. Among other things, these variables keep track of the error term(s) associated with rounding to an integer pixel grid and maintain the mathematical accuracy as a figure is drawn. To maintain full accuracy for common figures generated on typical display resolutions requires up to 32 bits of precision. For example, long and thin ellipses can roll over 16 bits of error term and have been known to cause rather undesirable effects.

The 34010's 32-bit-wide by 31-register file is large enough to hold all the decision variables for any of the common figure-drawing algorithms. If a figure to be drawn starts at a noninteger coordinate (as can easily happen when a figure is clipped or results from floating-point math), the decision variable must be initialized, a task the arithmetic instructions of the 34010 can perform.

Simple forms of antialiasing (gray-scale smoothing) can be performed relatively quickly by using the decision variable to control the intensity of what is being written.[13] As these antialiased figures are drawn on top of each other, operations other than replace or Boolean functions must be used to prevent the faint fringes of one figure from obliterating the solid part of a previously drawn figure.[14] The MAX and MIN operators can provide a reasonable mechanism for drawing and combining certain classes of antialiased objects.

Filled objects

Just drawing one-pixel-wide outlines of objects is not enough for most of today's graphics applications; filling is often required. The two basic ways to generate a filled object are by direct filling of the object from the equations defining it, or by seed (flood) filling a region enclosed by some boundary.

Seed filling is sometimes useful in artistic applications such as MacPaint, where it is intuitively easy to use; one simply draws a border (or outline) and floods the region with a color and/or pattern, analogous to children's coloring books.[15] Within the class of seed filling there are several variations. The algorithm can vary because of how the border is "connected."[16] The border can be defined by pixels that are equal to a certain value, not equal to a certain value, inside or outside a range of values, or by many other functions. Combinations of general-purpose instructions, fill (with pixel processing) instructions, and pixblts have been used to implement seed-filling algorithms on the 34010.

While seed filling is useful in some interactive applications, it cannot be used for many applica-

tions, and direct equation-controlled filling is necessary. Seed filling is slower than direct filling, as pixels must be read and checks for the border made. For a seed fill of a general figure not seeded by human intervention, it can be difficult to determine an "interior pixel" so that the drawing can begin. Seed filling cannot, for example, handle the drawing of one object on top of another because pixels in the object below might be interpreted as the border.

There are many different approaches to direct-equation filling, and often there are very important trade-offs between speed and the complexity of the object that the algorithm can render. The 34010 can, of course, be programmed to handle any degree of complexity.

Convex objects such as polygons (Figure 9a), circles, ellipses, and pie wedges lend themselves to faster fill algorithms. The procedure for performing convex polygon filling using the 34010 is outlined below (some special cases and details are left out for simplicity):

1. Sort the endpoints of all the figures to be drawn by their y (vertical) values. This task uses the 34010's x-y compares, and its conditional jumps and moves.
2. The filling starts with the two lines that share a common point with the lowest y value. With the exception of the special case of one of the two lines' being horizontal, there will be a left line and a right line. The 34010's fast arithmetic, conditional jumps, and special ADDXY instructions will be used to compute each line incrementally. Each line algorithm is run until it steps vertically. Once both have stepped vertically one pixel, a horizontal line is drawn between the two computed endpoints (using either a FILL or pixblt instruction). This step repeats until one of the two lines is done.
3. The endpoints of the next line segment from the y sorting (which should share an endpoint with the line just finished) are then used to start a line that replaces the one that has been completed. This process continues until all line segments are used up.

The convex polygon fill demonstrates how some of the features of the 34010 can be used. The 34010's register file is large enough to contain all the variables for both lines as well as the parameters for the FILL instruction. The instruction cache is large enough to hold both of the line-draw algorithms as well as algorithms for fetching new lines. Holding all the parameters and instructions on chip permits fast execution of the convex fill algorithm even though it can be programmed to the user's exact requirements.

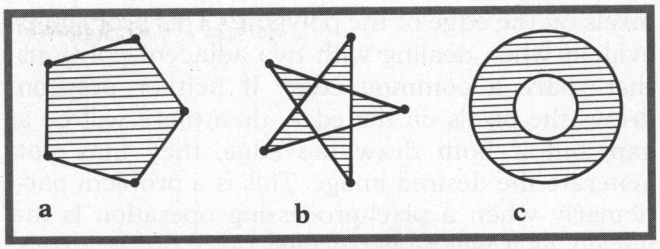


Figure 9. (a) Convex polygon; (b) self-intersecting polygon; (c) figure with a hole.

Filling more general figures (such as Figures 9b and 9c) may involve greater complexity. The five-pointed star of Figure 9b is a polygon that self-intersects. Figure 9c is a doughnut ring with a hole. In both cases, determining what is inside (filled) and what is outside (not filled) involves some ambiguity. Thus rules have to be adopted that define the inside and outside. With the parity rule, a pixel is defined to be inside the object if an odd number of boundary crossings is required to get to the given pixel. In the doughnut, for example, it takes two crossings to get to the hole, and thus the pixel is outside.

Perhaps the simplest way to implement arbitrary filled objects with the parity rule is to first draw the outlines defining the object at 1 bit per pixel into a scratchpad area. The scratchpad has a 1-bit-per-pixel mapping of the screen image[17] and is at least as large as the minimum enclosing rectangle of the image. When the outline is drawn, it is important to use the XOR operation; without it there is the possibility of "leaks" if the outline crosses. Once the outline is drawn, each horizontal line of the scratchpad space is scanned, counting the number of times a pixel defining the boundary is crossed. Between pixels where the number of crossings is odd, a fill (perhaps with a texture) is made to the corresponding part of the display memory at the full pixel depth.

With the 34010, the single CPU cycle left-most-one instruction can be very useful for scanning the horizontal line for the even or odd number of ones. The fact that the 34010 can flexibly handle different numbers of bits per pixel is useful in dealing with the 1-bit-per-pixel scratchpad and the screen's frame buffer, which has perhaps many bits per pixel. The ability to support different array pitches allows the 34010 to efficiently place the scratchpad area.

In dealing with filled objects (or almost any other graphics figure), there are many subtle and not-so-subtle variations that users may want to make; thus, the ability to use different algorithms is important. A classic problem is how to deal with the

pixels on the edge of the polygon.[18] This problem is evident when dealing with two adjacent polygons that share a common edge. If neither polygon draws the pixels on the edge, then there will be a gap; but if both draw the edge, they may not generate the desired image. This is a problem particularly when a pixel-processing operation is involved, as it will be performed twice on the pixels along the edge.

Image compression/expansion

Graphics images can require extremely large amounts of memory. A single 640 × 480 × 8 medium-resolution image requires about 300K bytes of memory, roughly the entire capacity of a typical PC's floppy disk, if stored uncompressed. Even with significant advances in storage technology, images will clearly have to be compressed for both storage and transmission.

Image compression is based on identifying some way in which the image contains redundant information and then using fewer bits to describe the redundant parts. For some applications, such as encoding a television picture or photograph, the goal may be to save an image that will appear good enough to the naked eye while allowing some sacrifice of the "digital fidelity" of the original image to improve the compression. In other applications it will be necessary to maintain an exactly reproducible version of the original image. Practical tradeoffs will also be made in the time required for compression versus the amount of compression an application may require.

Most algorithms assume a particular kind of redundancy that has been shown empirically to occur. Different images will compress better with different algorithms. More intelligent programs can adapt and select a compression algorithm that best compresses all, or given portions, of the image.

Take, for example, the CCITT group 3 and 4 compression standards,[19,20] which were defined to generate a reasonable compression for the type of images that commonly occur in facsimile transmission. The compression involves looking for alternating "runs" on each horizontal line of white and black ("0" and "1") dots and using variable-length codes. Shorter codes are used for run lengths that occur more frequently. Occasionally, the compression can be negative; that is, it may produce more bits than it saves. The standard has a provision to "adapt" and send a special code that indicates a string of uncompressed code.

In compressing a CCITT image, the 34010's single CPU cycle left-most-one (LMO) and shift instructions can be used to scan across bits, looking for the run lengths. (Using a NOT instruction before the LMO allows it to look for the leading "0.") Once the length of the run has been identified, a table lookup can be used to translate the run length into the variably sized code. Each table entry would have the bits corresponding to the code and the size of the code. The code size would be used as the field size for a 34010 MOVE instruction to move the code to the output stack. For decompression, a combination of table lookups and field moves can be used to extract and turn the codes into the number of bits to be drawn in a run. The run length is then used by the FILL instruction to determine how many bits are to be drawn.

Summary

The 34010 provides a unique combination of general programmability, graphics processing power, and system integration on a single integrated circuit. The design philosophy was to not restrict graphics to a few primitive functions, but rather to support whatever functions are required in any application. ■

References

1. D.H. Ingalls, "The Smalltalk Graphics Kernel," *Byte*, Vol. 6, No. 8, Aug. 1981, pp.128-194.
2. J.L. Wise and H. Szenjnwald, "Display Controller Simplifies Design of Sophisticated Graphics Terminals," *Electronics*, Vol. 54, No. 7, Apr. 7, 1981, pp.153-157.
3. K. Guttag, J. Vanaken, and M. Asal, "Requirements for a VLSI Graphics Processor," *IEEE CG&A*, Vol. 6, No. 1, Jan. 1986, pp.32-47.
4. Ray Pinkham, Mark Novak, and Karl Guttag, "Video RAM Excels at Fast Graphics," *Electronic Design*, Vol. 31, No. 17, Aug. 18, 1983, pp.161-182.
5. Mary C. Whitton, "Memory Design for Raster Graphics Displays," *IEEE CG&A*, Vol. 4, No. 3, Mar. 1984, pp.48-65.
6. Robert P. Colwell et al., "Computers, Complexity, and Controversy," *Computer*, Vol. 18, No. 9, Sept. 1985, pp.8-19.
7. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer*, Vol. 15, No. 9, Sept. 1982, pp.8-21.
8. J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems J.*, Vol. 4, No. 1, 1965, pp.25-30.
9. J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982, pp.144-148.
10. M.L.V. Pitteway, "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter," *Computer J.*, Vol. 10, No. 3, Nov. 1967, pp.282-289.
11. J.E. Bresenham, "A Linear Algorithm for Incremental Display of Digital Arcs," *Comm. ACM*, Vol. 20, No. 2, Feb. 1977, pp.100-106.

12. Jerry R. Van Aken, "An Efficient Ellipse-Drawing Algorithm," *IEEE CG&A*, Vol.4, No. 9, Sept. 1984, pp.24-35.
13. Akira Fujimoto and Kansei Iwata, "Jag-Free Images on Raster Displays," *IEEE CG&A*, Dec. 1983, pp.26-34.
14. A.R. Forrest, "Antialiasing in Practice," in *Fundamental Algorithms for Computer Graphics*, R.A. Earnshaw, ed., Springer-Verlag, New York, 1985, pp.113-134.
15. Henry Lieberman, "How To Color in a Coloring Book," *Computer Graphics* (Proc. SIGGRAPH 78), Vol. 12, No. 3, Aug. 1978, pp.111-116.
16. J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982, pp.446-451.
17. A.C. Gay, "Experience in Practical Implementations of Boundary-Defined Area Fill," in *Fundamental Algorithms for Computer Graphics*, R.A. Earnshaw, ed., Springer-Verlag, New York, 1985, pp.153-160.
18. Michael Dunlavey, "Efficient Polygon-Filling Algorithms for Raster Displays," *ACM Trans. Graphics*, Vol. 2, No. 4, Oct. 1983, pp.264-273.
19. "Standardization of Group 3 Facsimile Apparatus For Document Transmission," recommendation T.4, CCITT, 1984.
20. "Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus," recommendation T.6, CCITT, 1984.

Mike Asal is a design engineer in the Microprocessor and Microcomputer Products (MMP) Division of Texas Instruments in Houston, Texas. He is currently working on the definition and design of the next generation graphics microprocessor. Since joining TI in 1982, he has worked on the specification, internal architecture, and design of the TMS34010. His specific design responsibilities were CPU logic and microcode. His research interests include microprocessor architecture and the use of VLSI technology in computer graphics.

Asal received a BSEE and an MSEE from Bradley University in 1981 and 1982, respectively. He is a member of the IEEE.

Graham Short is a design engineer with Texas Instruments' MMP Division's Bedford, England, design center. He joined TI in 1980 and has worked on logic entry, design verification, and, more recently, on silicon test, debug, and characterization of the TMS34010. He worked for two years with the Design Automation Department in Bedford developing software for microcomputer boards, including a graphics board based on the TMS9995. After a brief assignment on robot controllers, he moved to MOS design to work on the TMS34010.

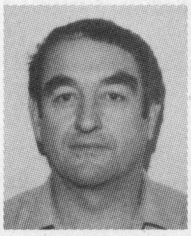
Short graduated from Kent University with a degree in math and computer science. He recently received an MSEE from the University of London.

Tom Preston is currently working for General Datacomm in Connecticut. Previously he spent almost 10 years with Texas Instruments in both the US and England. In his most recent assignment for TI, he was logic design manager on the TMS34010 from 1983 to 1986. Before that he worked on the TMS9995 and the development of VLSI memory components.

Preston received a BSc in physics from Birmingham University, England, and a master's degree from Southampton University, England. He was elected a senior member of the technical staff at TI in 1986. He holds several patents in the area of microprocessors, and he is a member of IEEE.

Richard Simpson is a group member of the technical staff at Texas Instruments' MMP Division's Bedford, England, design center. He was in charge of all circuit design for the TMS34010. Simpson joined the Bedford MOS Design Centre in August 1977 as a MOS design engineer and has worked on the design and testing of several VLSI MOS devices: the TMS9914 GPIB, the TMS9995 16-bit microprocessor, the TMS32010 digital signal processor, as well as the TMS34010. He is presently design manager for an upgraded version of the TMS34010.

Simpson received his BSc in electrical engineering from Imperial College, University of London, in 1977.



Derek Roskell is a design manager in Texas Instruments' Microprocessor and Microcomputer Products (MMP) Division's design center in Bedford, England. He was the design manager for the TMS34010. Roskell joined TI in 1965 and worked on semiconductor test equipment design until 1971. He transferred to the Bipolar Linear Department as a design engineer and worked on I2L design of teletext chips. In 1978 he joined what is now the Microprocessor and Microcomputer Products Division as a project manager on the TMS9995 16-bit microprocessor. In 1986 he became a senior member of the TI technical staff.

Roskell received his BSEE from Leicester College of Technology in 1963.

Karl M. Guttag is the graphics strategy manager for the Microprocessor and Microcontroller Division of Texas Instruments in Houston, Texas. Since 1982 he has been responsible for graphics products definition, including the TMS34010 graphics system processor and the early multiport video RAM definition. From 1979 to 1981 he was the IC architect of the TMS9995 and TMS99000 16-bit microprocessors. He started with TI in 1977 as a design engineer on the TMS9918 Video Display Processor. In 1982 he was made a senior member of the technical staff of Texas Instruments. He currently holds 15 patents in the areas of microprocessors and computer graphics.

Guttag received his BSEE from Bradley University in 1976 and his MSEE from the University of Michigan in 1977. He is a member of IEEE and ACM.

The authors can be contacted at Texas Instruments, PO Box 1443, MS 6407, Houston, TX 77001.