A HIGH-PERFORMANCE RASTER DISPLAY SYSTEM

Roger Bates
Jay Beck[1]
Terry Laskodi
Ed Reuss
Marc Wells

Computer Research Laboratory
Applied Research Group
Tektronix Laboratories
Beaverton, Oregon, USA 97077
(503)-644-0161

John Beatty
Kellogg Booth
Larry Matthies[2]

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L3G1
(519)-886-1351

ABSTRACT

The Geometry Processor is a high speed, programmable, floating point peripheral processor designed to rapidly perform the computations commonly needed for graphics applications. Its firmware is written in a high-level, machine oriented assembly language. A prototypical host software package, based on the GSPC CORE Standard, has also been completed.

KEYWORDS: geometric transform processor, machine oriented language, microprocessor, raster graphics.

Introduction

The economics of integrated circuit fabrication argue forcibly that future computer graphics workstations will make use of low- and medium-cost special purpose hardware to achieve high-performance graphical interaction with raster displays. To gain experience with such hardware, Tektronix Laboratories has designed, built and tested a microprogrammable geometric transform processor to perform the geometric manipulations commonly needed for graphical output. The architecture of this "Geometry Processor" (GP) is flexible enough to support its use either as a solitary graphics peripheral processor (the present configuration) or as part of a multiple-processor pipeline [8,9,18,19]. The Geometry Processor is presently attached to a PDP 11/44 running the Unix operating system.

The GP is programmed in a "high-level microassembly language" which provides both high-level control structures like an IF-THEN-ELSE or DO-loop and a natural syntax for machine instructions. The language provides full and direct access to the bare machine whenever necessary, but automates much of the book-keeping and error checking involved in writing correct horizontal microcode.

The assembler has been used to write a resident firmware package of approximately 1700 microinstructions. This firmware executes macroinstructions stored in host memory. Macroinstructions have been implemented which multiply matrices, apply matrices to homogeneous coordinate vectors, push/pop matrix stacks, line/polygon clip coordinate vectors, viewport transformed/clipped data, and transfer data into and out of the GP.

A host software library modeled on the GSPC CORE [1] has been implemented which allows application programs to pass data to the GP for processing, or to invoke a simulator for the GP. A display package has also been implemented which partially scan converts transformed/clipped polygons from the GP and passes them on to a separate z-buffer display.

The preparation of this paper was supported by the Natural Sciences and Engineering Research Council of Canada under grant A3022.

Geometry Processor Architecture

The Geometry Processor is constructed of about 465 MSI and LSI TTL integrated circuit chips, organized on three boards [2]. It was designed to maximize the speed with which floating point matrix operations can be performed. Microinstructions are 48 bits wide and operate on 32-bit integer and floating point operands. Control of microprogram flow is provided by an Advanced Micro Devices 2910 microprogram sequencer. Floating point hardware is built around a special 24x24 bit LSI multiply chip provided by TRW (not commercially available). Integer and logical operations are implemented by utilizing circuitry in the floating point adder. Two 32-bit parallel i/o ports are provided for GP-controlled i/o, and a separate host-controlled 8-bit *control* or *pilot port* is provided for downloading microcode and for microprogram debugging. The principal processor components and data paths (figure 1) are described below in somewhat more detail.

Data is stored in a 2,048 word by 32 bit *register memory* (RM) built of 55 ns high speed HMOS ram chips. RM is accessed through the A, B and C pointer registers. The A and B registers contain left and right operand source addresses for arithmetic and logical operations; the C register may be used to specify the RM address at which a result is to be stored. The A and B registers may be autoincremented by a variety of values (discussed below), and the C register may be autoincremented by 1. All three registers may be loaded from the data bus, by means of which the principal hardware components communicate.

Operand values may also be stored temporarily in the T register. 32-bit values may be transferred between the T register and the bus, and any byte in the T register may be loaded from any byte of the bus. This enables arbitrary 32-bit values to be constructed from 8-bit values placed on the bus by the pilot port.

[1] Present address: Evans & Sutherland Computer Corporation, 580 Arapeen Drive, Salt Lake City, Utah 84108, (801)-582-5847.

[2] Present address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213 (412)-578-2592.

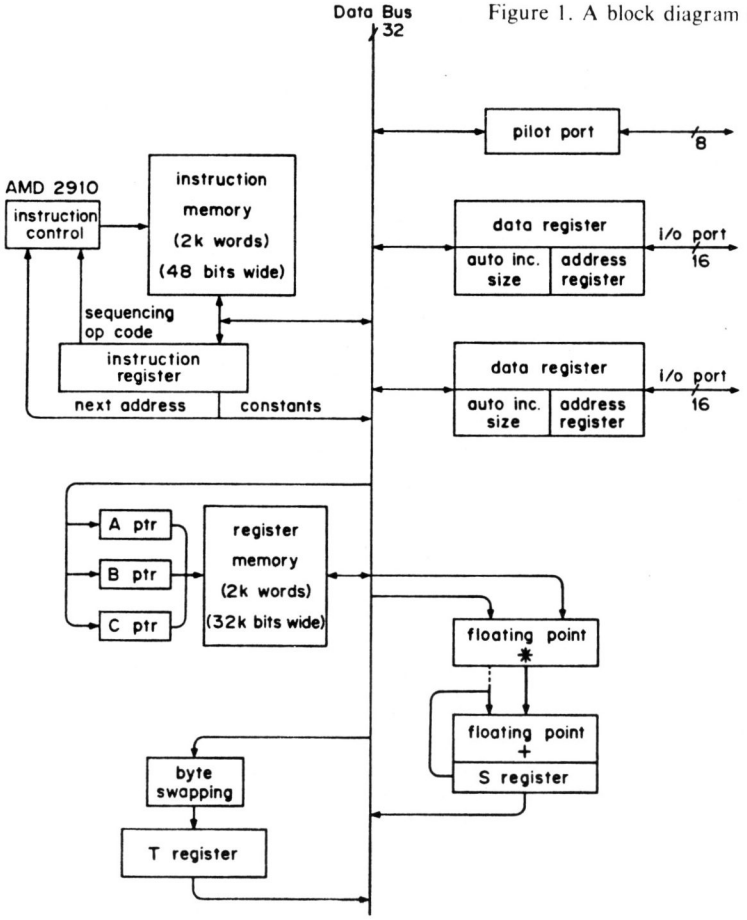
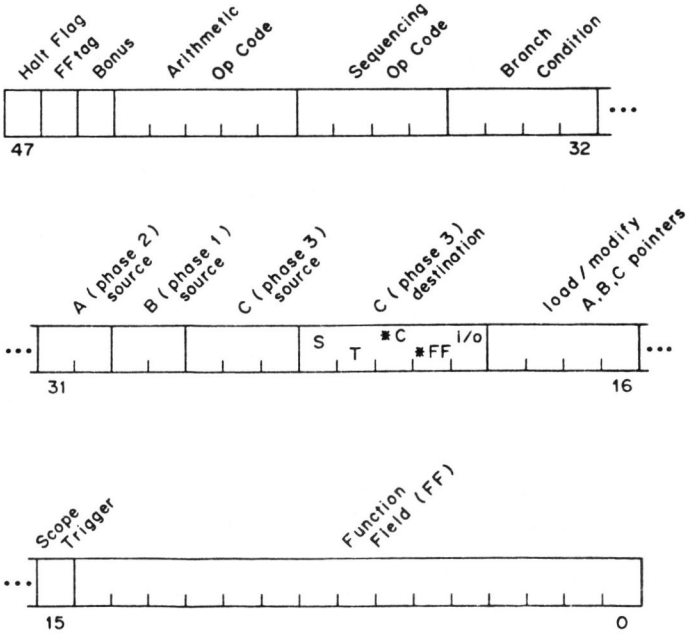


Figure 1. A block diagram of the Geometry Processor.

Figure 2. The layout of a Geometry Processor instruction.



To discuss the floating point subsystem intelligibly we need to make some preliminary remarks about instruction execution and system timing. Execution of a new instruction begins every machine cycle (300 ns). Most instructions complete in a single machine cycle, but the results of floating point additions and multiplications are available only at the end of the following and second following instructions, respectively. More precisely: each machine cycle is sub-divided into three *phases* of 100 ns each. If a microinstruction enables an arithmetic or logical operation, the right and left operands are acquired from the bus during the first and second phases of the instruction, respectively. Results are stored during phase three. However, most arithmetic and logical operations require three additional phases to complete once the operands are available, and their results are therefore stored during phase three of the following instruction.

Multiplications and divisions are an exception to this scheme, as they require six additional phases to complete. The multiplication itself is performed during the first three of these; during the second three the result is optionally added to the contents of the S register and normalized. Hence the result of a multiplication is available for storage during phase three of the second instruction following the instruction in which the multiplication was begun, so that a total of nine phases (three complete machine cycles) are required for operand fetch, computation, and storage of the result.

Thus the location at which the result of an arithmetic or logical operation will be stored is specified in the instruction during which the operation completes, not in the instruction which invoked the operation. This is a major source of programming complexity.

Although these operations require two or three machine cycles to complete, they are pipelined so that successive operations may begin on successive machine cycles. Thus a 4x4 matrix multiplication may be performed in approximately 23 microseconds, and a matrix may be applied to a vector in approximately 7 microseconds.

The GP transfers data between itself and the outside world through two 32-bit parallel i/o ports. Each has a 32-bit data register, a 12-bit address register, and a 4-bit autoincrement register. The hardware attached to an i/o port is expected to have a relocation register which is added to the port address. (In the case of the 11/44, to which the GP is currently attached, the resulting address is run through the Unibus map.) Reading or writing the data register for the selected i/o port triggers the corresponding operation on whatever is attached to the port, and the port address is autoincremented.

Microinstructions are stored in a 2,048 word by 48 bit *instruction memory* (IM) built of 150 ns ram chips. The layout of a microinstruction is indicated in figure 2. The following comments are intended to indicate the range of functionality available. Much unnecessary detail is omitted, for which the reader should be thankful...

If we schematically represent an alu operation by

C := A op B

then the integer and logical operations available are

C := 0
C := B - A - 1 [+1]
C := A - B - 1 [+1]
C := A + B [+1]
C := A xor B
C := A or B
C := A and B
C := all 1's

([+1] indicates that the result may optionally be incremented by 1) and the floating point operations available are

C := A + B
C := A * B + 0
C := A * B + S
C := A * B - S
C := A - B
C := A / B + 0
C := A / B + S
C := A / B - S

(S is the floating point accumulator). Floating point values are represented according to the proposed IEEE standard, and thus possess an 8 bit exponent and (implicitly) a 24 bit mantissa. (The DMA interface connecting the GP i/o ports and the PDP 11/45 is able to convert between DEC and GP floating point formats.) Hardware division results in 8 significant bits since a division by B is replaced with a multiplication by 1/B (an 8-bit value obtained from PROM); true division (required for perspective projection) is accomplished by a 3.5 microsecond firmware algorithm.

The floating point hardware sets status flags to record the occurrence of underflow or overflow, and a value is produced which can be consistently used in a subsequent arithmetic computation. Thus the firmware may check for the occurrence of a floating point error at the end of a sequence of operations and the value produced always makes sense.

The AMD 2910 microprogram sequencer provides a program counter (PC), subroutine call and return using a five word address stack, a loop counter (COUNT), and conditional or unconditional branching. The sequencer op code field selects one of the sixteen standard AMD 2910 instructions; these are listed in the next section. For appropriate 2910 instructions, the branch condition field selects one of the tests BUS=0, BUS>0, BUS>=0, i/o busy, interrupt pending, floating point overflow, and floating point error; the complement of each test may also be selected. The complement of overflow is underflow.

The B source and A source fields of the instruction may specify that the corresponding phase 1 and phase 2 operands are to be the:

(1) T register;
(2) constant defined by the FF field;
(3) RM word pointed to by the B or A register;
(4) RM word whose address is contained in the FF field.

The selected operand is placed on the bus during the appropriate phase.

The C source field selects the value which will be stored by phase 3 of an instruction. The values which may be selected include the:

(1) constant defined by the FF field;
(2) RM word pointed to by the C register;
(3) RM word whose address is contained in the FF field;
(4) alu output from phase 2;
(5) data register for the current i/o port (a read).

The C destination field selects the location at which this value will be stored. Any or all of the following destinations may be selected, in most cases simultaneously:

(1) output from the alu may be written to the S register;
(2) T register;
(3) RM word pointed to by the C register;
(4) RM word whose address is contained in the FF field;
(5) data register for the current i/o port (a write);

Routing alu output to the S register does not tie up the bus during phase three, so that another value may simultaneously be routed to any of destinations (2) through (5).

The "load/modify A, B, C pointers" field may be used to load any combination of these pointers, including all or none of them, from the bus during phase 3, or to cause the A and B pointers to be autoincremented during phase 2 and phase 1, respectively. In the latter case one of eight autoincrement pairs is specified. The corresponding values are defined by a PROM. Six such pairs are currently being used, including [1,0] (increment A by 1 and B by 0) and [1,4] (increment A by 1 and B by 4). The latter pair provides exactly the address modification needed when computing an entry in a matrix product; other pairs are selected for similar reasons. Autoincrement control of the C register is provided by the bonus bit.

Ordinarily the FF field serves as an immediate constant, an address, or is used to load the loop counter (COUNT) in the sequencer chip. (Of particular utility are the flags which allow the real values 1.0 and 0.5 to be generated as immediate constants.) When the FFtag instruction bit is set to 1, however, the FF field may instead serve a number of special purposes. (Exactly which is indicated by its leftmost two bits.) Briefly, these are:

(1) read or write IM from the bus during phase 3;
(2) select an i/o port;
(3) set or clear an i/o interrupt;
(4) load the selected i/o address register;
(5) load the selected i/o autoincrement register;
(6) shift alu output left or right by 1;
(7) reset the floating point error flags;
(8) select bus byte and T byte for a T register load;

The pilot port is not involved in the normal execution of the GP. It serves two purposes: it provides the means by which microcode is downloaded into instruction memory from the host, and it provides a mechanism for debugging firmware.

Data may be transferred in either direction through the pilot port, one byte at a time, between the host and any of the four bytes of the T register. The pilot port may also cause the contents of the T register to be loaded into the instruction register (IR). This enables a host program to manipulate the GP by loading an instruction into the T register, transferring it to the IR, and executing it. Microcode is thus downloaded by assembling a word in the T register and executing an instruction which writes the contents of the T register into either the high 32 or the low 16 bits of a word in IM.

To facilitate this process the pilot port may start, stop, single step and reset the GP. Interactive debugging is performed from the host by halting the GP, saving the contents of the T register and the address from which the IR was loaded (latched in a special register just for this purpose), and inserting an instruction into the IR which causes the contents of an arbitrary register to be moved to the T register, from which it can be read *via* the pilot port. Microprogram execution can then be resumed after restoring all modified registers. Breakpoints may be inserted in such programs by setting the leftmost bit of an instruction to 1; the result is to halt the processor before the instruction is executed so that the pilot port may examine or modify the state of the GP.

Microassembler

As can be appreciated from the preceding section, direct microcoding of the Geometry Processor is a tedious and error-prone occupation, made especially so by the number of fields in each instruction and the pipelining of arithmetic and logical operations. This is, of course, the usual argument for the use of high-level languages. On the other hand, the GP exists for the sole purpose of executing a small number of algorithms at the highest possible speed, motivating the careful preparation of optimal code. This can be done, in general, only by assembly-level programming. Hence the primary goal in designing a microassembler for the Geometry Processor was to provide high-level language constructs wherever possible while retaining the capability for setting specific fields within a microinstruction [6].

Statements within a GP program or procedure consist of declarations, procedure definitions, assignment statements, control statements, branch statements, and assembler directives. Each of these will be discussed below. A general design principle, however, was that each executable statement correspond to the notion of an instruction in a traditional assembler, modulo the effects of pipelining.

The overall intention was, then, to maintain the run-time efficiency and flexibility of machine code while gaining the advantages of a high-level language. This is a philosophy borrowed from Niklaus Wirth's "machine oriented language" PL360 [21]. The GP microassembly language was also much influenced by the work of Don Faul on a language for the ISI/Child programmable frame buffer designed in the mid-1970's at Lawrence Livermore National Laboratory [7].

The language accepted by the GP microassembler is a mixture of B, C, Pascal, and various traditional assemblers. In many cases this amounts to little more than "syntactic sugar" which renders the source code more readable. However, the microassembler will report any attempt to assign conflicting values to a microinstruction field; this feature is particularly useful in the case of arithmetic or logical instructions, which necessarily result in the setting of fields in subsequent microinstructions. To preserve the correspondence between source and object code, the assembler does not attempt to automatically merge consecutive instructions which use disjoint or compatible fields. It is expected that the small amounts of speed critical code involved will be hand optimized by the programmer.

We introduce the GP microassembly language by way of the following example program, which reads 100 homogeneous vectors from the input port, multiplies them by a matrix, and writes the result vectors to the output port. For simplicity the matrix is assumed to have been initialized elsewhere.

```
program Transform;

  const N = 100;

  var rm row[1:4], matrix[1:4,1:4];

  port input = 0, output = 1;
  set port to input,  set port increment to 4;
  set port to output, set port increment to 4;

  B := address(matrix);
  do N times;
    set port to input;
    A := C := address(row);
    [C+1]  := [IO+1];
    [C+1]  := [IO+1];
    [C+1]  := [IO+1];
    [C+1]  := [IO+1];
    S      := [A+1] ** [B+4], set port to output;
    S      := [A+1] ** [B+4]  ++  S;
    S      := [A+1] ** [B+4]  ++  S;
    [IO+1] := [A-3] ** [B-11] ++  S;
    S      := [A+1] ** [B+4];
    S      := [A+1] ** [B+4]  ++  S;
    S      := [A+1] ** [B+4]  ++  S;
    [IO+1] := [A-3] ** [B-11] ++  S;
    S      := [A+1] ** [B+4];
    S      := [A+1] ** [B+4]  ++  S;
    S      := [A+1] ** [B+4]  ++  S;
    [IO+1] := [A-3] ** [B-11] ++  S;
    S      := [A+1] ** [B+4];
    S      := [A+1] ** [B+4]  ++  S;
    S      := [A+1] ** [B+4]  ++  S;
    [IO+1] := [A+1] ** [B-15] ++  S;
    nop;   /* Wait for the result */
  end do;  /* The final store occurs here */

  $t$      /* Dump the symbol table */

end program Transform.
```

Since the assignment, control and branching statements described below often make use of distinct microinstruction fields, the programmer may make them part of the same statement (instruction) by separating them with a comma; a semicolon marks the end of such a list, and indicates that the assembly of a new instruction should begin. Thus

```
S := [A+1] ** [B+4], set port to output;
```

results in an instruction whose fields specify the initiation of a multiplication and the selection of an i/o port. It will be convenient to refer to the pieces of such a composite statement as "actions". If the actions in a statement require that any field of the corresponding instruction have two or more distinct values then an error message is generated.

We shall now sketch, in turn, the various kinds of statements accepted by the GP microassembler.

Declarations such as

```
const N = 100;
port input = 0, output = 1;
field f1 = BITS{15:8}; /* Third byte in a data word */
field f2 = FF{11:0}; /* branch address in FF */
```

provide for the creation of symbolic names for constants, i/o ports, and bit fields. Notice that the register names discussed in the previous section (such as FF, A, B, C, S, T and IO) are known to the assembler and may be used wherever appropriate. The name itself (such as A) represents the contents of the register, while enclosing the name in square brackets (for example: [A]) refers to the value pointed to by the address in the register. Variable declarations such as

```
var rm row[1:4], matrix[1:4,1:4];
```

may be used to associate a symbolic name with a block of storage in register memory (rm - the default) or instruction memory (im). Storage is allocated statically, so that a unique portion of the appropriate memory is permanently associated with every declared variable. The scope of an identifier is the entire program or procedure in which it is declared. Procedures may be nested. However, aside from defining the scope of identifiers declared within it, a procedure definition is equivalent to the definition of a label at the same location; there is no saving of registers, return addresses, or anything else at procedure entry and when a procedure exits the next instruction is simply executed. Label and procedure names must be declared before they are defined and the scope of the names is the scope of the declarations. Since the assembler operates in a single pass, every identifier must be declared before it is used.

Variables are not typed. Instead the floating point version of an arithmetic operation is distinguished from the integral version by repeating the operator. Thus + denotes integer addition and ++ denotes real addition. In order to preserve the correspondence between assembler statements and machine instructions, the assembler accepts as the right hand side of an assignment statement only those expressions whose computation can be initiated in a single microinstruction, such as

```
S := [A+1] ** [B+4]  ++  S;
```

Only the small number of autoincrement/autodecrement values implemented directly by the hardware are allowed. Multiple assignments such as

```
A := C := address(row);
```

are allowed in so far as they comprise compatible phase 3 destinations.

Control statements such as

```
SET PORT INTERRUPT
RESET PORT INTERRUPT
ENABLE PORT BASE LOAD
ENABLE PORT ADDRESS LOAD
SET PORT TO <0 or 1>
SET PORT INCREMENT TO <expr>
RESET FP ERROR
```

modify the state of flags in the GP in the obvious way.

A variety of branch statements allow the microprogrammer to specify the microinstruction fields which control program flow.

Most of these statements make use of conditional branch features of the AMD 2910 sequencer chip. The conditions available are

WHEN BUS = 0	UNLESS BUS = 0
WHEN BUS != 0	UNLESS BUS != 0
WHEN BUS > 0	UNLESS BUS > 0
WHEN BUS < 0	UNLESS BUS < 0
WHEN BUS >= 0	UNLESS BUS >= 0
WHEN BUS <= 0	UNLESS BUS <= 0
WHEN INTERRUPT	UNLESS INTERRUPT
WHEN BUSY	UNLESS BUSY
WHEN FP ERROR	UNLESS FP ERROR
WHEN FP OVERFLOW	
WHEN FP UNDERFLOW	
ALWAYS	NEVER

BUS conditions test flags indicating the value of the bus during phase 3 of the previous microinstruction. BUS may be replaced by any valid phase 3 source. If a pipelined operation is specified in the same instruction then the test is delayed until the operation is complete.

The various branch statements available make use of the COUNT register and address stack in the AMD 2910. The COUNT register may be used as the control variable of a loop, or may be loaded with an address through which a jump may subsequently be executed. The address stack may be used for nesting subroutine calls, although the hardware does not provide any mechanism for trapping stack overflow and the assembler does not attempt to predict such an eventuality. The primitive branching instructions, which correspond exactly to the capabilities of the hardware, are as follows:

GOTO <target>
GOTO <target> <condition>
GOTO <target> <condition> ELSE GOTO [COUNT]
GOTO [COUNT]

CALL <target>
CALL <target> <condition>
CALL <target> <condition> ELSE CALL [COUNT]
CALL [COUNT]

A <target> is either a label or a procedure. A condition is a WHEN or UNLESS clause. The COUNT register is assumed to have been loaded previously with the address of interest. A CALL pushes the current contents of the program counter on the address stack and transfers control to the <target> address.

PUSH
PUSH THEN COUNT <expr> TIMES
PUSH THEN COUNT <expr> TIMES <condition>
PUSH THEN LOAD <target> INTO COUNT
PUSH THEN LOAD <target> INTO COUNT <condition>
ITERATE ELSE POP
ITERATE ELSE POP TO <target>
COUNT <expr> TIMES
LOAD <target> INTO COUNT

PUSH simply causes the program counter to be pushed on the address stack. COUNT <expr> is used to initialize the COUNT register to <expr>-1 at the top of a loop; presumably a subsequent ITERATE statement will cause a branch back to the top-of-stack-address after decrementing COUNT if COUNT is non-zero, so that the loop will be executed exactly <expr> times. When COUNT goes to zero the address stack is popped and control either proceeds sequentially or is passed to the specified <target>. LOAD <target> initializes COUNT with <target>, which will presumably be used later as a jump address.

RETURN
RETURN <condition>
LOOP
POP
POP <condition> ELSE LOOP
POP TO <target>
POP TO <target> <condition>
POP <condition> ELSE ITERATE ELSE POP TO <target>

RETURN causes the stack to be popped after transferring to the top-of-stack address, LOOP simply causes control to return to the top-of-stack address, and POP simply pops the stack.

At a somewhat higher level, a construct much like the traditional "if-then-else" is accepted by the assembler. It is best described by an example.

```
T := [A] & [B], WHEN BUS = 0 THEN;
  ...
ELSE square := T ** T;
  ...
ENDIF [C] := T;
```

The "THEN-part" consists of the statements up to and including the statement which begins with ELSE, while the "ELSE-part" consists of the statements following the statement containing ELSE, up to and including the statement beginning with ENDIF. The <conditional> appearing in the first statement may be either a WHEN or an UNLESS, and selects which clause is to be executed. In this example the THEN-clause has as its last action a floating point multiply, which will require two machine cycles to complete. The assembler would normally arrange that the second following instruction cause the result to be stored into the variable square. Unfortunately this instruction lies in the ELSE-clause, and would not be executed. To avoid such anomalies the pipeline is always allowed to empty at the start of the conditional, and at the end of the THEN- and ELSE-clauses, by inserting NOP instructions as needed.

The GP assembler also accepts do-loops such as

```
T := T ** T, DO 10 TIMES;
  ...
ENDDO T := T ++ 1.0;
```

The optional action (here T := T ** T) is performed, COUNT is loaded with the number of iterations minus one, and the top-of-loop address is pushed onto the address stack. The body of the loop, including the final action on the ENDDO, is executed the specified number of times. Once again, the pipeline is allowed to empty at the top and bottom of the loop by inserting NOPs.

The assembler simplifies the programming of arithmetic and logical operations by automatically inserting the destination specified in a subsequent instruction, and verifying that this does not result in a conflict with other actions in that instruction. The

programmer must, of course, be aware that this will happen. When a branch occurs, however, the textually following instructions need not be the next instructions executed. As we have seen, the microassembler minimizes errors by "flushing" the pipeline whenever a branch is specified by inserting one or two NOP instructions prior to the branch so that the result may be stored before the branch is taken. For the sake of efficiency it is occasionally desirable to suppress this padding. Appending the keyword IMMEDIATELY to an action accomplishes this; notice, however, that the first or second textually following instruction will still contain the phase 3 source and destination fields needed to store the result. For example, the following loop will move N words using 3+2*N instruction cycles.

```
C := ADDRESS( <source> );
A := ADDRESS( <destination> );
DO N TIMES; /* N is a constant */
ENDDO [C+1] := [A+1] ++ 0;
```

while

```
C  := ADDRESS( <source> );
A  := ADDRESS( <destination> );
LOAD N-2 INTO COUNT; /* N-2 is computed by the assembler */
[C+1] := [A+1] ++ 0, PUSH IMMEDIATELY;
[C+1] := [A+1] ++ 0, ITERATE ELSE POP IMMEDIATELY;
NOP;
```

will require only 4+N instruction cycles.

When all else fails, the bits in a specific field of an instruction may be set directly by means of an action such as "BITS{5:4}=2".

A more detailed description of the GP assembly language may be found in [6]. The description supplied here should be sufficient to give the reader a feeling for the way in which the somewhat conflicting desires both for full, efficient access to the machine and for the simplicity and expressiveness of high-level language constructs have been resolved.

Programs are run through the C preprocessor [11] before undergoing assembly proper so that macros (with parameters), file inclusion, and conditional assembly are all supported. The assembler recognizes the line number and file markers embedded in preprocessor output so that error messages may specify correct line numbers and file names.

The assembler contains a lexical analyzer generated by LEX [14] and an LALR(1) parser generated by YACC [10]. Use of these tools greatly eased both implementation and maintenance of the microassembler.

Firmware

The motivation for designing the Geometry Processor and a microassembler with which it can be programmed was to construct a special purpose peripheral processor which could be used to rapidly perform the geometrical transformations and clipping operations necessary in 3-dimensional computer graphics applications.

Because the GP is progammable it can readily be adapted to a variety of system organizations. In the present configuration the GP is connected to a PDP 11/44, from which it reads instructions and data, and to which it returns results, although transformed and clipped output could equally well have been passed directly on to a second special purpose processor for scan conversion and display.

The present GP firmware, then, is periodically given the address of a macroinstruction list in host memory, which it executes. These macroinstructions are each 8 bits in length, and are either control, matrix, or operand instructions. The firmware maintains a number of variables in register memory, which various macroinstructions depend upon or modify, including an i/o address, two matrix stacks, a viewport transformation, and an operand/result data array. Within the GP vertices are always represented in homogeneous coordinates.

There are four control instructions. NOOP has no effect. STOP halts the GP after notifying the host of completion. RESET empties either or both of the matrix stacks. SETADDRESS may be followed by up to 3 bytes, and specifies an address in host memory (1) at which the next macroinstruction will be found, or (2) which a subsequent instruction will use for reading or writing matrices or data.

There are five matrix instructions. Each may modify either or both of the matrix stacks. MOVEM causes the matrix stored at the current i/o address in host memory to be pushed onto the selected stack(s), or returns one or both of the top-of-stack matrices to the host. DUPM causes the matrix at the top of the selected stack(s) to be duplicated, increasing the stack depth(s) by one. POPM causes the matrix at the top of the selected stack(s) to be released, decreasing the stack depth(s) by one. MULM causes the two matrices topmost on the selected stack(s) to be replaced by their product, again decreasing the stack depth(s) by one. APPLYM has exactly the same effect on the selected stack(s) as would a DUPM, followed by a MOVEM, followed by a MULM.

Matrices are transferred from the host in a "parameterized" form which contains only the non-zero entries. Thus the host representation of the matrix for a translation in x and y has a header containing a code for translation and bit flags indicating that dx and dy values are present; this 16 bit header is followed by the two 32-bit floating point values of dx and dy. (The MOVEM instruction expands the parameterized form into a full 4x4 matrix before pushing it onto the stack(s).) The parameterized representations of scaling and rotation matrices are analogous. There is also a full 4x4 format for matrices which do not fit this scheme.

The operand instructions are considerably more heterogeneous. The MOVEO instruction is used to transfer data between host memory and the GP operand/result array. There are four bit flags in addition to the four-bit op code. The first specifies the direction of data movement. A second indicates whether conversion is desired between homogeneous and non-homogeneous form. There is also a bit flag for each of the stacks. If the flag for stack i is set then the topmost matrix on stack i is applied to each 4-tuple transferred. If both flags are set then the topmost matrices on each stack are applied to alternate 4-tuples. (Thus if data being read from the host consists of 8-tuples comprised of a position and a normal, and if stack one is a transformation stack and stack two records only the rotations which have been accumulated, then we are able to apply the current transformation to positions and compute the effect of rotations on normals.)

The LINECLIP and POLYCLIP instructions cause a standard line or polygon clipping algorithm to be applied to the data in the operand/result array. Clipping against the near, far and side clipping planes may be individually enabled or disabled. Clipping automatically results in the conversion of vertices from

homogeneous to non-homogeneous form; the fourth coordinate is replaced by a "move/draw" flag. The POLYCLIP instruction has an additional flag which can be set to indicate that each incoming homogeneous coordinate carries with it a normal which should be interpolated, not clipped.

Finally, the VIEWPORT instruction is used to apply the current viewport transformation to the data in the operand/result array, or to extract a lightsource dot-product from the data in the operand/result array, or both (in which case the two kinds of data alternate). The lightsource computation amounts to no more than replacing a 4-tuple by its third coordinate, on the assumption that the 4-tuples to which it is applied are normals which have been transformed into a coordinate system in which the z-axis is parallel to the light source vector.

Thus the macroinstructions presently implemented allow the Geometry Processor to stack or concatenate the modeling and viewing matrices commonly used in graphics applications, to apply these matrices to vertex or normal data, and to apply a viewporting transformation. Scan conversion is the responsibility of a subsequent processor.

A detailed description of the way in which the Geometry Processor is interfaced with the PDP 11/44 to which it is presently attached may be found in [5,12,13,20]. Briefly, to use the GP the requesting process is locked in core. The pilot port is then used to download the firmware, initialize the i/o port base addresses, and patch into an agreed-upon location in instruction memory an address in the host program at which an instruction list pointer, error return code field, run/halt flag and completion flag can be found by the GP. The pilot port is then used to start the GP executing. A control loop in the GP firmware samples the run/halt flag to determine when a new instruction list has been presented for execution. Execution of the STOP instruction terminating the instruction list is indicated by setting the completion flag and generating an interrupt, which is propagated to the host program as a signal.

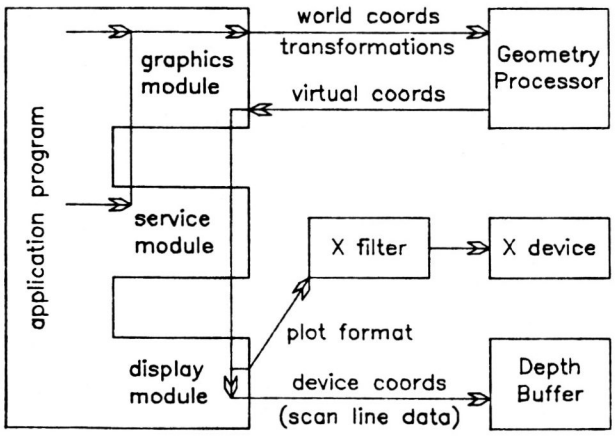


Figure 3.

Host Software

A schematic of the host software with which an application program may currently use the Geometry Processor appears in figure 3.

The graphics module, which is the actual interface between a program and the GP, contains the following kinds of routines.

1) Control Routines. These are used to initialize and terminate the graphics module, and provide a means of switching between a *data mode* in which lines, polygons and transformations are passed to the GP for processing, and a *viewing mode* in which the viewing transformation is specified.
2) 3D Viewing Routines. For the most part these conform to the GSPC CORE Standard [1]. For convenience, however, a *camera mode* is available in which the view plane normal is automatically determined to be the vector from the eye point to the view reference point (suggested by Tim Stevenson). Also, following [17], a foreshortening ratio and receding angle are used to specify parallel projections.
3) Modeling Transformations. These routines cause an appropriate matrix to be passed to the GP, where it is concatentated to the current transformation and lighting matrices (in the case of a rotation). These matrices may first be saved on stacks in the GP. The lighting matrix is initialized so as to transform normals to which it is applied into a coordinate system in which the light source vector is aligned with the z-axis. The projection of a unit normal onto the light source vector is then simply the z coordinate of the vector which results from applying the lighting matrix to the normal.
4) Data Routines. Vertices are passed to the GP in arrays. The vertices defining a polygon may optionally carry a normal which the GP will interpolate while clipping. Information about the direction and nature of the lightsource may also be specified, since the GP can be asked to apply the usual cosine law to compute the intensity at a vertex from an associated normal and a light source vector.

The display module is a separately compiled set of routines which can be used to display data transformed by the graphics module and GP. The primary output device used while debugging the GP was a 512x512 z-buffer having 24 bits of colour information and 16 bits of depth at each pixel [15,16]. The display module breaks polygons into scan line segments, which a Z80 contained in the display (slowly) processes. Constant, facet and Gouraud shading (with and without depth modulation) are supported (figure 4). Alternatively, the display module can also generate images on a variety of line drawing displays *via* the standard Unix plot utility.

The general scheme, then is as follows. An application program will build and manipulate data structures containing coordinate information, modeling transformations, colour information, and additional application dependent object attributes. When a picture is desired, appropriate viewing parameters are passed to the graphics module, and the Geometry Processor is initialized with a transformation effecting the desired viewing projection. The application data structures, which typically contain a mixture of modeling transformations and polygon data, are then traversed.

As each transformation or polygon is encountered during traversal,

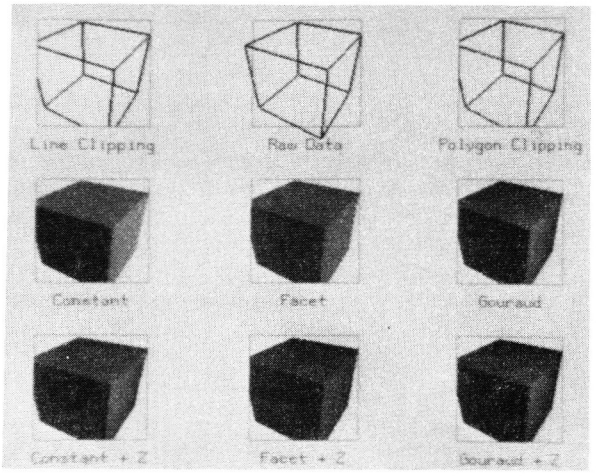


Figure 4.

+ for each rotation, translation, or scaling desired, a graphics module routine is called to request that the GP update the current transformation and lighting matrices, possibly altering the stack as well, or
+ the coordinates defining a polygon (or line trace) are placed in an array and an appropriate graphics module routine is called to transmit the array to the GP, where it is transformed, clipped and viewported.

The resulting polygons, now described in virtual coordinates, are then retrieved from the GP and passed to a display routine, where shading computations are performed, and scan segments are passed on to the z-buffer for depth-comparison with the current image and possible display.

The application program may call routines in the graphics module and in the display module directly, but for convenience a standard set of "service" routines have been provided which send polygons off to the GP to be processed and automatically pass returned virtual data on for display.

Debugging

An important part of the Geometry Processor's design was the provision of convenient mechanisms for debugging. At the lowest level, bit 15 of the instruction is not used by the GP itself. When the hardware is thought to be malfunctioning, a small loop which includes the offending instruction is repeatedly executed. Bit 15 of this instruction only is set to 1 and used as a "scope trigger" while pertinent signals are probed and compared with their nominal values. Thus faulty signals can be easily located.

At a slightly higher level, the HALT bit can be used in conjunction with the pilot port to implement a fully interactive, host-resident microprocessor debugger. Indeed, firmware debugging from the PDP 11/44 via the pilot port is no more difficult than the interactive debugging of host programs. One can hardly exaggerate how greatly this facilitates firmware debugging and maintenance.

Development of the resident firmware and host software were also facilitated by writing a simulator for the GP which ran on the host [4]. By loading with the appropriate interface routines host application programs were able to run either the GP hardware or simulator. Comparing the results for a given test program made it easy to determine whether anomalous results were due to GP firmware/hardware problems or to bugs in the test driver and host software library. Because the simulator was written before the firmware, the GP instruction set had evolved to a stable state before microcoding began. Existence of the simulator also made it possible for host software development to proceed normally when the GP was unavailable.

Conclusions

The Geometry Processor hardware, firmware and host software are fully operational, and the first phase of the project has been brought to a close; work has now shifted to the development of a 1000 line, Motorola 68000 based personal workstation to which the GP will be optionally attached. Hence this is a convenient point at which to take stock of what we have learned.

The scope trigger and breakpoint bits are invaluable. So to is the provision of a mechanism like the pilot port for monitoring and debugging the microprocessor interactively from the host.

On the minus side, the A, B, and C pointer registers cannot be read owing to a lack of board space and because of timing considerations, nor can the 2910 address stack. These restrictions are a substantial inconvenience; it should be possible to read anything which can be written.

It is now clear that it would have been simpler to use separate circuitry for integer and logical operations, rather than arranging for the floating point hardware to handle these as well. Indeed, most of the arithmetic (including address computations) performed by the GP is integral. Also, a relatively simple and useful modification to the GP would allow the transfer of a value between RM locations in one machine cycle instead of two by making it possible to load the T register during any phase, instead of only phase 3. Finally, the primary source of difficulty in fully utilizing the parallelism available in a GP microinstruction arises from the multiplicity of uses for the FF field. The most common conflict results from an attempt to use FF both as an address and for an immediate constant.

A more important observation has to do with what the GP spends its time doing. As is apparent from the design of the alu pipeline, the major emphasis was placed on rapidly computing matrix products, and rapidly applying matrices to vectors. Since the GP is programmable, it seemed a natural place at which to clip and viewport as well. Roughly speaking, nearly 2800 instruction cycles are required to read a four vertex polygon from the host, apply the current transformation matrix, clip the result (assuming the polygon leaves and re-enters the clipping frustrum once), apply the viewporting transformation, and return the processed polygon to the host. Of these only about 75 instruction cycles are actually involved in applying the current transformation. The most expensive step is polygon clipping (about 2275 cycles), followed by viewporting (about 285 cycles). Also, to use the GP most effectively a separate high speed scan conversion processor is needed; further work on such processors is anticipated.

The effort of designing and implementing a structured assembler was clearly worthwhile. Indeed the GP assembly language has already been used as a model for the design of a language for another microprocessor. It is not clear, however, that it was wise to stay so close to the hardware in designing the language accepted by the GP assembler. Even at the level of the Geometry Processor, programs consist of a few inner loops which should be executed rapidly and a lot of setup code whose execution time is relatively unimportant. The latter is shot full of special cases, decision making, and control logic of the kind found in ordinary programs, whose syntax is clearly easier to understand than is the syntax accepted by the GP assembler (which closely reflects the hardware).

Indeed a better approach may be to strip unnecessary features out of an existing language like C and compile relatively inefficient code for the microprocessor, dropping into a primitive assembly language within procedures to implement the small inner loops on which total execution time mostly depends. We are pursuing this alternative in designing a high level language for a comparable microprocessor which is attached to an Ikonas frame buffer system at Waterloo, with the intention of comparing the two approaches.

Note

The hardware described in this paper is experimental, and should not be construed as a product commitment by Tektronix, Inc.

References

[1] "Status Report of the Graphics Standards Planning Committee," *ACM Siggraph Quarterly*, 13,3 (August 1979).

[2] Bates, Roger D., *Special Purpose Geometry Processor Architectural Description*, Technical Report CR-79-12, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[3] Bates, Roger D., *Special Purpose Geometry Processor Pilot Port to DEC 11/xx Architectural Description*, Technical Report CR-80-11 (April 1980), Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[4] Beatty, John C., *A User Interface for the CRL Geometry Processor*, Technical Report CR-81-8, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[5] Beck, Jay W. and John C. Beatty, *Special Purpose Geometry Processor Functional Specification*, Technical Report CR-80-14, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[6] Booth, Kellogg S. and Marc Wells, *Special Purpose Geometry Processor Microcode Assembler*, Technical Report CR-81-11 (July 1981), Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[7] Faul, Donald R., *The Design and Implementation of a High-Level Language for a Programmable Frame Buffer*, Technical Report UCID-17745 (October 1977), Lawrence Livermore National Laboratory, Livermore, California 94550.

[8] Fuchs, Henry, "Distributing a Visible Surface Algorithm Over Multiple Processors," *Proceedings of the ACM National Conference 1977*, October 1977.

[9] Fuchs, Henry and B. Johnson, "An Expanded Architecture for Video Graphics," *Proceedings of the Sixth Symposium on Computer Architecture*," April 1979.

[10] Johnson, Steven C., *YACC - Yet Another Compiler Compiler*, CSTR 32 (1974), Bell Telephone Laboratories, Murray Hill, New Jersey.

[11] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).

[12] Laskodi, Terry, *Special Purpose Geometry Processor - User Software Interface*, Technical Report CR-80-2, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[13] Laskodi, Terry, *Special Purpose Geometry Processor - Software Generation and Maintenance* Technical Report CR-80-3, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[14] Lesk, M., *LEX - A Lexical Analyzer Generator*, CSTR 39 (1975), Bell Telephone Laboratories, Murray Hill, New Jersey.

[15] Matthies, Larry H., *FBLIB - The Frame Buffer Support Library*, CRL PDP-11/70 online documentation FBLIB(7), Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[16] McCann, Ben and Larry H. Matthies, *The CRG Frame Buffer*, Technical Report CR-80-16, Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[17] Michener, James C. and Ingrid B. Carlbom, "Natural and Efficient Viewing Parameters," *Computer Graphics*, 14,3 (July 1980) pp 238-245.

[18] Parke, Fred I., *A Parallel Architecture for Shaded Graphics*, Technical Report, Computer Engineering Department, Case Western Reserve University, January 1979.

[19] Parke, Fred I., "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems," *Computer Graphics* 14,3 (July 1980) 48-56.

[20] Reuss, Ed, *Transform Processor to PDP-11 DMA Interface User Description*, Technical Report CR-80-19 (September 1980), Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon 97077.

[21] Wirth, Niklaus, "PL360 - A Programming Language for the 360 Computers," *Journal of the ACM*, 15,1 (1968) 37-74.