United States Department of Commerce
National Institute of Standards and Technology

NIST Technical Note 1288

Video Processing With the Princeton Engine at NIST

Bruce F. Field and Charles Fenimore

NIST Technical Publications

Periodical

Journal of Research of the National Institute of Standards and Technology—Reports NIST research and development in those disciplines of the physical and engineering sciences in which the Institute is active. These include physics, chemistry, engineering, mathematics, and computer sciences.
Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Institute's technical and scientific programs. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Institute's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NIST, NIST annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NIST under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published bi-monthly for NIST by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW., Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Institute on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NIST under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NIST administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NIST research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NIST publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NIST publications—FIPS and NISTIRs—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NIST pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NIST Interagency Reports (NISTIR)—A special series of interim or final reports on work performed by NIST for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.

NIST Technical Note 1288

Video Processing With the Princeton Engine at NIST

Bruce F. Field
Charles Fenimore

Electricity Division
Electronics and Electrical Engineering Division
National Institute of Standards and Technology
Gaithersburg, MD 20899

August 1991

U.S. Department of Commerce
Robert A. Mosbacher, Secretary

National Institute of Standards and Technology
John W. Lyons, Director

National Institute of Standards
and Technology
Special Publication
Natl. Inst. Stand. Technol.
Tech. Note 1288
51 pages (Aug. 1991)
CODEN: NTNOEF

U.S. Government Printing Office
Washington: 1991

For sale by the Superintendent
of Documents
U.S. Government Printing Office
Washington, DC 20402

Table of Contents

The NIST Program in Digital Video

Program Objectives

The Institute has embarked on a program of measurement technology for advanced imaging systems as part of its mission to provide support to industry and government in the development of measurement techniques and standards. The program is designed in part to respond to the emerging technologies for digital video processing by developing the technical basis for making measurements and setting standards.

The first major component of the program is the creation of the NIST Video Processing Laboratory, a real-time, video processing facility centered around a special purpose video supercomputer, the Princeton Engine. The Princeton Engine was developed by the David Sarnoff Research Center and provided to NIST by the Defense Advanced Research Projects Agency (DARPA) because NIST is open to government and industry users and has a tradition of independence and objectivity. It is intended that this program will contribute to the development of generic technology for image and video processing through open collaborations with other government agencies, universities, and industry. We will also cooperate with, and provide technical information to, voluntary standards organizations.

Outside Users

Although provided to NIST primarily to support DARPA contractors developing improved video and imaging systems, other academic and industrial researchers working on digital video processing, storage, and transfer may apply for access to the NIST Video Processing Laboratory and use of the Princeton Engine. Projects which contribute to the development of measurement technology and of open, interoperable, standards are of special interest. Because NIST strives to contribute to the development of measurements in an open manner, research which is principally proprietary or which has immediate commercial impact, especially in the consumer electronics market, is not appropriate. Those projects which are suitable for collaborative research with NIST personnel and which exploit the capabilities of the Princeton Engine at NIST will be given a high priority.

The purpose of this publication is to summarize for potential users the resources of the NIST Video Processing Laboratory including the capabilities of the Princeton Engine. It is our hope that this information will enable you to assess the applicability of the Princeton Engine and of the NIST facility to your projects. Interested users may contact the technical personnel listed on page 25.

The NIST Video Processing Laboratory

Facility Description

The NIST Video Processing Laboratory has been created to provide hardware and technical support for governmental, industrial, and academic researchers working on digital video processing. It is located at the NIST Gaithersburg campus and offers users access to laboratory video equipment and office space.

The centerpiece of the facility is a video supercomputer, the Princeton Engine. Designed and constructed by the David Sarnoff Research Center in Princeton, NJ, it was delivered to NIST in April 1991. The Princeton Engine provides real-time video and image-processing capability. It can accept a variety of video formats over multiple, wideband input channels and can output NTSC, high definition, or other video formats. Because the Princeton Engine is programmable, it is possible to use it to evaluate prototypes of video processing components rapidly and at a cost below that of building hardware. The Princeton Engine at NIST is the only one open to governmental, industrial, and academic users.

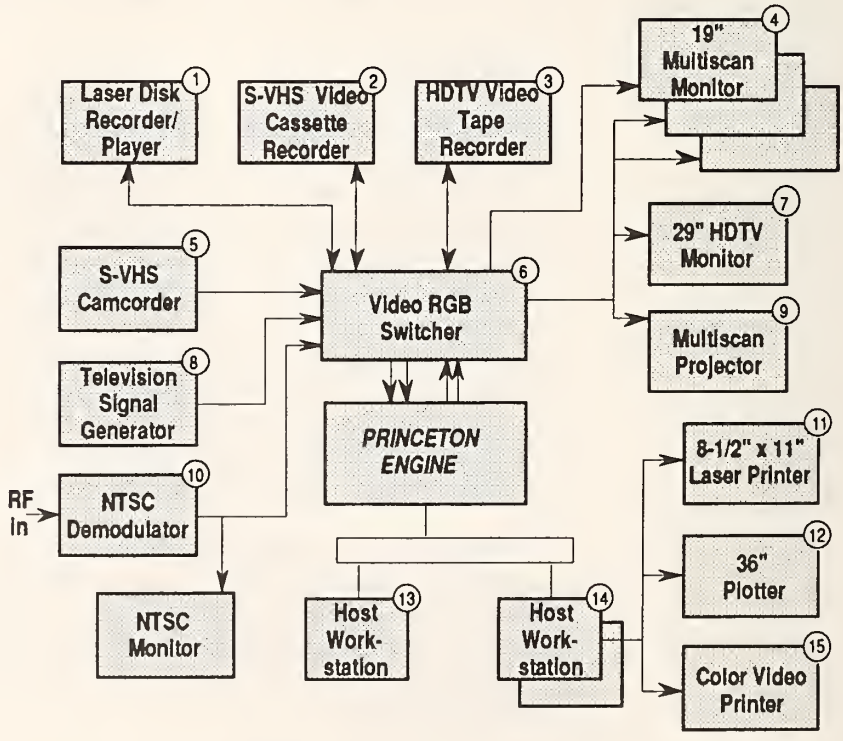


Fig. 1. Approximate representation of equipment configuration available in the NIST Video Processing Laboratory.

Supporting Equipment

The specific supporting equipment available with the Princeton Engine is evolving, however, figure 1 describes the laboratory as it soon will be configured. Typical operation involves connecting a video source to the Princeton Engine through the video switcher, downloading an executable code segment from a host workstation, and viewing or recording the processed video output on a monitor or video recorder.

The listing below includes a more complete identification of the available equipment. (Numbered items are keyed to the numbered circles attached to the blocks in figure 1.) The use of specific product names does not indicate that the item is the best available for the application nor does it constitute an endorsement by NIST; names are shown only to clearly identify the equipment in use.

1. Sony LVS-5000A, Laser Disk Processor and Recorder/Player, with 12" monitor
2. JVC RRS600U, S-VHS Video Cassette Recorder, with 400-line resolution
3. Sony HDD1000PAC, HDTV Digital Processor and Recorder/Player (*)
4. Barco ICD451B, 19" Multiscan Video Monitor (3 units)
5. Panasonic AG540, S-VHS Camcorder, with character generator
6. Dynair FR-8704A, RGB Video Switcher
7. Shibasoku CM65B6, 29" HDTV Multiscan Monitor
8. Tektronix TSG 1001, Programmable Television Signal Generator
9. Sony, Multiscan Projector
10. Videotek DM141S, NTSC Demodulator
11. QMS 820, 8-1/2" x 11" Laser Printer
12. Calcomp 58436XP, 36" Plotter (*)
13. Apollo DN400tc, Color Graphics Workstation (*)
14. Apollo DN4500, Color Graphics Workstation (2 units)
15. Shinko CHC-743MV, Color Video Printer

Not shown, but also available:
Lyon-Lamb RTC, Converter
Lyon-Lamb ENC, Encoder/Transcoder
JVC RRS600U, S-VHS Video Cassette Recorder with 19" Monitor

(*) not presently available, to be delivered

In addition to the equipment listed above, the workstations in the laboratory are linked to other workstations at NIST (and to the Internet) for data transfer to and from a variety of additional disk and tape storage units. Generally, data transfer to and from the Princeton Engine is accomplished through the high-speed video channels. But, small amounts of data can be downloaded from the host workstations, or captured from the Princeton Engine outputs and saved on a host workstation, if necessary.

The Princeton Engine

General Description

The Princeton Engine was developed at David Sarnoff Research Center, originally to provide television system developers with the capability of simulating video systems in real-time. It processes a video signal one scan line at a time, performing either an identical set of operations on each scan line, or one of several sets of operations in a line-dependent manner. Field and frame processing is accomplished by storing samples of successive scan lines in processor memory. "Programs" resemble electronic circuit diagrams and are developed using computer-aided-design (CAD) tools on a host workstation. Instead of electronic components that are connected by wires, the "circuit" consists of functional modules, representing predefined computational subroutines, that are connected by data flow paths. After compilation, the object code is downloaded to the Princeton Engine and run in real-time.

The ability to make changes to the circuit diagram and re-run the modified simulation quickly, as well as the ability to define run-time user parameters, allows the Princeton Engine to serve as a testbed for new system/circuit designs where the engineer can ask "what if?" and observe the results as real-time video. The architecture and programming environment is designed to enable the user to simulate digitally, in real-time, very complex analog and digital video processing devices.

A simplified diagram of the architecture is shown in figure 2. The Princeton Engine is a Single-Instruction-Multiple-Data (SIMD) massively parallel supercomputer. That is, all the processors execute the same instruction simultaneously but use different input data. In its present configuration at NIST it has 1024 processors.

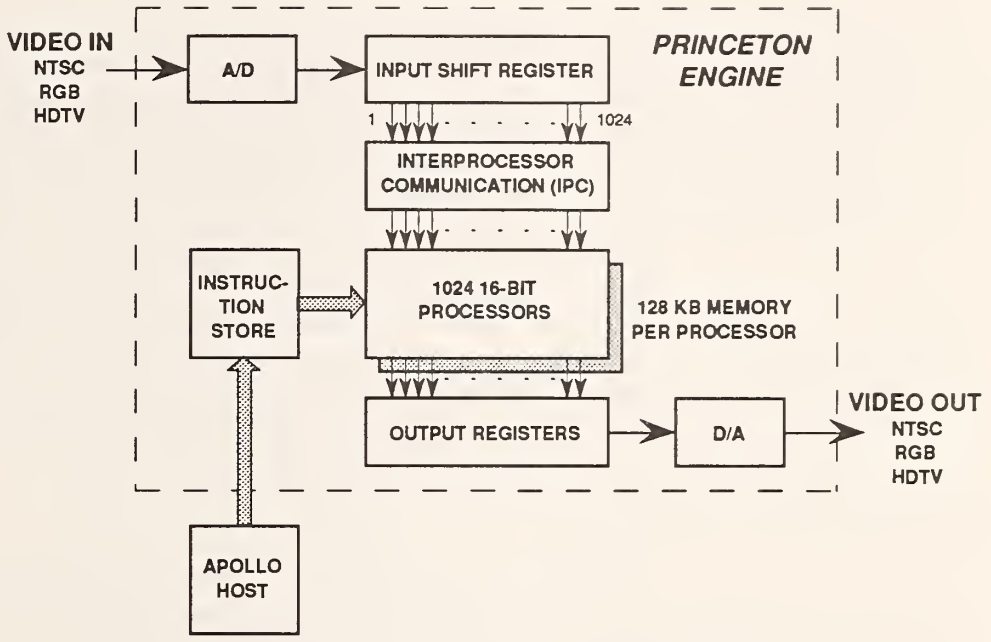


Fig. 2. Simplified functional diagram of the Princeton Engine and the Apollo host workstation.

Data Flow Within the Princeton Engine

The architecture of the Princeton Engine is one of the most distinctive features of the machine. As shown in figure 2, the incoming video data stream (either composite or component) is sampled and converted from analog to digital form, line-by-line, by one of several 8-bit analog-to-digital (A/D) converters. The sampling rate can be set by the user to 14.32, 28.64, or 57.27 MHz. Additional circuitry (not shown) provides synchronization to the scan line rate for NTSC, PAL, and several HDTV formats. Moreover, this circuitry may be software-configured to employ sampling rates that are independent of the syncronization rate.

As the data samples are acquired along the scan line, they are moved serially into the upper shift register, and once each scan line the samples are moved in parallel directly to the processors, one sample or pixel per processor. Each processor operates on one pixel in each scan line. The output data are then moved to registers and thence to digital-to-analog (D/A) converters which reconstruct an analog signal for output to a video display device.

Instruction Flow Within the Princeton Engine

In general all of the 1024 processors of the Princeton Engine execute the same instruction at the same time. Thus for the purposes of programming, the processor array may be modeled as if it were a single processor. Instructions for all the processors are stored in a single instruction store memory, and each instruction is sent in turn simultaneously to all processors. The instruction sequence is restarted at the beginning of each scan line.

All program development is done on the Apollo host system, including creating (writing) and compiling programs. After being compiled on the Apollo host, instructions (object code) are downloaded into the instruction store memory of the Princeton Engine and executed. As mentioned above, generally all processors execute the same instruction, however rudimentary program branching is possible by conditionally "locking" a subset of processors, forcing them to execute null operations, while the unlocked set continues execution of the main instruction stream.

Execution of different programs on different scan lines is also possible. For example, one program may execute during the first half of the frame or field, and a second program during the second half providing comparison viewing. As, another example, one program may execute during the visible portion of the picture and a second may operate during the vertical retrace interval. This process is discussed in more detail in Line Dependent Programming (LDP) below.

Real-Time Operation

In real-time operation, data are processed and output at the same rate as they are input. This imposes a limit on the number of instructions for each scan line because the processing time per scan line must not exceed the horizontal scan period. For NTSC this real-time instruction limit is approximately 910. For other video formats the real-time instruction limit may be calculated from the horizontal scan rate and the processor instruction clock of nearly 14.32 mega-instructions per second. For an HDTV standard, 1050 lines/frame, interlaced scan, 29.97 frames/second, the maximum number of instructions is 455 , i.e., 14,318,182/(29.97 x 1050).

Parallelism within the processor permits up to six processor operations to occur within one instruction. Processor operations include moving data between registers, accessing local memory, multiplying two operands, and performing arithmetic logic operations. Not all operations can be executed together within the same clock cycle, but significant reductions in the number of required instruction cycles can be achieved by efficient scheduling of operations.

Non-Real-Time Operation

For those video processing algorithms that exceed the real-time instruction limit, instructions may be included to store the incoming video data (at incoming video rates) into local processor memory. Once sufficient data have been accumulated (or the memory is full) processing of the stored data can be started. When complete, the processed data (still in local memory) are distributed to the output for reassembly into a continuous video stream for viewing as real-time video. This mode of operation is called video-clip processing.

The maximum length of a video clip is determined by the processor memory and the format of the video sequence to be stored. For example, each processor has 128 Kbytes of memory organized as 64 K of 16-bit words, with approximately 49 Kwords available for user storage. NTSC video has 525 lines per frame and a 1/29.97 second frame rate, thus requiring 525 x 29.97 = 15,734 pixels per second per processor. Packing two pixels into every 16-bit word, 50,176 words per processor provides up to 6.37 seconds of NTSC video storage.

More generally, non-real-time operation is possible with either video or non-video data. Integer arrays or fixed point real arrays may be stored in the local processor memory subject to the limits discussed above. The independent *instruction store memory* (common to all the processors) can hold up to 64 different programs each of which may be as long as 4096 instructions. By combining multiple programs so that they execute as one, a program of up to 262,144 instructions can be executed. This permits the execution of very long algorithms.

Advanced Features

In addition to "standard" video data flow, hardware has been included to:

- provide multiple viewable outputs on one viewing screen and/or multiple viewing screens for side-by-side comparison of algorithms,
- execute different programs on different scan lines, for comparison of multiple algorithms,
- transfer pixel data between processors,
- acquire portions of the output data in a capture memory for subsequent transmission back to the Apollo host,
- route selected digital output data back to the input for further processing.

Please refer to figure 3, a more detailed diagram of the Princeton Engine, for the following discussion of the advanced features.

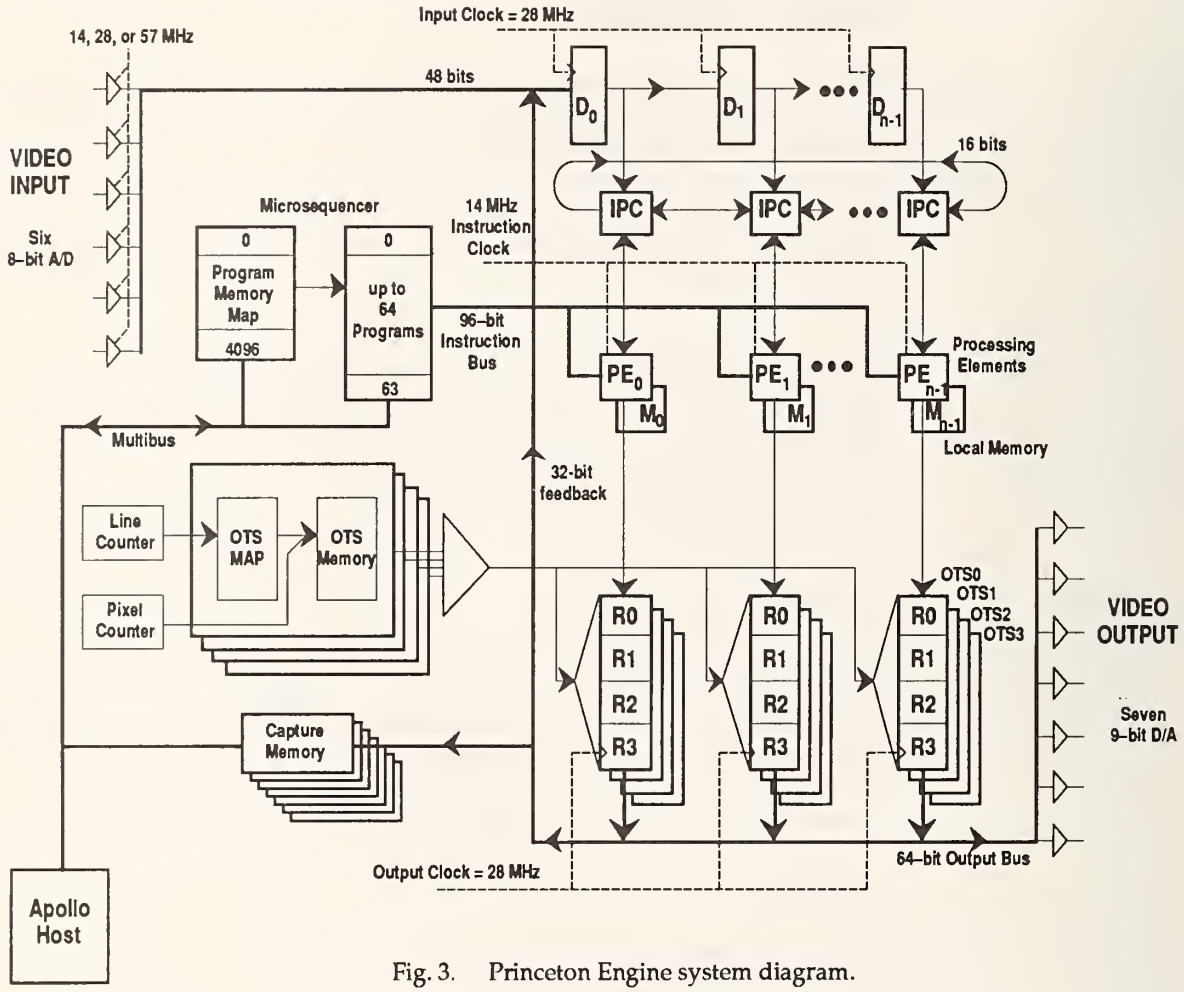


Fig. 3. Princeton Engine system diagram.

Comparison Viewing

A specialized output formatter, the Output Timing Sequence (OTS) facility, permits split images on the output video monitor where each image is derived from a different video signal. For example, two outputs could be displayed, each occupying a vertical stripe of width one-half of the total screen width. A typical use might be to compare the results of two algorithms; or with three stripes to display the two results and the difference between them. Different outputs could be assigned to different points along a circuit diagram to observe the progression of the signal through the processing chain. Up to four vertical stripes may be defined.

In addition to using the OTS to specify the formatting of the entire picture (i.e., a vertical stripe), several OTS patterns can be constructed and each "mapped" to operate on certain scan lines. This "line-dependent" OTS feature can be used to specify up to 16 OTS patterns per channel (64 total). The screen can thus be broken into a checkerboard of video outputs.

Line-Dependent Programming

The program memory map in the microsequencer allows the user to execute different programs during a single field or frame, as opposed to normal operation where the same program is executed for each scan line. This permits the user to compare the results of different programs for example by specifying program "A" for the top half of the screen and program "B" for the bottom half.

The advantage of line-dependent programming for comparison viewing of multiple algorithms in real-time is particularly apparent. It is possible to combine multiple algorithms into a single program and use OTS mapping to select outputs from the different algorithms for comparison viewing, but in this case the multiple algorithms must all run (sequentially) within a single scan line period. In LDP each program is executed independently for its particular scan line(s). Thus, (for real-time NTSC operation) *each* line-dependent program/algorithm is limited to 910 instructions, while with OTS, the *total* number of instructions for all algorithms combined must not exceed 910 instructions.

Up to 64 different programs, of up to 4096 instructions each, and a program sequence map can be downloaded into the microsequencer to specify which of the 64 programs is to be executed for each scan line.

Communication Between Processors

In the discussion so far, the data for each pixel on a scan line was sent to its corresponding processor; no data sharing or transfer between processors was attempted. However some applications will require that a processor have knowledge of data sent to, or computed by, a neighboring processor. The InterProcessor Communication (IPC) bus allows any data within a processor to be sent to another processor.

To use the IPC, data generated (or received) in a processor is loaded into the IPC bus register for that processor and an IPC bus transmit command is executed (by all the processors) to shift all the loaded data either left or right on the bus (multiple times if necessary) until they reach their destination

processors. Data at each end of the bus may be looped around to the processor at the other end of the bus (to the leftmost processor for a right shift, the rightmost processor for a left shift) or a constant user specified value may be shifted into the ends.

Also permitted is selective transmission and reception of shifted data by processors. Any processor may be excluded from exporting data to the IPC bus and/or receiving data. For example, data from every fourth processor may be sent to the three adjacent processors to its left (or right) or every second processor can send data to the second processor on its left, skipping its nearest neighbor. Finally, a single processor may be selected to broadcast to all other processors, or a subset of all other processors.

Feedback—Output-to-Input

A 32-bit wide digital path connects the final digital output of the Princeton Engine back to the input. This permits iterative processing of data, or comparison of processed data to incoming data. One possibility is to use OTS to map the feedback path to different processors. This mapping method may be more efficient than using multiple IPC shifts and/or broadcasts, which require one or more processor instruction cycles per shift.

Data Capture

It is also possible to "capture" a portion of the output data stream and upload it to the Apollo host workstation where it is stored as numerical data in a file. The user must specify (in advance, via a mapping file) which scan lines for which processors are to be captured. At present a maximum of 32 lines may be captured at one time.

The reverse of this process, that is, taking numerical data from the Apollo and downloading it into the Princeton Engine for processing is accomplished in a round about way. Directly dumping data from the Apollo to the Princeton Engine input is not practical. The data must be loaded into specific local processor memory locations before processing is started, and the Princeton Engine program must be written to expect the input data in the local processor memory rather than from the usual video source.

Programming the Princeton Engine

Programming Philosophy

Unlike conventional computers where a program is created as lines of text, programs for the Princeton Engine are created graphically. A "circuit diagram" is drawn to represent a video processing function the user wishes to simulate. Boxes represent modules of program code and connecting "wires" indicate data flow paths. Figure 4 is an example of such a circuit diagram or program. Here, a composite NTSC video signal is digitized by an analog-to-digital converter and then fed to a delay line (HDEL.M), adders, subtractors, dividers (DIV2.M), and filters (FIR7.M) to separate the luminance and chrominance components. Further manipulation by various modules produces the red, green, and blue video components which drive the three digital-to-analog converters. (The synchronization and timing circuits of the Princeton Engine cause this program to be run at the start of each scan line.)

This one circuit diagram represents the code for all 1024 processors as each processor executes the same instruction as its neighbor, but with different parts of the video signal as an input. This one processor model will be used nearly universally when discussing programming. (One exception to this model is the ability to conditionally prevent specific processors from executing instructions while normal program execution proceeds on the others, i.e., rudimentary branching.)

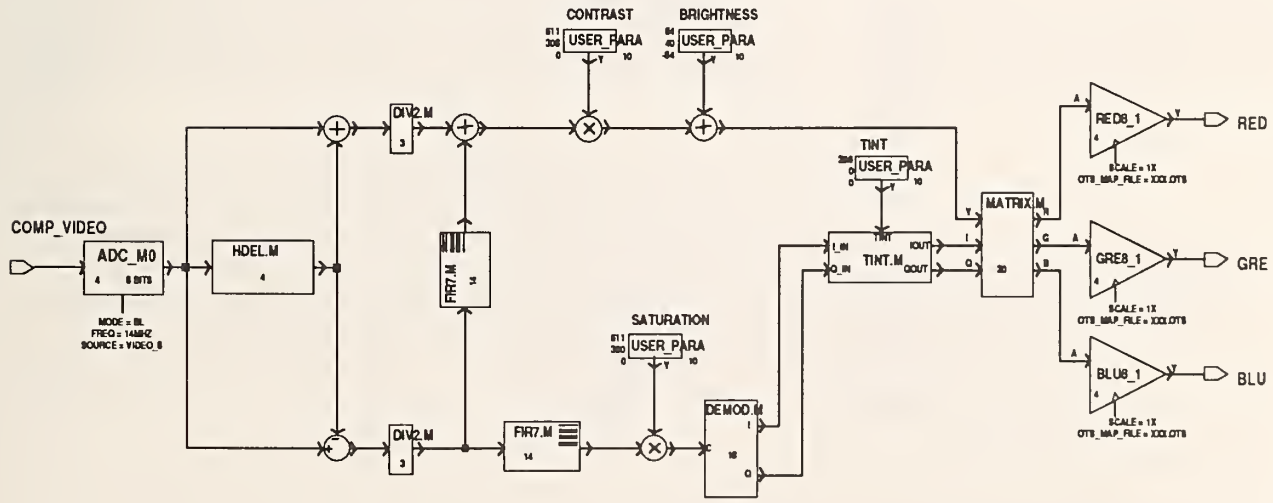


Fig. 4. A program for the Princeton Engine. This program decodes an NTSC composite video signal into red, green, and blue component video.

Many modules have been previously coded and are available in a user library (see Appendix A for a list of the available modules). When necessary, new modules may be created by the programmer. Code within the modules is based on the 16-bit arithmetical and logical computational abilities of the individual processors.

All programming, creation of modules and circuit diagrams, compiling, and linking, is done on an Apollo workstation and only the final machine code is downloaded to the Princeton Engine for execution. Although the Engine is a single user machine, multiple users may share its use by developing programs simultaneously on the Apollo workstations and running their code in turn. Video monitors are provided alongside all Apollo workstations for viewing the video outputs.

More traditional text-based compilers are under development, and may in the future augment or partially replace the programming tools available today. These compilers are discussed in the "Future Programming Languages" subsection on page 24. However, to appreciate the role these compilers will play in program development we suggest you read the "Programming Examples" section (below) first.

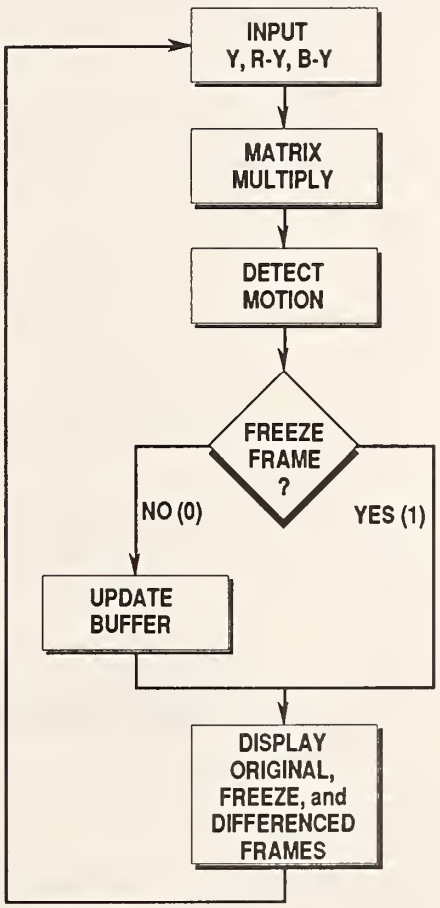
Programming Examples

The programming environment for the Princeton Engine is unusual in that it is based on a computer-aided-design tool—the Mentor Graphics CAD system for circuit diagram construction. This has the advantage of being a familiar environment for many electronics engineers, but computer scientists and other programmers may need to translate their traditional techniques to this new method.

As implied by the discussion in a previous sub-section, two levels of programming are available for the Princeton Engine. "High level" programming is the construction of the circuit diagram. In many cases all the necessary modules for the circuit have already been created and construction of the circuit diagram is all that is required. However, if some specialized modules are not available, "low level" assembly language programming will be required for creation of the modules. The two examples that follow illustrate these two programming processes.

Example 1 – Creating Circuit Diagrams

This first example demonstrates the construction of a circuit diagram. Figure 5 is a flow chart for the algorithm to be implemented. It processes a 3-component input signal in color-difference format (Y, R-Y, B-Y) and produces three 3-component outputs: the input converted to RGB format, a frame delayed or a still image (frame frozen) RGB output, and the difference between the first two outputs (the motion components).



- Input a 3-component video signal (Y, R-Y, B-Y).
- Matrix multiply the (Y, R-Y, B-Y) color difference vector, converting it to RGB format.
- Difference incoming frame with current frame buffer contents (last or frozen frame) to detect motion.
- Under run-time user control, either replace frame buffer contents with present input, or leave frame buffer unchanged (freeze previous image).
- Output three RGB video signals: the original signal, the frozen or delayed signal, and the difference or motion detected signal.
- Repeat for each frame.

Fig. 5. Flow chart for the example algorithm to be implemented on the Princeton Engine. A 3-component color difference video signal is processed to produce a 3-component RGB video output signal and to detect motion between video frames.

The NETED window environment is illustrated in figure 6. NETED is the NETwork EDitor of the Mentor Graphics CAD system, and is used for creation of all circuit diagrams/programs. Mouse controlled menus are used for window management, drawing, and editing functions. Most operations take place in the EDIT window where the circuit program is built from modules and interconnecting wires, or nets in the NETED teminology.

In figure 6 construction has been started on the motion detection and freeze frame circuit. A freeze frame and differencing sub-assembly has been created by selecting the FREEZE.M and SUB.M module symbols, one at a time, from the parts list (which automatically placed them into the active part window). From the active part window they were copied to the edit window and

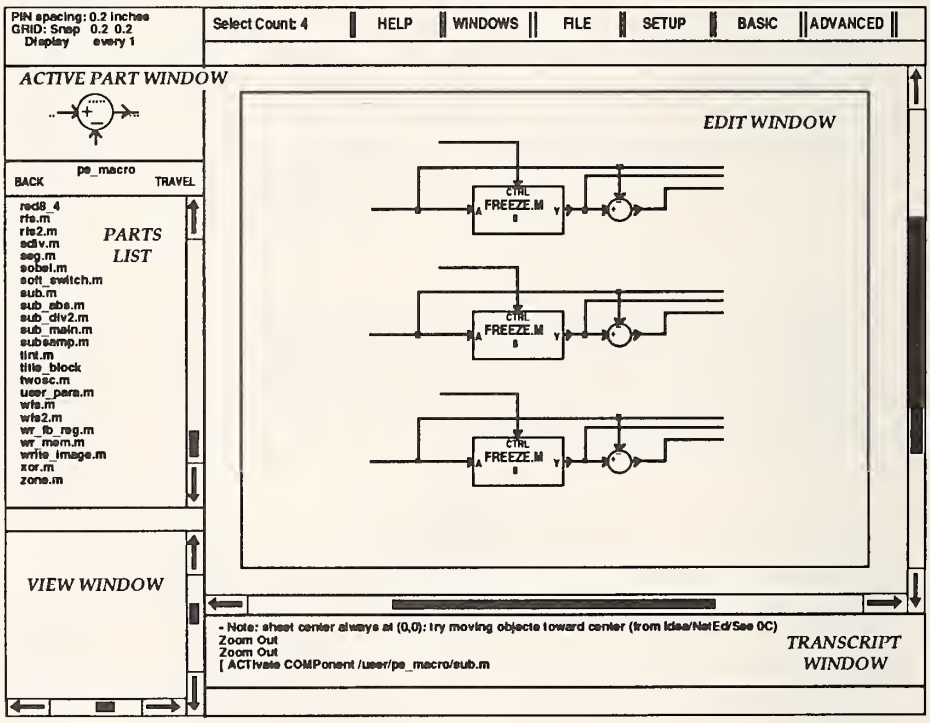


Fig. 6. NETED, the graphical circuit editor for developing programs for the Princeton Engine. Major constituents are the EDIT WINDOW (upper right) where drawing is done, the ACTIVE PART WINDOW (upper left) where parts are loaded from the PARTS LIST in preparation for copying to the edit window, the VIEW WINDOW for simultaneous viewing of a different part of the circuit, or a different circuit, and the TRANSCRIPT WINDOW which contains a historical list of the commands that have been executed.

placed in their desired locations. Interconnecting nets were then routed between the module pins. Finally, the subassembly was copied twice to produce the complete drawing shown in the figure. (As an example of the capabilities of the drawing program, note that the standard SUB.M module, shown in the active part window, has been flipped about its horizontal axis before being placed in its final position in the edit window.)

The circuit is completed by copying the necessary remaining modules into the edit window and drawing connecting nets. When the final wiring is complete, the design "syntax" is checked for disconnected or misconnected nets and if no errors are obtained the design is saved to disk. After construction with NETED, the design must be compiled and linked using the Graphical Program Composer (GPC). The resulting machine code may then be downloaded and run on the Princeton Engine.

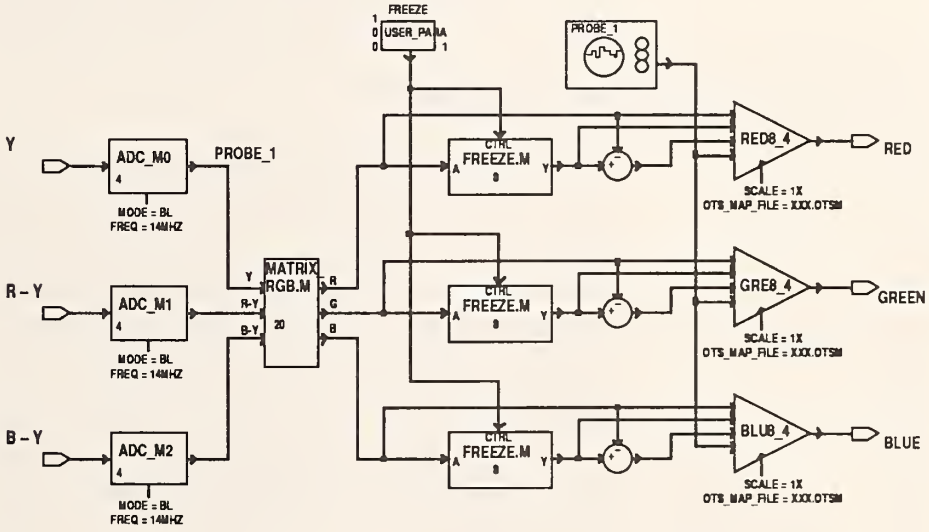


Fig. 7. The complete example circuit converts color-difference video signals to RGB. The circuit displays four outputs (as indicated by the 4 input signals to each DAC), the original video input, a frame delayed or a still image video signal, the difference between the delayed or still image signal and the original signal, and a probe output. The probe input can be temporarily attached to any net (wire) in the circuit for viewing the signal along that segment. (For this example the probe input has been attached to the output of ADC_M0 as indicated by the PROBE_1 label; routing wires are not used to indicate probe input connections.)

Example 2 – Creating New Modules

New modules may be needed when precoded modules are not available to do a specialized operation, or if it is desired to combine several modules into a single module to eliminate redundant instructions. New modules are created in a two-step procedure.

1. Create machine code using the Graphical Program Editor (GPE).

2. Generate a symbol to represent the code on a NETED schematic using the Mentor Graphics SYMbol EDitor (SYMED).

As a second exercise we examine an already coded module which has been developed using GPE. The module FREEZE.M has two inputs, A and CTRL, and one output Y. The purpose of the module is to freeze (or pass through) one video frame, input through A, and output to Y depending on the status of CTRL. If CTRL = 0 then A is passed through to Y delayed by one frame time and is simultaneously stored in a frame buffer in local processor memory. If input CTRL $\neq$ 0 then the last stored frame is output.

Creating code with GPE

The Graphical Programming Environment (GPE) is used to produce the machine-level code which makes up the low level modules in the programming hierarchy. Figure 8 shows the GPE programing environment with no instructions yet defined. As with NETED, because all the processors execute the same instruction, the entire processor array may be modeled as a single processor. GPE shows a representation of that processor on the screen, and its various components (registers, ALU, RAM access, etc.) can be interconnected by drawn lines. GPE shows three instructions simultaneously, the one being created or modified in the main or lower panel, the previous instruction in the upper-left panel, and the next instruction in the upper-right panel. (The previous and next instruction panels are blank here as we are showing how a new module is started and no instructions have been defined.)

Processor resources available to the programmer include:

- a 64-register register file (REG_FILE) for temporary storage,
- access to the interprocessor communication bus for shifting data to neighboring processors (LEFT, IPC_BUF, RIGHT, COM_REG),
- a 2-input arithmetic-logic unit (ALU) for arithmetic and logic operations,
- a 2-input multiplier (MPY, PP, and P) for multiplication and product bit extraction,
- access to RAM for local data storage and module input and output, and
- the use of intermediate registers IREG1, IREG2, DI1 for access to the register file (REG_FILE) and intermediate register DI for access to RAM.

Operations generally consist of moving data to and from the ALU or multiplier and the intermediate registers. To improve code readability, the registers may be labeled by the user to indicate their contents.

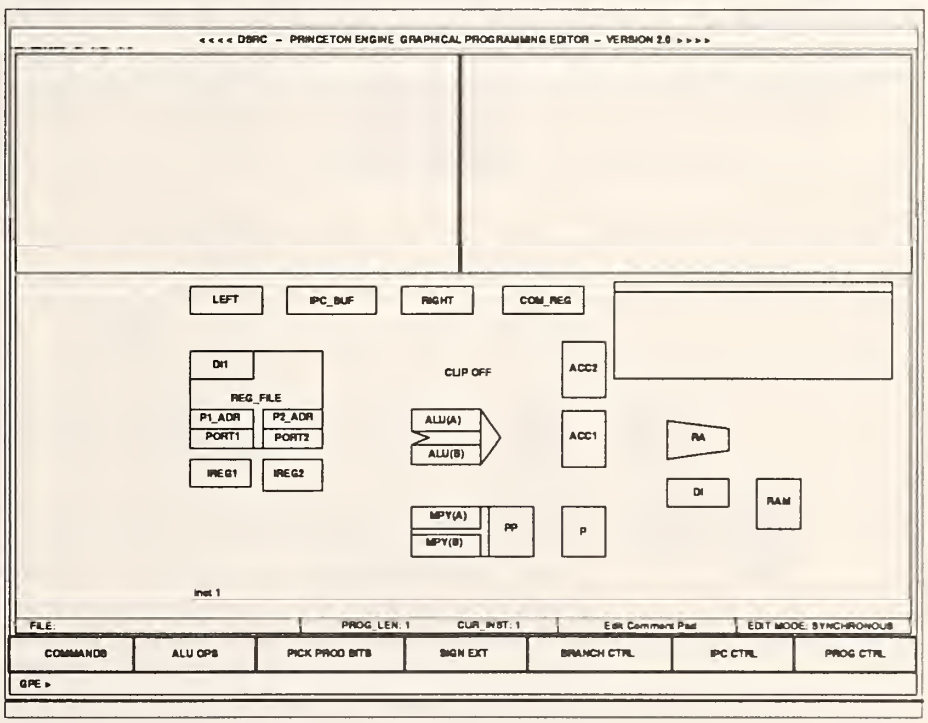


Fig. 8. Graphical Programming Editor showing the PE processor model and pop-up menu controls for selecting processor operations.

We will now briefly discuss the GPE program instructions for the FREEZE.M example. Figure 9 shows the first instruction of the program listing of the FREEZE.M module as produced by GPE. (The seven additional instructions for the module continue on the next several pages.) Data flow between registers is indicated by lines with arrowheads which are drawn between the source and destination register. We recommend that labels be assigned to the registers to indicate their contents. For example, note that the register indicated as IREG1 in figure 8 (its real name) is labeled 0 in figure 9 below, as it will contain zero after execution of the instruction. Not all registers can be directly interconnected, hardware restrictions prohibit connecting the register labeled LCM to accumulator ACC1, for example. Pop-up menus (not shown) are used to select ALU operations. A complete listing of all processor operations is included in Appendix B.

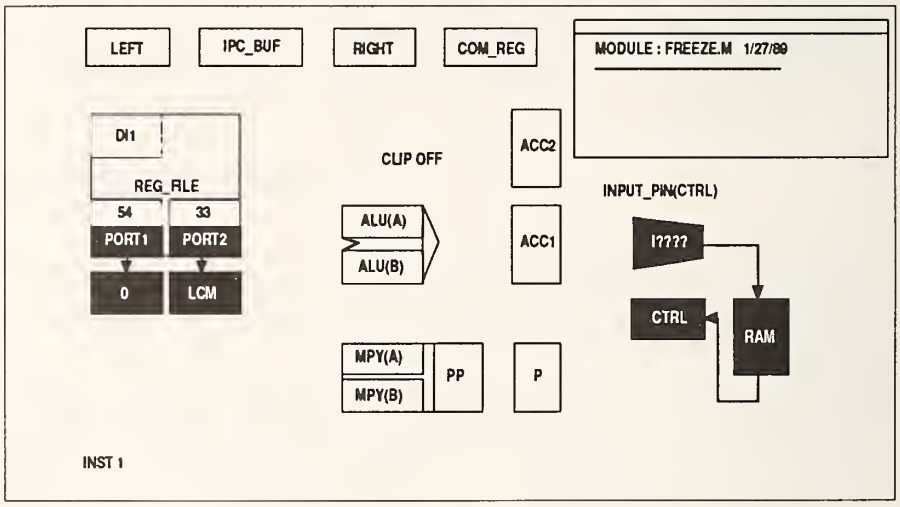
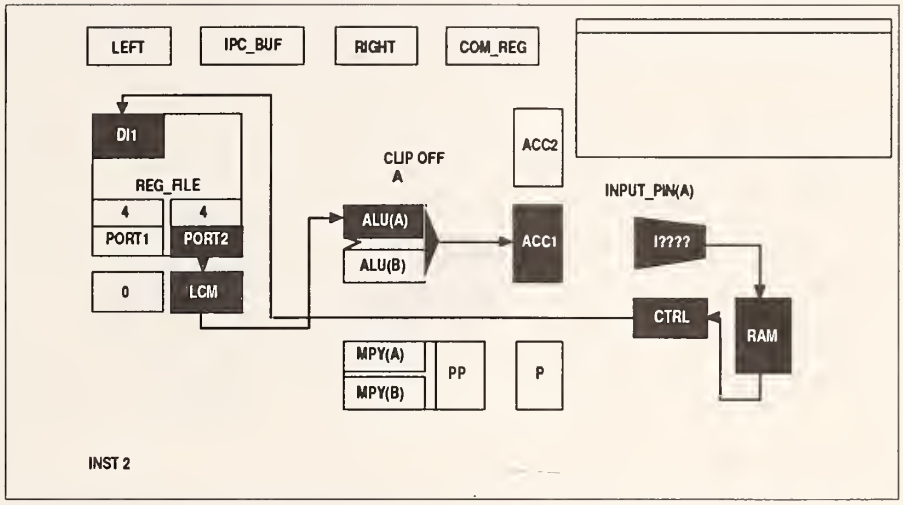
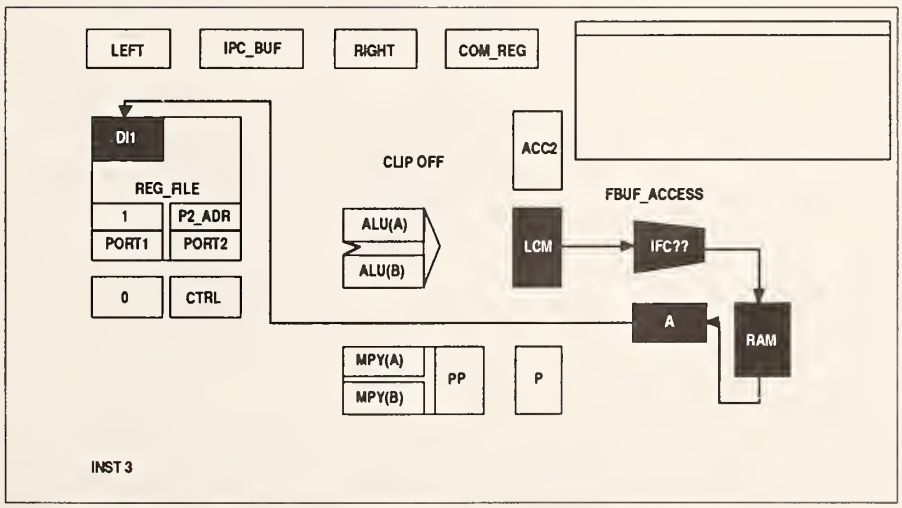


Fig. 9. The first GPE instruction for the FREEZE.M module. (Seven additional instructions continue on the next 4 pages.)

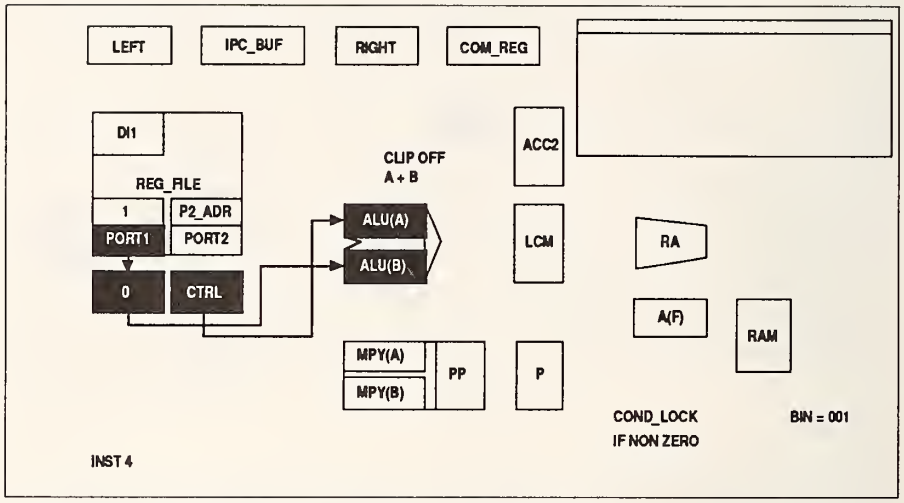
Instruction 1: Communication between modules is accomplished by storing output data in a defined location in local processor memory where the next module will be instructed to look for it. The program compiler (Graphical Program Composer, GPC) resolves these memory location definitions between modules; the programmer assigns an input or output to the appropriate pin name on the module symbol. For our example, input to the module is obtained through the input pins A, and CTRL. In instruction 1 input CTRL is loaded from RAM into the RAM-intermediate-register, here labeled "CTRL" to remind us of its contents. At the same time registers 54 and 33 of the register file are loaded into PORT1 and PORT2 intermediate registers labeled "0" and "LCM." Registers 54 and 33 contain predefined values of zero, and the current scan line number, LCM, respectively.



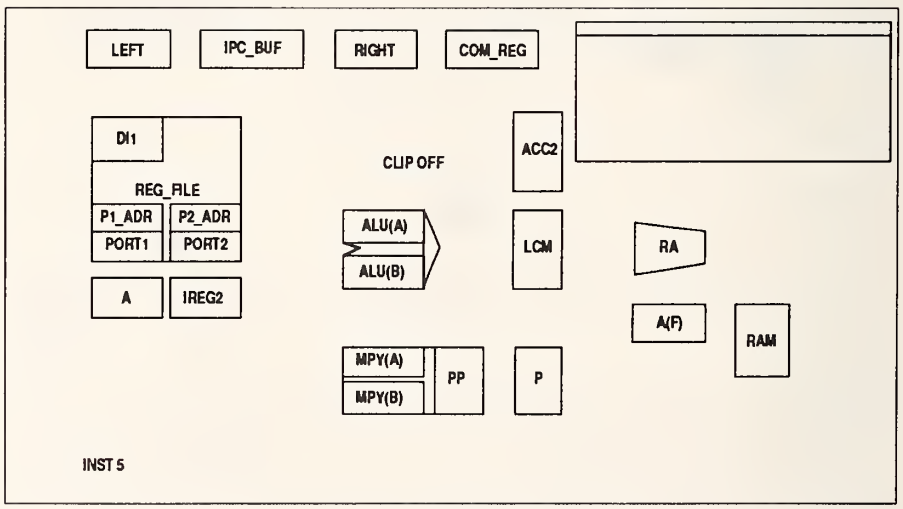
Instruction 2: LCM is moved from its PORT2 intermediate register, through the ALU (which is set to output ALU(A)) to accumulator ACC1. CTRL is moved from its RAM intermediate register through intermediate register DI1 into register 4 of the register file as designated by the 4 in the PORT1 address register and simultaneously into the PORT2 intermediate register previously occupied by LCM. Further, input A is accessed from memory and moved into the RAM intermediate register just vacated by CTRL. The updated contents of the registers will be available for the next instruction.



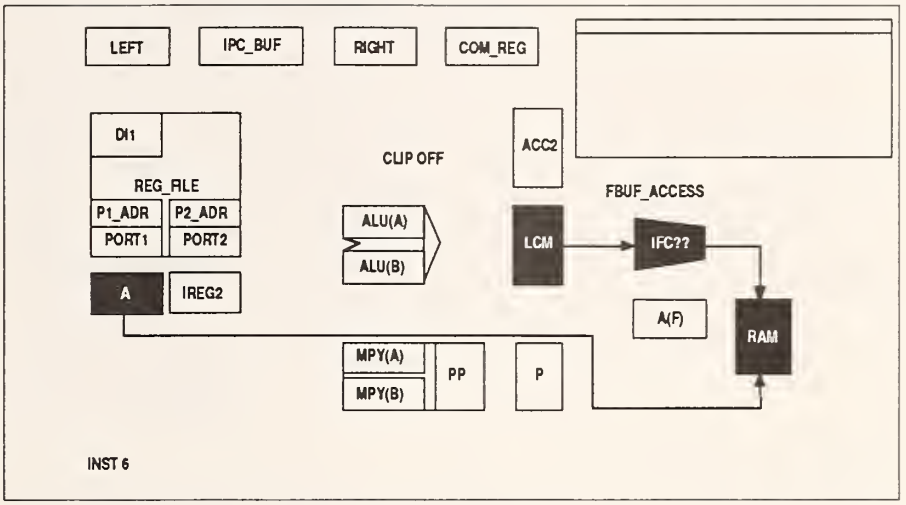
Instruction 3: Input A is moved through intermediate register DI1 to register 1 of the register file. The line counter, LCM, is used as an index to be added to the base address of the frame buffer, FBUF, located in local memory. The frame buffer contents are moved into the RAM intermediate register vacated by input A.



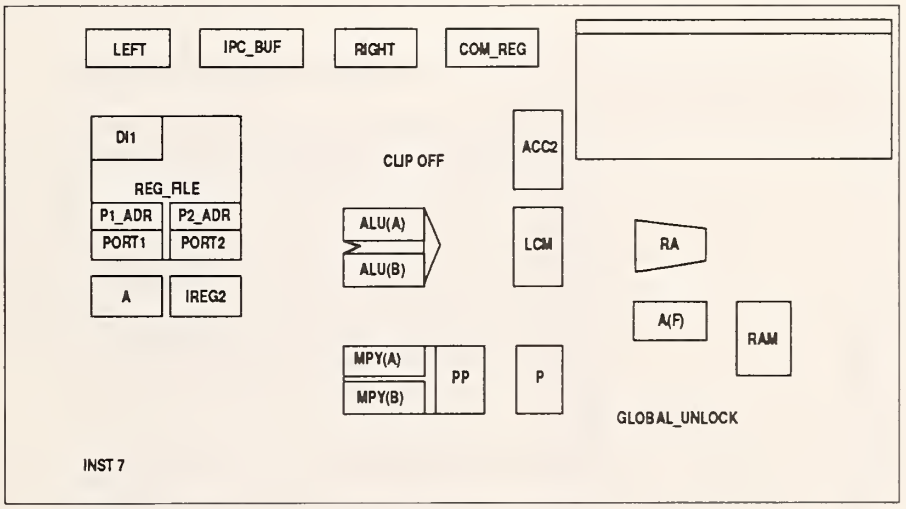
Instruction 4: The zero in the PORT1 intermediate register is moved to ALU(B), and the value of A in register 1 replaces it. CTRL from the PORT2 intermediate register is moved to ALU(A) and the two values are added and tested for equality to zero. This effectively tests whether CTRL is zero or not.



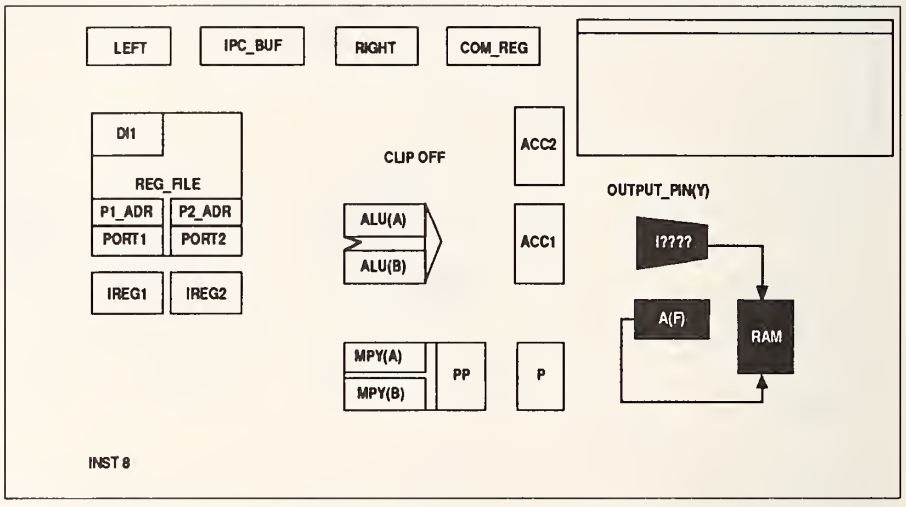
Instruction 5: If the test in the previous instruction was true, i.e., CTRL $\neq 0$, the processors are "locked" from executing further instructions. This produces a global locking (or not) of all processors since the test condition, CTRL $\neq 0$, produces the same results for all processors.



Instruction 6: This instruction is only executed if the processors were not locked in the previous instruction. If the processors are locked, this instruction will effectively be replaced by a no-operation. The current value of input A is stored in the frame buffer at index LCM overwriting the existing contents.



Instruction 7: Globally, unconditionally, unlock all processors.



Instruction 8: Move the old value of A obtained from the frame buffer, and previously stored in the RAM intermediate register, to output pin Y.

Note that if CTRL ≠ 0, then globally locking the processors prevents the frame buffer from being updated in instruction 6, thus the value of A that will be obtained from the frame buffer during the next cycle (in instruction 3) will be the last stored value, thus freezing the picture.

Generating a symbol

Symbol generation is done using the Mentor Graphics SYMED program. Generation is simplified however by the use of a symbol generation macro that interrogates the user about the number of input and output pins, their names, and their locations, and then draws an appropriate symbol part. The symbol (figure 10) is also labeled with the number of instructions in the module (8). In most cases this is all that is required. For specialized symbols, all the tools of the SYMED program are available to customize the size, shape, and other features of the symbol.

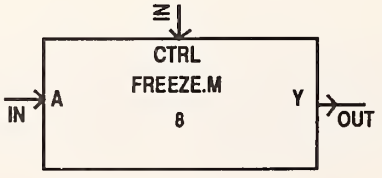


Fig. 10. NETED symbol for FREEZE.M module.

Controlling/Debugging a Running Program

Control of the Princeton Engine is accomplished by sending specialized commands (SPES commands) from the Apollo workstations. These 43 predefined SPES commands control all aspects of Princeton Engine operation, including loading programs, setting input and output configurations, changing program variables while the program is running, initializing or loading data into local processor memory, and capturing output data. Although the user can type the commands directly, two methods have been developed to make the system easier to use. A graphical control environment (GCE) program can be run on the Apollo to provide an interface between the user and the SPES commands, or NETED can be placed in a GCE mode to provide control over some operations.

The GCE display is shown in figure 11 with the applications menu pulled down. This configurable menu allows the user to execute a series of SPES commands (previously defined in a text file) to set the Princeton Engine environment and download an application in one operation. User parameters defined in the downloaded program will show in the boxes to the left. The present value of the parameter is shown in the box immediately below the parameter name, and the value is changed by clicking on the up or down arrows. Additional menus at the top of the display allow the user to conveniently execute some of the more common SPES commands.

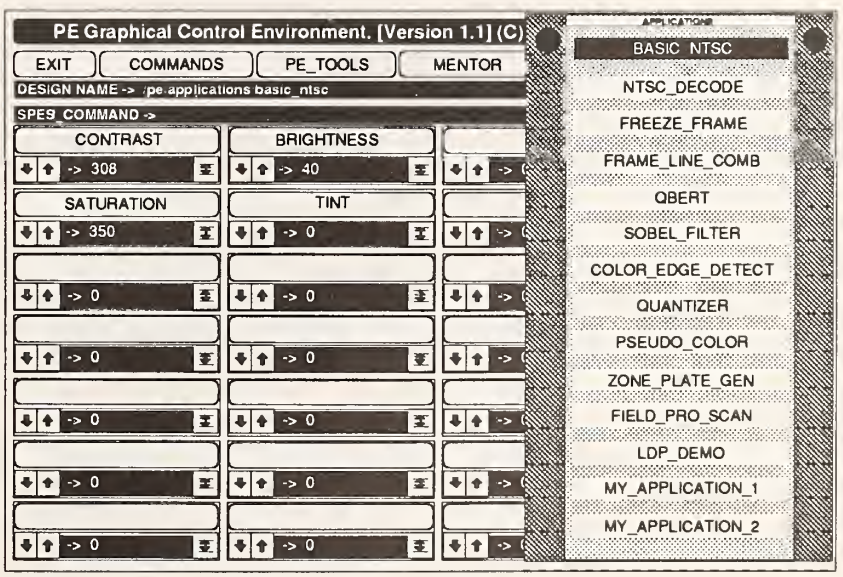


Fig. 11 Graphical Control Environment display with controls for modifying user parameters contrast, brightness, saturation, and tint of the BASIC_NTSC program shown in Fig. 4.

When NETED is used to control the Princeton Engine, it is placed in the GCE mode. Drawing operations are suspended, but revised menus are made accessible for downloading the program displayed in the NETED edit window, controlling the input and output registers, and changing user parameters and filter coefficients. The most important feature, however, is the ability to attach a moveable probe to different parts of the circuit and "view" the data at that point. To do this the outputs of up to 3 probes may be assigned to DAC inputs, and the actual probe-input position in the circuit is then assigned (or changed) at run time. (See figure 7 for an example of a circuit diagram with a probe.) Probing is one of the more powerful methods for debugging circuits.

Future Programming Languages

A C-compiler and a FORTRAN compiler are presently under development at the David Sarnoff Research Center. Although both compilers will be cross-compilers, i.e., they run on the Apollo workstations and produce code for the Princeton Engine, their functions will not be interchangeable.

At the present time the Graphical Program Editor (GPE) is the only tool available for developing assembly code for a module for the Princeton Engine. The Princeton Engine C-compiler (PEC) will implement a subset of the C language and eventually can replace GPE in the code development process. It is important to note that the PEC produces code for a *module* which then must be linked to other modules using a higher level programming environment such as NETED; one cannot develop a complete program using the initial release of the PEC. Initial testing of the C-compiler suggests that the code which it produces is nearly as efficient as hand-optimized code produced using GPE, moreover, program control functions such as "loops" may be used *only* via PEC. Delivery of the compiler is expected in the near future.

Alternatively, the Princeton Engine FORTRAN 90 compiler (which will implement a subset of FORTRAN 90) will be a substitute for NETED for the construction of a complete program. Preliminary results suggest that there will be a high overhead associated with the FORTRAN compiler.

The NIST Training Program

NIST will provide training in the use of the Princeton Engine for DARPA contractors and users from other collaborating organizations. This includes training for:

- the Apollo/Aegis operating system,
- Mentor Graphics CAPTURE schematic drawing software,
- using previously constructed library modules,
- construction of user-programmed modules,
- and using Princeton Engine-specific run-time operating software.

The more advanced features of the Princeton Engine (line-dependent programming, OTS mapping, and line-dependent OTS) will not normally be included in the training because they will not be needed by most users, are relatively complex, and require a thorough knowledge of the Princeton Engine hardware. (See the section on *The Princeton Engine – Advanced Features* for further detail about these topics.) Instead, NIST personnel will assist the user directly, providing specific solutions for the user's problem if the use of such advanced capabilities becomes necessary.

The training program consists primarily of self-directed study using reference material and workbook exercises provided by NIST. NIST experts will be on hand to answer questions or to explain difficult concepts. Sufficient student time will be made available on the Apollo workstations and the Princeton Engine for running and testing the workbook exercises or other problems the student may wish to try.

The training program is expected to take from 1 to 2 weeks to complete, depending on previous experience the student may have with the Aegis operating system or the Mentor Graphics CAD software. At the end of the program the student should have basic competency in developing programs for the Princeton Engine and running and debugging those programs.

NIST Contacts

For more information about the NIST laboratory or the Princeton Engine at NIST contact:

Dr. Bruce F. Field
(301) 975-4230, email: field@eeel.nist.gov
or
Dr. Charles Fenimore
(301) 975-2428, email: fenimore@eeel.nist.gov

National Institute of Standards and Technology
B344, Metrology Building
Gaithersburg, MD 20899

APPENDIX A

Module Library for the Princeton Engine

There are presently over 150 modules in the Princeton Engine Library. These modules are general purpose code elements that form the basis for developing Princeton Engine "programs" using the NETED circuit diagramming software. (See *Programming the Princeton Engine* for more detail about NETED). A list of the modules is presented below, categorized by module function.

Analog-to-Digital Conversion Modules (video input)

Several modules have been created to represent and control the analog-to-digital (ADC) hardware inputs. They convert an input analog video signal to a digital stream for processing by other modules. The output is represented in either two's-complement or binary format depending on the module used. Different A/D modules are also used to represent one of three main or three subchannels.

ADC_M0	8-bit ADC	main channel #0, two's-complement
ADC_M0B	8-bit ADC	main channel #0, binary format
ADC_M1	8-bit ADC	main channel #1, two's-complement
ADC_M2	8-bit ADC	main channel #2, two's-complement
ADC_S0	8-bit ADC	sub channel #0, two's-complement
ADC_S1	8-bit ADC	sub channel #1, two's-complement
ADC_S2	8-bit ADC	sub channel #2, two's-complement

Digital-to-Analog Conversion Modules (video output)

These modules are used to route the processed digital video signal from other modules to the output digital-to-analog (DAC) converters. Some modules include additional digital inputs (up to four) that are routed to additional OTS registers so that the separate video signals may be displayed in vertical stripes on the same monitor. (See *The Princeton Engine, Advanced Features* for additional information about OTS channel outputs.) All modules are two's-complement.

RED8_1	8-bit DAC	single-input red DAC
RED8_2	8-bit DAC	2-input red DAC
RED8_4	8-bit DAC	4-input red DAC
GRE8_1	8-bit DAC	single-input green DAC
GRE8_2	8-bit DAC	2-input green DAC
GRE8_4	8-bit DAC	4-input green DAC
BLU8_1	8-bit DAC	single-input blue DAC
BLU8_2	8-bit DAC	2-input blue DAC
BLU8_4	8-bit DAC	4-input blue DAC
DAC3.8_1	8-bit DAC	single-input DAC#3
DAC3.8_2	8-bit DAC	2-input DAC#3

DAC3.8_4	8-bit DAC	4-input DAC#3
DAC4.8_1	8-bit DAC	single-input DAC#4
DAC4.8_2	8-bit DAC	2-input DAC#4
DAC4.8_4	8-bit DAC	4-input DAC#4
DAC5.8_1	8-bit DAC	single-input DAC#5
DAC5.8_2	8-bit DAC	2-input DAC#5
DAC5.8_4	8-bit DAC	4-input DAC#5
DAC6.8_1	8-bit DAC	single-input DAC#6
DAC6.8_2	8-bit DAC	2-input DAC#6
DAC6.8_4	8-bit DAC	4-input DAC#6

Logical/Arithmetic Modules

These modules perform the indicated computation on one or more 16-bit two's complement inputs and produce a 16-bit output. Inputs are typically denoted by A, B, ... etc. (Exceptions are noted.)

ABS.M	| A |
ADD.M	A + B
ADD3.M	A + B + C
ADD_DIV2.M	(A + B) / 2
AND.M	Bitwise logical 'AND' of A and B
CLIP.M	Clip input-A to lie within inputs LOL and UPL.
COMP.M	Output 1 if input-A >= input-TH, 0 if A < TH.
CONST.M	Constant (user specified on NETED).
D1V2.M	$A / 2^N$ (N = 2 to 7).
- DIV128.M	
INV.M	Binary NOT(A)
LT1.M	A limited to N bits (N = 1 to 9).
- LT9.M	
MEDIAN3.M	Median of three inputs, A, B, C
MIN.M	Minimum of two inputs, A, B
MAX.M	Maximum of two inputs, A, B
MAX3.M	Maximum of N inputs (N = 3 to 7).
- MAX7.M	
MIXER.M	A x K + B x (1-K) (A, B, and K are inputs, K is an 8-bit input, 0 < K < 1).
MULT.M	$(A \times B) / 2^8$
MULT2.M	$A \times 2^N$ (N = 2 to 7).
- MULT128.M	
ONESC.M	one's complement(A)
OR.M	Bitwise logical 'OR' of A and B
PROC_NUM.M	Processor number (= 0 to 1023).
QUANT8.M	A quantized to 8 bits.
SDIV.M	Two outputs, Q = INT(A/B); R = Remainder(A/B)

SEG.M	If (input-A is between two inputs ST and END) then output = input-MAX else output = input-MIN
SUB.M	A - B
SUB_DIV2.M	(A - B) / 2
TWOSC.M	two's complement(A) (A is one's complement).
XOR.M	Bitwise logical exclusive 'OR' of A and B

Control Structures

Branching and looping are presently supported only by forcing selected processors to execute NOPs (no operations) while other processors continue to execute the instruction stream.

BRANCH_TEST.M	Branch test module is an example of this conditional execution.
MUX2.M	2-input multiplexer, one of two inputs selected based on third input CNTL = 0 or 1.
MUX4.M	4-input multiplexer, one of four inputs selected based on third input CNTL = 0, 1, 2, or 3.
SOFT_SWITCH.M	Effective dissolve between 2 inputs A and B. Four inputs and one table are required, A, B are video inputs K, and TH are control inputs, and table T1 is the dissolve mapping function.

Control of Interprocessor Communication Operations

BC1.M - BC5.M	Input broadcast to other processors according to broadcast pattern BC1. Modules provide from one to five wait instructions.
BP.M	Configures the IPC circuitry to bypass processors according to a pattern defined at compile time.
CLEAR_IPC.M	Clear Interprocessor Communication circuitry erasing any previously loaded broadcast or bypass pattern.
IPC_LS.M - IPC_LS3.M	Interprocessor left shift N times. (N = 1 to 3).
IPC_RS.M - IPC_RS3.M	Interprocessor right shift N times. (N = 1 to 3).

Filters

A number of finite impulse response filters for spatial and temporal filtering are included. Initial values for the filter coefficients are specified while creating the circuit using NETED but they may be updated later during run-time.

FIRXX_YY.M is a generic two-dimensional filter with the following naming convention:

XX = the horizontal filter length, and
YY = the vertical (temporal) filter length.

The internal accuracy of these filters is limited to 8 bits.

FIR00_03.M, FIR00_05.M, FIR00_07.M, FIR00_09.M, FIR03_00.M, FIR03_03.M, FIR03_05.M, FIR05_00.M, FIR05_05.M, FIR07_00.M, FIR07_07.M, FIR09_00.M, FIR09_09.M

FIR3.M, FIR7.M, FIR9.M	Horizontal 8-bit filters with 3, 7, and 9 taps respectively.
FIR16_39_00.M	A two-dimensional 16-bit accuracy filter, horizontal filter length = 39, vertical filter length = 0.

Delay Modules and Local Processor Memory Operations

FRAME_BUF.M	Output is frame delayed version of input. The frame time is defined by the video input format.
FRAME_BUF2.M	Two outputs are frame delayed versions of inputs A and B. (Frame size is defined by the video input format.)
FREEZE.M	Output frame delayed version of input-A if input-CTRL = 0, if CTRL = 1 output previously stored frame.
FREEZE2.M	Output frame delayed versions of inputs A and B if input-CTRL = 0, if CTRL = 1 output previously stored frame.
HDEL.M	Input delayed by one scan line.
HDEL01.M - HDEL07.M	Each module produces multiple outputs (Y01 ... Y0N, N = 1 to 7), delayed versions of input by N scan lines.
RD_MEM.M	MEM_LOC(3800 + input-OFF) Read (and output) a memory location in local processor memory specified by address input-OFF (relative to 3800 HEX)

RD_FB_REG.M	Read a memory location in local processor memory specified by a compile time address. Address also provided to output R_AD.
RD_IFRAME_STORE.M	Read input-A into frame buffer specified by input-ST_R.
READ_IMAGE.M	Output data from frame buffer specified by input-ST_ADRS.
RFS.M	Output data for global (to all modules) frame buffer specified by input-ST_R.
RFS2.M	Two simultaneous outputs from double global frame buffer specified by input-ST_R.
WFS.M	Write input data to global frame buffer specified by input-ST_R.
WFS2.M	Write input data to double global frame buffer specified by input-ST_R.
WR_FB_REG.M	Output data from local processor memory from address specified by input-R_AD. Write input-A to a memory location R_AD.
WR_MEM.M	MEM_LOC(3800 + input-OFF) Write input to a memory location in local processor memory specified by address input-OFF (relative to 3800 HEX).

Video Controls and NTSC Specific Modules

BRCT.M	Modify video input by brightness and contrast values.
CBS_AT.M	Given Y and C inputs, apply brightness, contrast, and saturation values, and separate into RGB components.
CTBR.M	Modify input by contrast and brightness values.
DEMOD.M	Chroma demodulator separates the NTSC chroma (C) signal into two components (I and Q).
DEMOD_SUB.M	Chroma demodulator for the sub channel.
FCOMB.M	Separates composite NTSC signal into luminance output and chrominance output using a frame comb.
FLD262.M	Output F262 is input A delayed by 262 horizontal scan lines.
FLD_DELAY.M	Outputs D262, D263, and D264 are input A delayed by 262, 263, and 264 horizontal scan lines respectively.
FLD_SWITCH.M	If NTSC field is even then output=0 else output=1.
FRAME_COUNT.M	Outputs a count that increments on the first line of every NTSC frame (every 525 lines).

FRMFLD_BUF.M	Provides two outputs, a 262 line delayed, and a frame delayed version of the input.
FRMFLD2_2.M	Provides 262, 263, and 525 line delayed outputs for two inputs.
G525.M	Outputs 0 to 524, a line counter.
MATRIX.M	Converts Y, I, and Q into R, G, and B using standard NTSC weighting.
PROBE_1 - PROBE_3	Assigns probe channels to DAC ports. Probes allow internal signals on the NETED circuit diagram to be displayed on the video monitors.
PROBE_1S - PROBE_3S	Assigns probe channels to DAC ports for sub channels.
SUB_MAIN.M	Synchronize timing of subchannel to main channel.
TINT.M	IOUT and QOUT are the phase rotated versions of the quadrature inputs I_IN and Q_IN.

Miscellaneous Modules

INPUT_CONTROL	Control variable set by GCE during run-time (to select different algorithms for example).
EXT_FB0_IN	Input from external feedback channel 0
EXT_FB1_IN	Input from external feedback channel 1
EXT_FB2_IN	Input from external feedback channel 2
EXT_FB3_IN	Input from external feedback channel 3
EXT_FB0_OUT	Send input to external feedback channel 0.
EXT_FB1_OUT	Send input to external feedback channel 1.
EXT_FB2_OUT	Send input to external feedback channel 2.
EXT_FB3_OUT	Send input to external feedback channel 3.
LUT8.M - LUT10.M	Output value from lookup table using address input. Pathname of lookup table specified using NETED. (Module number specifies number of address bits of lookup table.)
SUBSAMP.M	Subsample input, output = input-IN AND MEM_LOC(input-OFF + 3800).
USER_PARA.M	Output user parameter to circuit. Parameter appears in GCE control environment for user modification at run time.

Larger Demonstration Modules

QBERT.M	Adaptive Line Comb NTSC Decoder
SOBEL	Sobel Edge Detection Module
ZONE.M	Zone Plate Test Pattern Generator

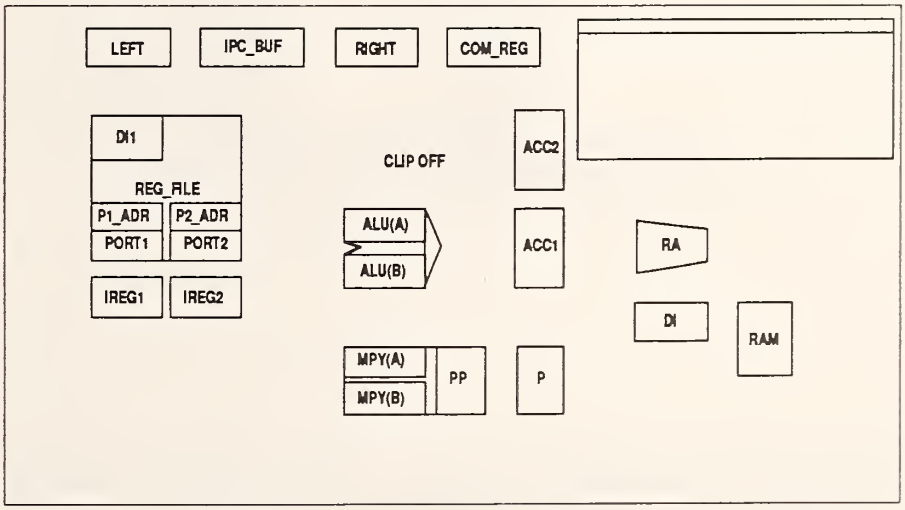
Processor Operations

Processor operations consist of moving data between the intermediate registers, the arithmetic logic unit (inputs ALU(A) and ALU(B)), the multiplier (inputs MPY(A) and MPY(B)), and the interprocessor communication bus (IPC_BUF).

The ALU performs arithmetic, logical, and functional operations on 16-bit data with its output routed to one or both of the output accumulators, ACC1 or ACC2 depending on the operation.

The multiplier operates on two 16-bit two's complement inputs routed to MPY(A) and MPY(B) producing a 32-bit intermediate value. The Product Picker (PP) allows the user to select 16 contiguous bits of the 32-bit product for placement into the output register P. The product picker effectively provides division by powers of 2 and can facilitate fixed point arithmetic. The figure below is a representation of the processor.

The interprocessor communication bus is used to transfer data between processors. Within a processor, data is routed to the IPC_BUF register before the transfer and the data received from a second processor is routed from the IPC_BUF register after the transfer is complete. The LEFT and RIGHT boxes on the diagram serve to initiate shifting operations. More complicated transfer patterns are invoked using the COM_REG.



GPE processor model which includes an ALU, hardware Multiplier, 64-register register file, RAM access, and IPC access.

APPENDIX B – Processor Operations

ALU Operations

CLIP ON	Prevents overflow by limiting ALU output to 7FFF or 8000 HEX.
CLIP OFF	No overflow correction is performed.
A + B	Add inputs A and B.
A + B + C	Add inputs A and B with carry from previous operation.
A – B	Subtract B from A.
B – A	Subtract A from B.
A – B + C	Subtract B from A with borrow from previous operation.
B – A + C	Subtract A from B with borrow from previous operation.
A + B + 1	Add inputs A, B, and 1.
CON[A – B]	Conditional subtract. If (A – B) ≥ 0 result is A – B, otherwise result is A.
A OR B	Bitwise logical OR of A and B.
A AND B	Bitwise logical AND of A and B.
A XOR B	Bitwise logical XOR of A and B.
ABS(A)	Absolute value of input A.
A	Route ALU input A through ALU to ACC1.
B	Route ALU input B through ALU to ACC2.
1SC(A)	Convert two's complement to one's complement.
2SC(A)	Convert one's complement to two's complement.
MAX(A, B)	Maximum of A and B.
MIN(A, B)	Minimum of A and B.
ST_DIV	Start software divide of A/B.
CONT_DIV	Continue software divide (one instruction per bit).
END_DIV	End software divide with quotient and remainder.
PACK	Pack lower 8 bits of A and B into 16-bit result.

APPENDIX C – Technical Paper

The Princeton Engine: A Real–Time Video System Simulator

D. Chin
J. Passe
F. Bernard
H. Taylor
S. Knight

IEEE TRANSACTIONS ON CONSUMER ELECTRONICS
Vol 34, No. 2, pp. 285-297, May 1988

NOTE: A few technical details in "The Princeton Engine: A Real–Time Video System Simulator" may disagree with material presented in the body of this technical note. This is a result of upgrades and improvements made to the Princeton Engine after publication of the paper. In the event of a disagreement it is generally safe to assume that the information in the body of the technical note is correct.

THE PRINCETON ENGINE: A REAL-TIME VIDEO SYSTEM SIMULATOR

D. Chin, J. Passe, F. Bernard, H. Taylor, S. Knight
David Sarnoff Research Center
CN 5300, Princeton, NJ 08543-5300
(609) 734-2301 TELEX: (609) 734-2221

Abstract

The Princeton Engine is a 29.3 GIPS image processing system capable of simulating video rate signals - including NTSC and HDTV video - in real-time. It consists of a massively-parallel arrangement of up to 2048 processing elements. Each processing element contains a 16-bit arithmetic unit, multiplier, a 64-word triple-port register stack (one write, two read), and 16,000 words of local processor memory. In addition, an interprocessor communication bus (IPC) permits exchanges of data between neighboring processors during one instruction cycle. We further describe a new method of parallel programming for DSP applications and provide several examples.

Introduction

The design of an NTSC digital television has required extensive computer simulations to verify digital signal processing algorithms. High-level language programs have been used to simulate a few fields of the target video system [1, 2]. New signal formats such as Advanced Compatible TV (ACTV) [3] also require significant manpower and simulation time to obtain acceptable results. While these simulations are important to the design process, they provide limited information about the performance of the actual system under real-time conditions. This has resulted in a costly development cycle in which hardware prototypes are built for each of several generations of experimental systems.

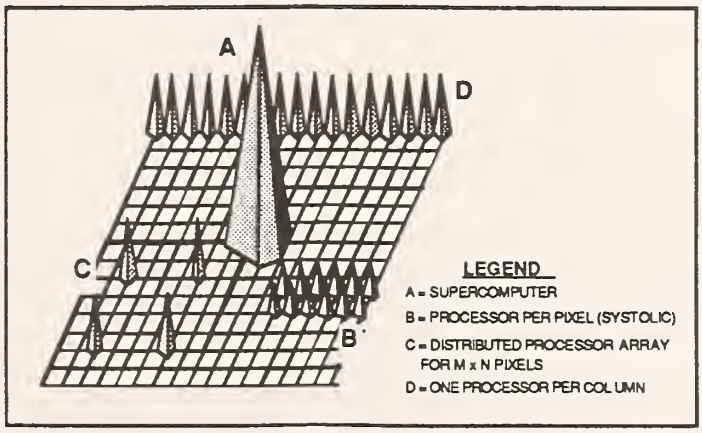


Figure 1.
Processor To Pixel Mapping Schemes.

The problem of performing true, real-time *video* simulations can be characterized in the following terms: the algorithms necessary to implement an advanced, motion adaptive, NTSC decoder requires about 1400 algorithmic steps *per pixel*. If each pixel is clocked at 14MHz (70ns cycle) rate, a single processor would have to be able to execute one algorithmic step every 20 picoseconds to sustain real-time operation. This is about two orders of magnitude greater than the *next generation* of supercomputers.[4] In addition to the intensive computational requirements, a real-time video simulation system must be able to continuously sustain I/O at 14MHz or better.

Numerous attempts have been made at applying supercomputer or multiple processor architectures to image processing and real-time video simulation problems. Figure 1 compares the different approaches in terms of processor topology - how they are mapped onto an array of pixels. The first approach ("A" in Figure 1) employs a single, very high performance computational node or several nodes such as a Cray X-MP. In the Connection Machine [5] system, 64,000 sequential single bit processors operate in a Single Instruction Multiple Data (SIMD) mode. Pixel data is mapped in a processor per pixel mode, as shown in "B" in Figure 1, for the entire array of pixels. This method is also referred to as fine grain parallel processing, wherein many simple processors are used to simultaneously perform the same computation on a large array of data [6].

An alternative to fine grain architectures for a multiple processor system is the course grained approach in which considerably fewer processors of greater computational power are employed. One such system from NHK [7], uses standard bit-slice processors as processing elements. Up to eight 16-bit processing units running at 7.16MHz and connected unidirectionally have been implemented in this system. Each processor contains a replication of a full frame of image data in memory, elim-

Contributed Paper
Manuscript received March 17, 1988.

0098 3063/88/0200 0285$01.00 © 1988 IEEE

inating memory access contention. As shown in approach "C" in Figure 1, each processor executes the algorithm for a specific region of pixels in the image plane. This system has been used for real-time frame synchronizing and mixing.

Another mapping of processor topology to an array of pixels is the Scan Line Array of Processors approach, or SLAP [8]. In this system, a linear array of identical processors are connected in a nearest neighbor fashion and are operated in an SIMD mode (as shown in "D" of Figure 1). Each processor contains an integer arithmetic unit, register file and a single stage of a shift register. As an image scan line is loaded into the array, each processor latches one pixel. Algorithm execution proceeds with processors in parallel. A 512 processor SLAP implementation with a 250ns instruction cycle time can perform about 125 real time instructions for each pixel of a 512x512 image. A more recent implementation of SLAP will yield about 500 instructions [9]; however, extended instructions, such as multiply, will take as many as ten instructions. Thus, real-time simulations of large video systems are not possible.

Several architectures have been proposed which combine features of both coarse and fine grain processing. One such architecture is the Warp Computer, a linear systolic array of processor cells [11]. Each processor cell contains a 32-bit multiplier and ALU unit capable of sustaining 10 Mflops. I/O between processor cells occurs at 20MHz, and the combination of high, single processor performance and I/O bandwidth are claimed to make both fine and coarse grain processing possible. However, real-time video and image processing simulations are greatly limited by a small number of processing cells (see "B" in Figure 1) and the need to interface an I/O frame store buffer. Temporal processing across multiple fields is limited by a mapping scheme where each processor cell must act as a fine grain processing element for a large number of pixels.

The Princeton Engine combines features of both coarse and fine grain processing architectures. It is implemented from a large number (up to 2048) of high speed (14 MHz), single cycle, 16-bit processors. Processing of video is performed in a processor per column architecture, similar to the SLAP system (see "D" in Figure 1). The processors are tightly coupled by a communication network which supports nearest neighbor exchanges of data in a single cycle and *random* exchanges between any two processors in a 64 processor boundary in one cycle. A local memory with sufficient storage for 32 frames of video data makes it possible to implement both temporal and vertical algorithms.

System Operation Overview

The initial application for the Princeton Engine will be in performing real-time video simulations of NTSC and

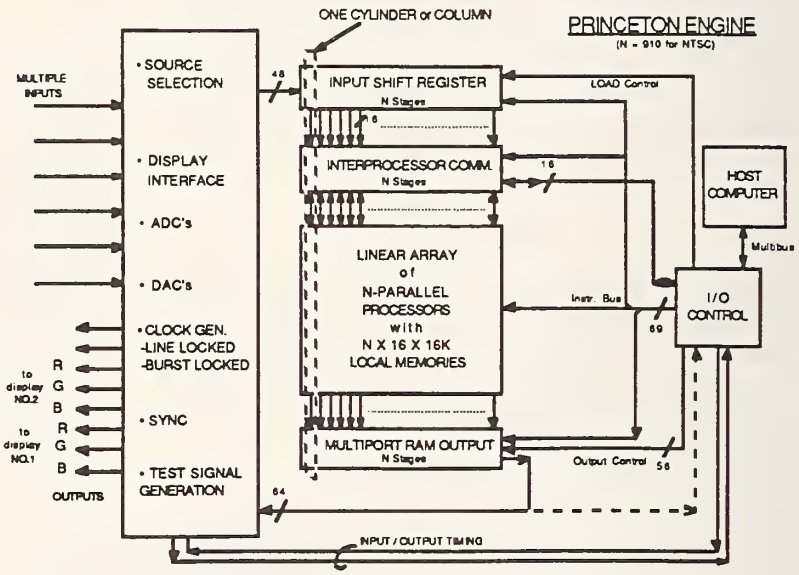


Figure 2. Princeton Engine System Overview

ACTV video systems. In the Princeton Engine system, digitized video is continuously shifted into the input shift register (top of Figure 2). After a line of video is completely loaded into the registers, the video is transferred in parallel to the interprocessor communication (IPC) buffers. The processor can then fetch the data and operate on it locally or globally. After all pixels are processed, they are downloaded to the output section within a line time. In this way, continuous video can be produced at the output. The processing should be completed in less than a line time to achieve real-time simulation. For four times subcarrier sampled NTSC data, the maximum number of instructions for real-time simulation, N_{RE}, is:

$$\frac{\text{input parallel load period}}{\text{instruction period}} = \frac{910 \times 70\text{ns}}{70\text{ns}} = 910 \text{ instructions}$$

for a 910 processor system. This is the total number of instructions available to accomplish the 1400 algorithmic steps required to implement the example NTSC system. In the Princeton Engine, each processor micro-instruction can typically achieve three algorithmic steps yielding a total capability of 2730 algorithmic steps. By doubling the number of processors to 1820, N_{RE}, is:

$$\frac{2 \times 910 \times 70\text{ns}}{70\text{ns}} = 1820 \text{ instructions,}$$

for a total of 5460 algorithmic steps.

The Princeton Engine achieves a linear speedup in the number of instructions which can be executed while still maintaining real-time operation. Likewise, this same speedup can be realized by halving the instruction period.

Engine Controller Input and Output

All system program control, video input and output is contained in the controller. A Video Input section which allows source signal selection and data conversion (A/D and D/A converter) is shown at left in Figure 2. Up to 48 total bits of video input source with three independent clocks can be processed (six 8 bit sources, three 16-bit sources, etc). A multibus interface between the host computer and the controller permits program and control code to be downloaded. Within the controller, there is a 16,000 instruction memory (word length is 89 bits). The controller transfers instructions from this program memory by sending a stream of identical instructions to each of the processing elements in the array. Video data is transferred in parallel to the array of processors - one line of video at a time - each processor receiving a single sample. The overall transfer rate of the input video data is 1.3 Gbit/second (48 x 28.64MHz). The controller also includes a video output timing and display section. In this section, a stream of pixels, 64 bits wide (i.e, eight 8 bit channels, four 16-bit channels, etc.), is transferred from the array of processors back to the output and display section at 1.8 Gbit/second (64 x 28.64MHz). The arrangement of pixels at output is completely programmable via the Output Timing Sequence bus (OTS) within the Graphical Control Environment (GCE) and occurs in parallel with instruction execution.

The Processing Element

The engine core consists of an array of up to 2048 processing elements. Figure 3 shows a block diagram of the processing element. Each element has a 16-bit ALU, a 16-bit Multiplier, a 64-word triple-port register stack and a 16-bit address/data external memory interface. In addition, processing elements are connected at the chip level via a 16-bit programmable data bus (IPC), which supports rapid exchanges of data between processors. A full compliment of processors can realize a throughput of 29.3 GIPS (2048 x 14.32MHz).

The processing element also contains special hardware support for maintaining lookup tables in external memory. External addressing can be of either an absolute, an indirect or a table index type. During each instruction cycle memory addresses can be directly encoded into the immediate field of the instruction. Next, the address can be generated indirectly from a register or the accumulator; finally, the address can be formed by combining a

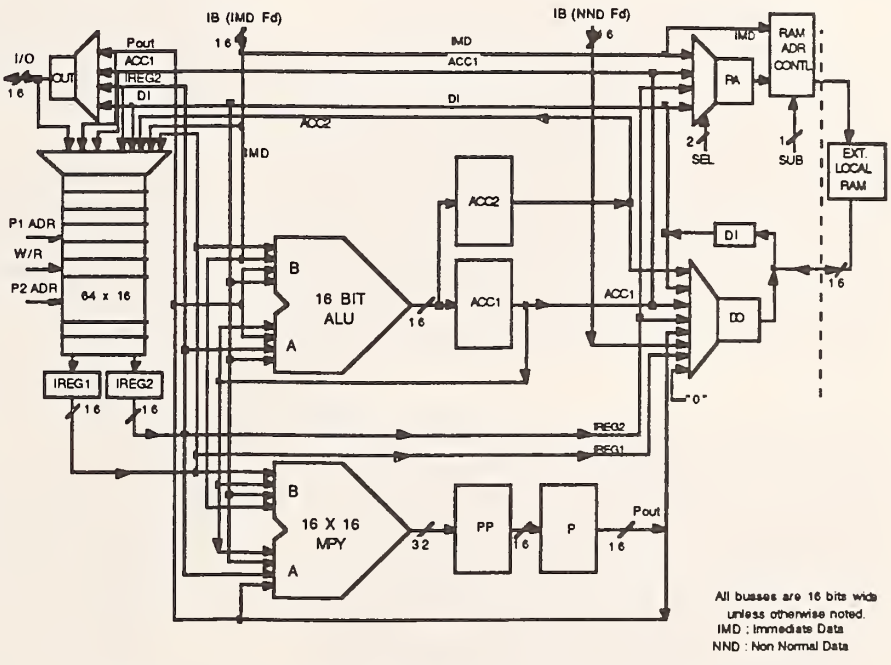


Figure 3. Processing Element Block Diagram.

register value with immediate data to form a table index. In table index mode, 8 MSB bits of immediate data provide a bit mask index. This mask determines which bits of the lower eight bits form the table address. Memory access is completed in one cycle. The immediate field of any instruction can also be used to load a data constant into the ALU, Multiplier or on-chip memory stack.

Total throughput of the system is significantly increased by the incorporation of multiple internal data paths within the processing element. This permits a high degree of secondary parallelism in program operation. For example, in filter operations, a pixel can be shifted left, while simultaneously, an ALU operation, a multiply operation, and an external memory access are being performed. The use of secondary parallelism in this way results in at least a 3:1 reduction in the number of instructions when compared to the number required on a conventional microprocessor.

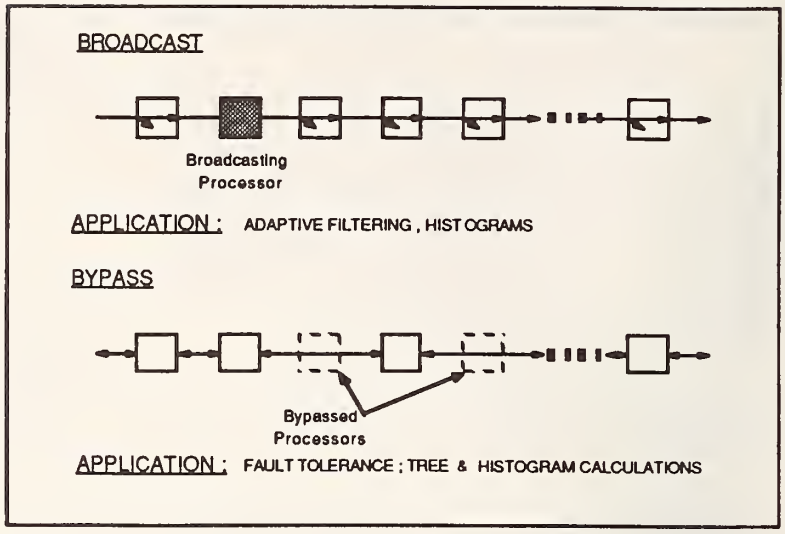


Figure 4. Broadcast and Bypass IPC Modes.

Interprocessor Communication Bus

The Interprocessor Communication Bus (IPC) provides high speed exchanges of data between processing elements and the video input/output processing logic. IPC bus operations can be of a broadcast type (one processor to many) or of a bypass type (where there are random length non-overlapping bidirectional connections between processors). A new IPC bus topology for the entire engine can be generated in two instructions. Figure 4 illustrates these two communication schemes and their applications.

In the Broadcast mode (top of Figure 4), one processor is designated as the sender and as many of the other processors as are required by the algorithm can be designated as receivers. Data transfer is accomplished in one instruction to any of the processors within the transmitter's 64-processor boundary. In the worst case (assuming 2048 total processors), it would take five instructions from any one processor to all others.

Bypassing operates in a similar manner, except multiple processors can be connected in any *pattern*, provided no two paths cross. Once a bypass pattern is set, communication is bidirectional - e.g. left and right I/O operations send corresponding data n processors up or down stream, according to the bypass pattern configuration. Connections within a 64-processor boundary will require only one instruction, while a worst case bypass pattern will require five instructions. During exchanges which require more than one instruction, processors can continue to perform all other ALU, Multiplier, internal register and external memory operations.

This communication topology is the key to implementing horizontal filtering algorithms. And, because of the large local memory and register stack sizes, vertical and temporal filter operations can be efficiently performed, as well. Figures 5 and 6 illustrate possible vertical and temporal operations using local memory to store data for

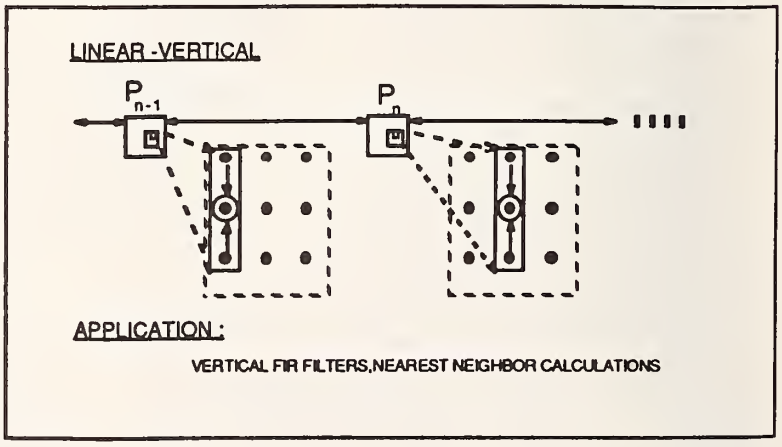


Figure 5. Vertical Filter Operations.

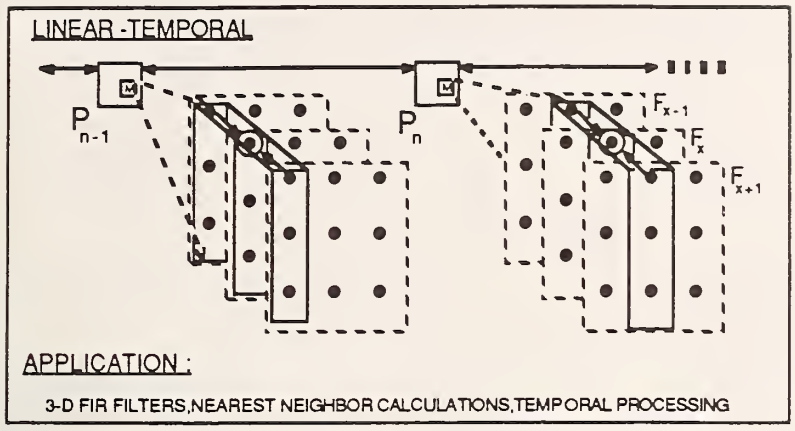


Figure 6. Temporal Filter Operations.

image columns (N-1) and (N) in processors, $P_{(N-1)}$, and P_N, respectively. Each local memory can store up to 32 frames of 16-bit video data for one column. Usually, only two or three frames at most will be stored. Local memory will also store lookup tables, constants and modified or intermediate field data that may be of interest at display time.

Fault Tolerant Linear Array Model

Several researchers have considered the issue of implementing testable and reconfigurable fault tolerant arrays [11,12]. In Kumar [11], a design criteria for a successful model of fault tolerant computing is proposed. This model includes the following:

A) A linear processor arrangement with local parallel busses. The processor configuration and IPC bus in the Princeton Engine (PE) meets this criterion and is fully programmable.

B) Propagation delay is assumed to be proportional to wire length. Introduce unit delay whenever a processor is bypassed. In the case of an isolated faulty processor, PE bypassing modes permit single instruction exchanges of data between adjacent processors. In general, bypassing faulty processors in the PE will meet this criterion.

C) The clock rate is independent of the number of faults in the array. This criteria is met given the limit of the fault covering a boundary of 64 processors. In the PE, the I/O clock rate is independent of the processor clock rate.

D) Busses, unlike processing elements, are assumed to be reliable. This is also the assumption in the PE.

E) Fault tolerance depends on being able to connect good PE's into a linear connected array. The IPC bus of the Princeton Engine guarantees this.

Fault tolerance for video applications is maintained in the Princeton Engine, provided there are more processors available than pixels in the particular display format. In a two cabinet engine, there are positions for 1024 processors, which is 114 more than the number required for NTSC signal processing. A diagnostic program runs during system initialization and tests each processor. Those processors which fail are immediately bypassed. Provided there are sufficient processors, and regardless of the resulting bypass configuration, all programs will run unaltered.

Graphical Programming

The complexity of a system of 2048 processors requires a new method of program development which enables the engineer to implement algorithms at a high level of abstraction without having to consider details of code generation for all the potential processors in the system. A software development environment, which permits a high degree of programming parallelism, has been implemented. Figure 7 shows the overall system software flow. The development system consists of four major components: a Graphical Program Composer (GPC), a Graphical Programming Editor (GPE), the Concurrent System Simulator and Debugger (CSSD), and the Graphical Control Environment (GCE).

Signal processing engineers conceptualize systems in terms of high-level building blocks, where the functional behavior of each block is usually well understood. It is the unique composition of these blocks which creates new and novel systems. Simulations for such systems using conventional programming languages require a change in the designers' conceptual framework from an inherently parallel one to a sequential one. In actual DSP system implementations, however, processing is frequently performed in parallel.

GPE

The GPE permits the user to symbolically lay out an al-

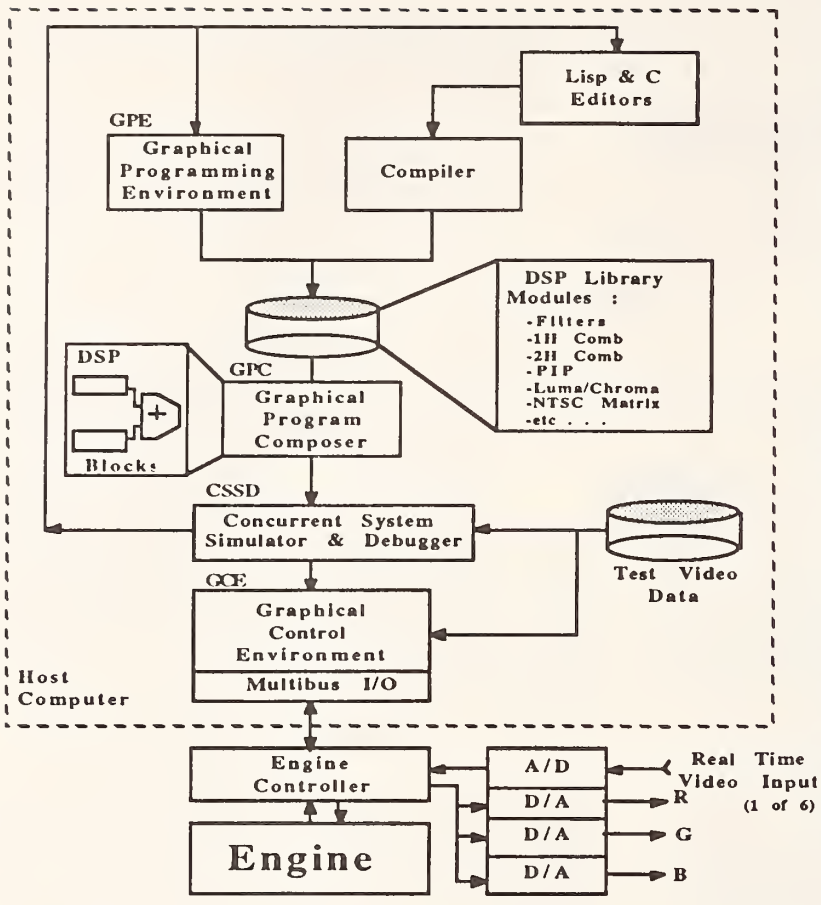


Figure 7. Software Development Environment.

gorithm using a picture of a single pseudo-processor (refer to Figure 8). Each instruction is created by graphically routing data between various source and destination points within the processor. Instruction flow is controlled from within the GPE environment by simple interactive graphic commands using a mouse pointing device. Paths are selected by pointing to the graphic icon representing the operator or register within the processor. For example, referring to Figure 8, *Inst 2*, a path is selected from the memory input buffer (DI) to the multiplier where the product of a filter coefficient and a pixel are generated. The filter coefficient value was entered via the immediate field and appears on the display as the I00C3 lable on the multiplier icon. At the same time, the data from the memory input buffer (DI) is routed to the I/O buffer (IPC_BUF).

The multiplier has a programmable product picker which enables any 16-bit subrange of the 32 bit product to be selected as output. In the third instruction, *Inst 3*, the 16-bit *picked product* (PP) output is routed to the ALU and stored in the accumulator at the end of the instruction cycle. In addition, data from the I/O buffer (IPC_BUF) is sent "left" as indicated in the graphical layout. At any specific time within the GPE, the user can increment or decrement instructions, return to the first or last instruction, insert or delete instructions or print out a graphical transcript of the entire program.

The selection of an appropriate ALU operation is made from an on-screen, mouse sensitive menu which displays all possible ALU operations. These ALU operations include several functions specifically designed to simplify the task of programming DSP algorithms, such as clipping, sign extension, byte packing, byte swapping, minimum, maximum, absolute value and two's complement arithmetic. Minimum and maximum functions are useful for median filters while byte swapping and packing improve local memory storage efficiency. Two's complement capability provides for simplified arithmetic operations on bipolar signal inputs.

As instructions are graphically generated, a transcript of operations is maintained. This includes data slots for labels, lookup tables, branch operations and immediate field entries. Branch operations give the system the capability to conditionally execute a string of instructions based on the result of an ALU operation. Status conditions for nearly all ALU operations have been provided: <=0, >0 >=0, =0, <>0, A>B, A<B, overflow and underflow. Branch control is achieved by conditionally locking those processors which fail the status condition. They remain locked until either a conditional unlock or global unlock command is issued. The GPC code generator inserts the correct branch ID codes into the instruction field to accomplish branching. Up to 256 branch ID's can be used in one simulation.

Program controls and binary codes for the engine are generated automatically by the system from this transcript of graphical operations. In the Princeton Engine system, the user builds a simulation data base using the GPE to implement primitive DSP functions such as FIR filters. Figure 9 shows the block diagram for a basic five-tap filter. Filter coefficients, C1 through C5, are ini-

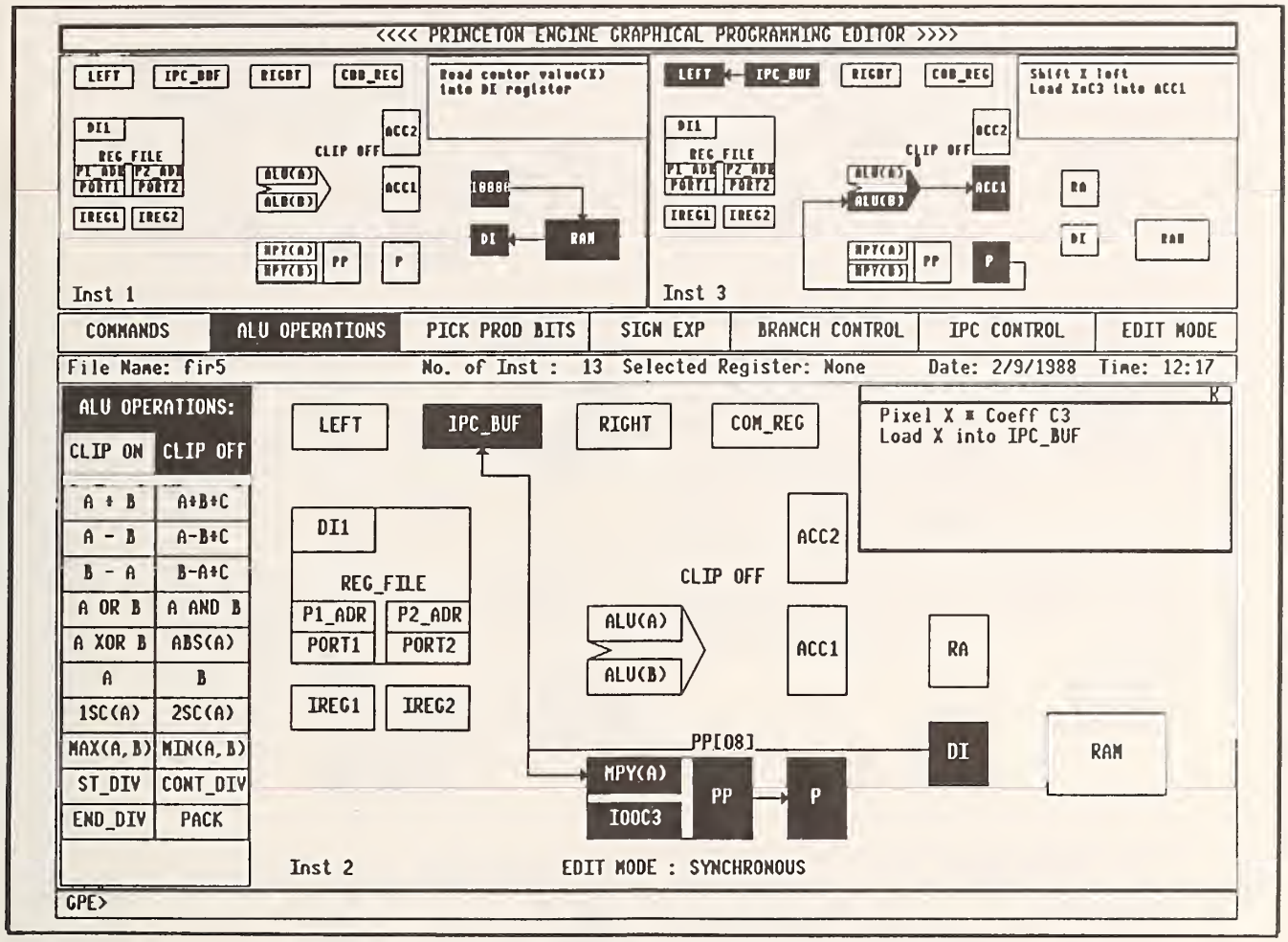


Figure 8. GPE Example of FIR Filter Program.

respectively. The GPE program requires twelve microinstructions to perform the 35 algorithmic steps necessary to implement this five tap FIR filter. In general, an N-tap horizontal FIR filter can be programmed in N+8 processor instructions. Figure 8 shows the GPE envirment and first three instructions of the horizontal FIR Filter example.

The GPE implementation steps are as follows:

First, the current pixel is loaded from external memory into data input port, DI, using the immediate field for the address. During the second instruction, the immediate field contains the value of the filter coefficient, Cn, and is loaded into multiplier input port, MPY(B). Then the current pixel is multiplied by the coefficient, Cn, and again, *in parallel* , the pixel (X) is transfered to the IPC_BUF. During the third instruction, the product is summed into the accumulator, while, *in parallel* , the pixel (X) is shifted left. Because each processor is performing the shift operation simultaneously, at the end of the third instruction cycle each processor's IPC_BUF will contain the pixel from the processor to the right, (X+1). Figure 10 shows the entire graphic programming sequence for the twelve instruction program. During the fourth instruction, this pixel must be stored in a register, while, *in parallel*, the next shift operation is performed. This pattern of instructions repeats to the

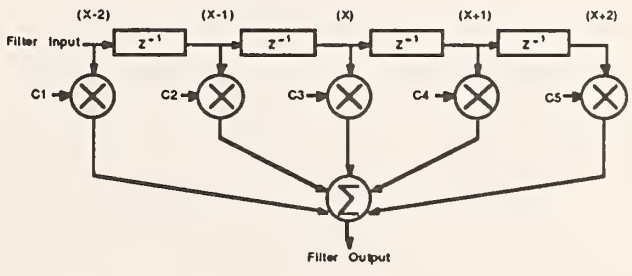


Figure 9. Five Tap FIR Filter Example.

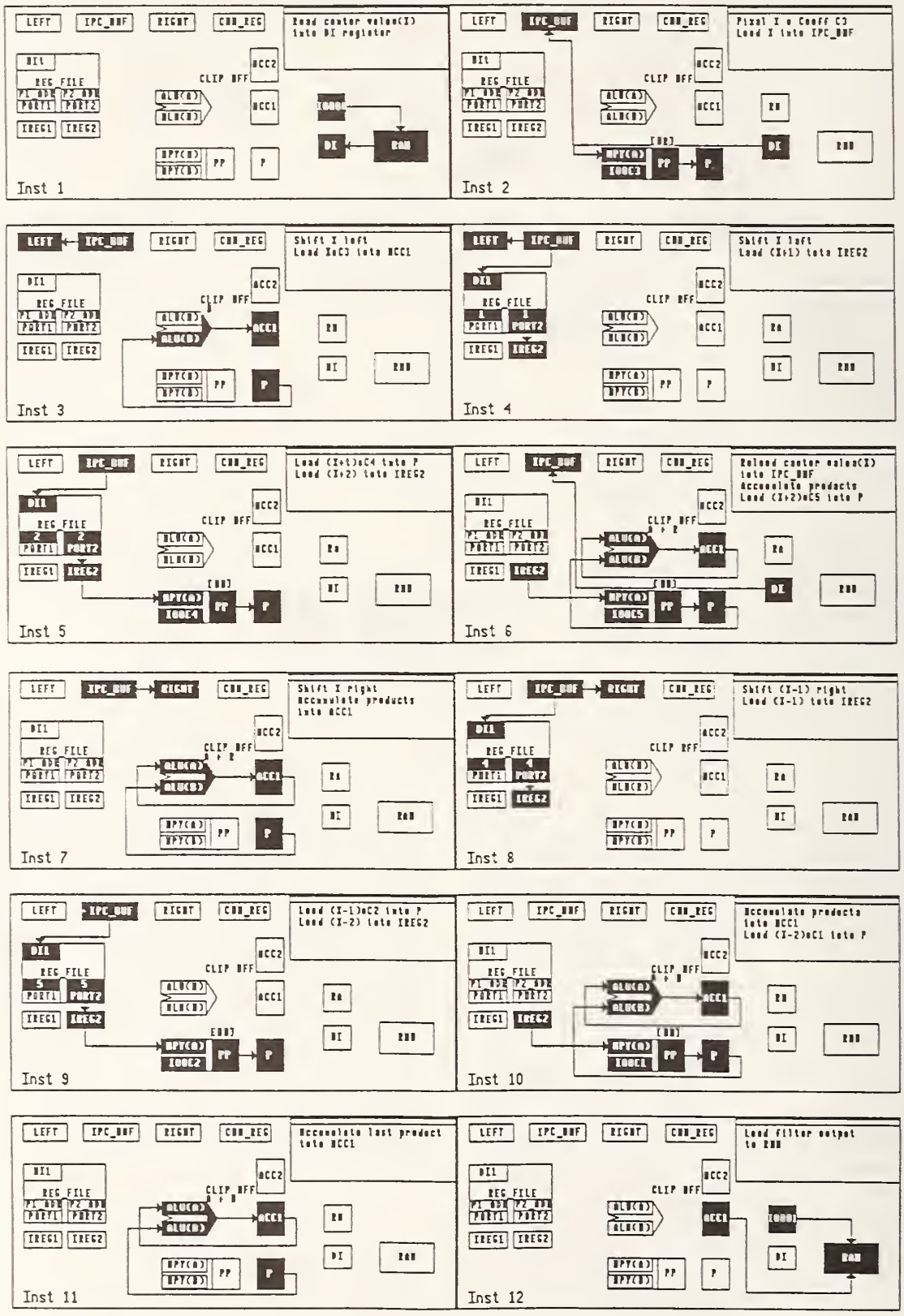


Figure 10. GPE Program for Five Tap FIR Filter.

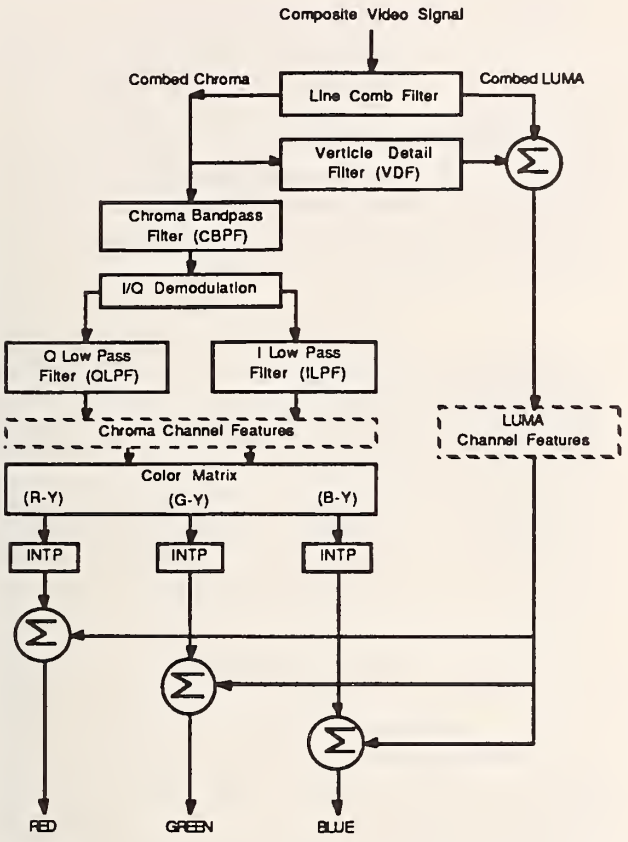


Figure 11. GPC Example of DTV.

formed. This pattern of instructions repeats to the left and right until all the pixels are multiplied by their corresponding coefficients and summed into the accumulator.

It should be noted that the frequent use of the immediate field in the previous example illustrates the degree of modularity possible in the programming environment of the system. A filter module can be created in which the coefficients are entirely parameterizable. For each instantiation of a particular module, the immediate field slots can be filled in with appropriate coefficient values.

GPC

Most system engineers will use the Graphical Program Composer (GPC) to assemble video and DSP systems by composing block diagrams of their design ideas. A robust library of primitive DSP functions has been created using the GPE. In addition, a block diagram component has been created for each DSP function. Figure 11 shows a Digital Television system [2] composed of GPE building blocks. Engineers will configure the engine for NTSC or any television standard, entirely via the GPC. Since most of the existing video systems (NTSC, PAL, etc.) will be included in a library, these will become a logical starting place for new users.

The initial GPC environment uses a commercial EWS schematic capture and netlisting facility. This provides a robust graphical editor in which the user creates a block diagram of the target system using components from a DSP library. This approach provides DSP system designers the same tools, symbols and notation used by other engineers to construct system hardware and VLSI block diagrams. Once the design has been captured, an expansion program extracts the topology and component names from the design and links together the resulting binary codes from each of the individual modules. If the design contains lookup tables, then the corresponding file name must be attached to the symbol for that function. When a particular DSP primitive module (i.e. a new filter design) does not exist, it must be created using the GPE.

CSSD

Developing simulation data bases for complex video systems will be a comprehensive and time consuming task. In most engineering facilities, a single, full capacity simulation system will be shared among the community of engineers. In order to enable hierarchical modules to be developed and debugged, we have implemented a host computer based software simulation system, the Concurrent System Simulator and Debugger, or CSSD.

The CSSD is a LISP-based, object-oriented simulation of the engine system. It is used to develop algorithms and debug application code. Each engine processor is modeled as an independent object. Internal registers and external memory are accessible as data arrays associated with each processor instance. Processors exchange data by sending messages from one processor *object* to another. During each instruction time, a global message (89 bit word length) is sent to all processors, which, in turn, decode the operands and execute the appropriate operations. At any time during the simulation, the user can examine internal registers or status by sending a message to the specific processor or processors. CSSD supports edit and debug operations including break point, tracing and single step.

GCE

The run time environment on the Princeton Engine, the

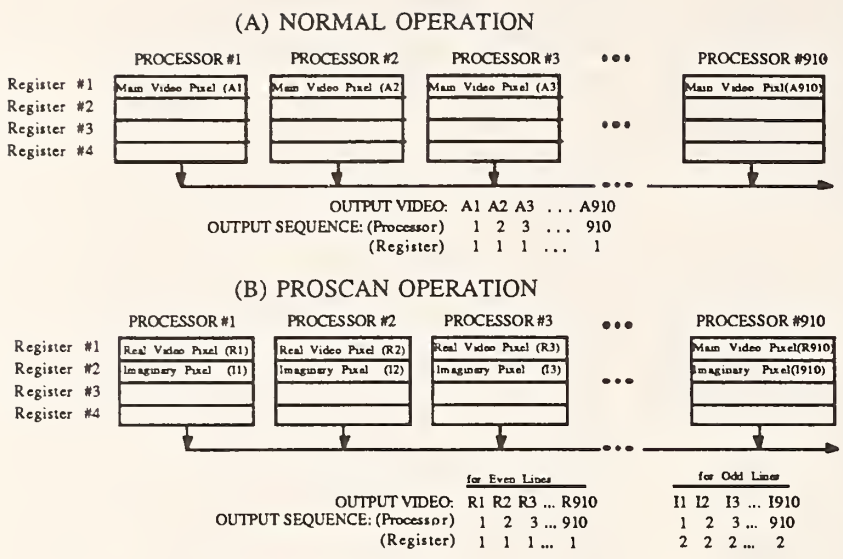


Figure 12. OTS Programming of Progressive Scan.

GCE, has the look and feel of a studio - as though the designer is using a signal generator, logic analyzer and appropriate monitors to evaluate a proposed system on actual video signals. There is a high degree of flexibility in signal source selection as well as output display layout. The display can be configured for picture-in-picture processing or with graphic overlays for histograms or non-video analysis. If desired, the user can make small changes to the GPE or GPC based design, re-compile and re-run a simulation.

Actual control of an engine simulation is accomplished through the interactive GCE program running on the host computer. A complete assembly of engine instruction binary codes are downloaded by the host via a multibus interface into the program memory located on the controller logic board. When a simulation is invoked, the stream of instructions are passed in parallel from the I/O controller to each of the processors. The control program running on the host must perform all the initialization functions including preloading processor memory with program data and running system diagnostics to verify that the correct number of processors are operational.

Simulation attributes, such as video sources, clock and sync signal generation, are all controlled by the GCE. Control command sequences must be sent to the controller at simulation initialization. In addition, selection and planning of the output display layout, pixel placement, and selection of system clocks are accomplished under the GCE. Additional control commands permit users to modify simulation attributes during run time. This permits ordinary video system control operations, such as changing tint, saturation and contrast to be performed in real-time.

The ability to make real-time updates to the entire microword of the processor instruction provides a robust mechanism for modifying filter coefficients and other simulation control variables. Within the GCE environment, the value of any field of an instruction in the sequencer program memory can be changed during the vertical blanking interval.

Dynamic configuration of the output display layout is provided through the Output Timing Sequencer (OTS) bus, which is programmed under control of the GCE. The OTS bus controls the order of pixels being sent to the video output channels. Unlike most parallel machines, the data output processing of the Princeton Engine is completely random, as if all the output data were stored in a single RAM. Furthermore, each of the 64-output bit streams can select from one of four registers from each processor. If the 64-output bits of a 2048 processor machine are programmed as 8 channels, each 8 bits wide, then the total number of 8-bit registers which can be randomly addressed by its corresponding OTS bus would be:

(# of processors) X (# of registers/channel) = 2048 X 4
= 8192

The OTS bus has a processor address field and a register address field. Figure 12 shows a pictorial representation of the output RAM for one of four output groups (each 16-bits wide). The OTS control of output data read from the output RAM is given by the output processor sequence and register numbers. In a normal output sequence, the OTS bus will fix the register address at 1 and sequentially address the processor number as shown in Figure 12A. This would occur anytime one particular channel needs to be transferred to the output at 14MHz. In the case of progressive scan, the second line, or the computed, imaginary line, has its pixel data stored in register 2 as shown in Figure 12B. The OTS bus will sequentially address the processor with the register ad-

dress fixed at 1 and then change the register address to 2 at the end of the first line. The output sequence rate is 28MHz for progressive scan operation.

There are some cases in which it would be necessary to use all four registers to generate the complete display. In the case of a multiple Picture-in-a-Picture (PIP) simulation, register 1 will store the main picture data and registers 2 through 4 will be used to store additional picture inserts.

Running a simulation on the Princeton Engine requires assembling three major GCE program segments. The first segment is the SIMD program code, a run-time sequence of instructions for the processing elements implementing all the algorithms in the target system. This sequence of instructions is the result of the design expansion of a GPC-based description of the system. The second segment is the overall control program which must preload any tables in processor memory, switch the appropriate video sources and select the Output Timing Sequencer registers. Wrapped around these two segments of the GCE is a third segment which is an interactive program running on the host computer. The engineer interacts with this segment to organize the output, to change run-time parameters and to start the simulation. This interactive segment must load the program and control segments into the controller and perform any of the multibus operations necessary to transfer data and program, as well as begin program execution.

Implementation

Initial Princeton Engine systems will be comprised of a sufficient number of processors to implement ACTV and HDTV systems (1216 and 1536 processing elements, respectively). A system is implemented in processor boards each with 32 PROC IC's and 16 I/O IC's. The processor boards are 22"x17", contain 12,000 holes and have been implemented using high density discrete wire technology. Each PROC IC contains two processing elements. The PROC and I/O IC's have been implemented in two 75,000 gate, 1.5 micron, CMOS gate arrays. Each IC is packaged in a 223 pin grid array. Local memory consists of four 16Kx4 static RAMS per processor which are contained on a daughter board assembly mounted adjacent to each double processor IC.

The speed of the PROC IC is currently limited by the gate-array designed multiplier. We believe that a custom design of the PROC chip would realize instruction rates around 30MHz.

An Engine cabinet contains eight processor boards for a total of 512 processors. All controller and analog to digital interfaces are contained in a separate cabinet. A 1536 processor Princeton Engine consists of four small cabinets and consumes two kilowatts of power.

Summary

This paper described a powerful parallel computer architecture which has been organized specifically for video and image processing. This architecture combines a high speed dual processor IC with an I/O IC specifically designed to perform very high speed video simulation computational tasks. The approach makes possible for the first time, true, real-time simulations of very large video systems and permits an entirely new design methodology to be realized in which engineers can explore their design ideas interactively. In addition, the system can be expanded from a single 64-processor board up to a system totaling 2048 processors, with full upward compatibility of GPE/GPC based libraries and making the startup cost low. The Princeton Engine system will be used in the development of signal processing functions and features for existing TV standards (NTSC, PAL,...), as well as to define signal formats for ACTV and HDTV.

While the emphasis of this paper has been on an application specific parallel computer, the SIMD architecture used here is also applicable to the solution of a broad range of problems. SIMD machines have been used to solve a variety of problems including FFT's, terrain mapping, fluid dynamics [13], circuit simulation[14] and logic simulation[15]. They have also been proposed for a variety of other computational problems including IC process and device simulation and animation.

Acknowledgement

We would like to acknowledge the contributions to this project of Robert Maturo, Anthony Guyer, John Lee and Smith Freeman. Bob and Tony were instrumental in seeing the system hardware of the Princeton Engine become a reality. John kept all our computers working and Smith helped produce test vectors for the PROC and I/O IC's.

We also wish to thank the Thomson/GE/RCA Consumer Electronics Division for support of this work.

References

1. Strolle,C.H., Smith,T.R., Reitmeier, G.A., and Borchardt, R.P., "Digital Video Processing Facility with Motion Sequence Capability", International Conference on Consumer Electronics Digest of Technical Papers, p.178, June 1985.

2. Bernard, F. S., Law, K. A., "An Engineering Workstation and Video Frame-Store Peripheral for Simulating Digital TV Signal Processing", RCA Review, Vol. 47, March 1986, pp. 32-66.

3. Isnardi, M.A., Fuhrer, J.S., Smith, T.R., Koslov, J.L., Roeder, B.J., Wedam,W.F., "A Single Channel, NTSC Compatible Widescreen EDTV System", presented at the HDTV Colloquium, Ottawa, Canada, Oct 4-8, 1987.

4. Hwang, K., "Advanced Parallel Processing with Supercomputer Architectures", Proceedings of the IEEE, Vol. 75, No. 10, October 1987, pp. 1348-1379.

5. Hillis,W. D., "The Connection Machine", MIT Press, Cambridge, Mass, 1985.

6. McMahon, E.P., "Applications of Fine Grain Parallel Processing", Signal, April 1987, pp 69-74.

7. Enami, K., Yagi, N., and Murakami, K., "Real-Time Video Signal Processor", SMPTE Journal, December 1987, pp. 1158-1165.

8. Fisher, A. L., "Scan Line Array Processors for Image Computation", Computer Architecture News, Vol. 14, No. 2, p. 338, June 1986.

9. Fisher, A.l., Highman, P.T., Rockoff, T.E., "Architecture of a VLSI SIMD Processing Element.", ICCD, 1987.

10. Kung, H. T., Webb, Jon A., etal, "Warp Architecture and Implementation", Conference Proceedings on the 13th Annual International Symposium on Computer Architecture", 1986, pp. 346-356.

11. Kumar, V.K.P., Tsai,Y.C., "On Mapping Algorithms to Linear and Fault Tolerant Systolic Arrays.", to appear in IEEE Transactions on Computers.

12. Rosenberg, A. "The Diogenes Approach to Testable Fault-Tolerant Networks of Procesors.", IEEE Transactions on Computers, CO-32, No.10, 1983, pp. 902-910.

13. Waltz, D.L., "Applications of the Connection Machine.", Computer, January 1987, pp 85-96.

14. Webber, D.M. and Sangiovanni-Vincentelli, A. "Circuit Simulation on the Connection Machine." 24th ACM/IEEE Design Automation Conference, 1987, pp 104-113.

15. Agrawal, P., Dally, W.J., Ezzat, A.K., Fischer, W.C., Jagadish, H.V., Krishnakumar, A.S., "Architecture and Design of the MARS Hardware Accelerator", 24th ACM/IEEE Design Automation Conference. 1987, pp.101-107.

Biographies :

Danny Chin (MTS) received a BSEE degree in 1979 and an MSEE degree in 1981, both from Columbia University. He joined the David Sarnoff Research Center in 1979 where his work has covered digital TV tuning systems,VLSI designs, automatic manufacturing methods, digital TV circuit designs including delay line convergence, progressive scan and all aspects of chroma processing, parallel computing architectures and real-time simulation. He has received nine U.S. patents and one RCA Laboratories Outstanding Achievement award.

Stanley P. Knight (GH) received the BSEE degree from the University of Kentucky in I96I and the MSEE degree from Newark College of Engineering in 1969. In 1961, Mr. Knight joined RCA Astro-Electronics where he worked until 1970 on RF system and circuit design and the development of thin and thick film technologies for microwave circuits. He worked at Zenith Radio from 1970 to 1973 on advanced TV tuner designs. He joined the David Sarnoff Research Center in 1973 and became Group Head in 1977. He and his groups have been involved in all aspects of TV receiver design. Since 1982 his group has concentrated on digital signal processing functions and features for future digital TV's. He holds four US patents and has received an IR100 Award for work on MIC's plus three RCA Laboratories Outstanding Achievement awards.

Herb Taylor (MTS) received a B.A. in Physics in 1976 from Franklin and Marshall College. He worked at TI from 1977 to 1981 on various VLSI designs relating to graphics and computers. He joined the David Sarnoff Research Center in 1981 where he contributed to the development of a design methodology of VLSI for digital television, to architectures for parallel computing and to a graphics chip design for the RCA-GE DVI system. From 1985 until 1987, Mr. Taylor represented RCA as a Member of the Technical Staff of the Microelectronics and Computer Technology Corporation (MCC) in Austin, TX.

Joseph Passe (MTS) received a BSEE degree in 1967 and an MSEE degree in 1968, both from Rutgers University. He served as an officer in the United States Marine Corps from 1969 to 1972. He held positions of Senior Engineer through Staff Engineer from 1972 to 1985, working at TAL-STAR Computer Systems, Autodynamics Inc., and Lockheed Electronics Co., Inc. where his work included all phases of digital logic design for micro- and mini- computer based systems. Since joining the David Sarnoff Research Center in 1985, he has been involved in the design of digital TV signal processing circuits and parallel computing architectures.

Francis S. Bernard (MTS) earned a BS degree in Electrical Engineering from Rutgers University in 1982 and a MS degree in Electrical Engineering from Georgia Institute of Technology in 1984. He joined the David Sarnoff Research Center in 1984. His work has concentrated on digital signal processing and features for digital TV including picture-in-a-picture, delay-line convergence, field progressive scan systems and graphical programming techniques for parallel computing architectures. He holds one US patent.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NIST/TN-1288

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
August 1991

4. TITLE AND SUBTITLE

Video Processing with the Princeton Engine at NIST

5. AUTHOR(S)

Bruce Field and Charles Fenimore

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED
Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Same as item #6.

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This document describes the NIST program in digital processing, including a newly created Image Processing Laboratory at NIST that is available to governmental, industrial, and academic researchers working on digital image processing. The centerpiece of the laboratory is a video supercomputer, the Princeton Engine, designed and constructed by the David Sarnoff Research Center. The engine provides real-time video and image-processing capability, accepting a variety of video formats over multiple wideband input channels and outputting real-time video for immediate viewing. Because the Engine is programmable, it is possible to use it to evaluate prototypes of image processing components rapidly and efficiently.

The hardware capabilities of the Princeton Engine are described as well as the available supporting video equipment in the Laboratory. Two programming examples are included to demonstrate the unusual programming environment and "language" used to program the Engine. Appendices list the available predefined library modules, and the processor assembly language instructions.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

data compression; digital video; image processing; supercomputer; video processing

13. AVAILABILITY

X	UNLIMITED
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
X	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
51

15. PRICE

ELECTRONIC FORM

U.S. Department of Commerce
National Institute of Standards and Technology
Gaithersburg, MD 20899