REAL TIME PICTURE PROCESSOR: DESCRIPTION AND SPECIFICATION

NCI/IP Technical Report #7a

March 31, 1976
Revised June 23, 1977

Peter Lemkin, George Carman,*
Lewis Lipkin, Bruce Shapiro,
Morton Schultz

National Cancer Program

U.S. Department of Health, Education, and Welfare / National Institutes of Health / National Cancer Institute

NCI/IP-77/02

Real Time Picture Processor - Description and Specification

NCI/IP Technical Report #7a

Peter Lemkin, George Carman*, Lewis Lipkin,
Bruce Shapiro, Morton Schultz

Image Processing Unit
Division of Cancer Biology and Diagnosis
National Cancer Institute
National Institutes of Health
Bethesda, Md. 20014

*Carman Electronics, Inc.
Corvallis, Oregon

March 31, 1976

Revised June 23, 1977

ABSTRACT

The concepts and general design specifications of a new hardware picture processor are presented. The design was strongly influenced by the characteristics of biological images. This device, now in the early stages of construction at the National Institutes of Health, will meet some of the requirements for interactive design, specification and testing by biologists of algorithms for cell classification, description and measurement. The RTPP is but one component, albeit a major one, of our entire system which is intended to permit on-line description construction by the cytologist.

T A B L E O F C O N T E N T S

L I S T O F F I G U R E S

L I S T O F T A B L E S

SECTION 1

Introduction

This paper discusses the functional hardware design specification of a Real Time Picture Processor (RTPP) ([Lem74], [Carm74]) for use in designing image processing systems for biological materials. Real time, as is used here, denotes time proportional to that required for comfortable human interaction.

The concepts and general design specifications of a new hardware picture processor are presented. The design was strongly influenced by the characteristics of biological images [Lem74]. This device, now in the early stages of construction at the National Institutes of Health, will meet some of the requirements for interactive design, specification and testing by biologists of algorithms for cell classification, description and measurement. Additional detailed hardware specifications of the Real Time Picture Processsor, RTPP, are documented [Carm76]. The RTPP is but one component, albeit a major one, of our entire system which is intended to permit on-line description construction by the cytologist. The overall discussion of the system components is in [Lem76c].

The Real Time Picture Processor in its role as a microscope controller is designed to perform at least the set of operations performed by the NCI Grain Counter-1.1 [LipL74]. These operations include joystick control of an optical microscope stage in X, Y, and focus. The RTPP, in addition, allows stepping motor control of a 4:1 microscope zoom, a rotary monochromator, and a rotary neutral density filter.

The system is designed for extremly rapid serial digital processing of digitized images to be carried out in what for the user, seems like, real time. Special purpose hardware makes this speed possible. Output from the RTPP is in the form of images to be displayed, lists of properties of objects in the image, or processed images. Because of powerful gray scale manipulation instructions, computations are not limited to processing planes of binary valued images.

These capabilities allow flexible experimentation with an on-line microscope image picture processing facility. Employing this facility, users have the ability to generate precise definitions of biological cell classifications using the RTPP as input for an interactive relational data model residing on a remote PDP10 computer. Furthermore, the system will allow the user to make measurements of cell parts and to develop heuristic measures for cell characterization.

Because processing is fast (on the order of tenths to a few seconds), the interaction between user and the system is well matched to allow for experimentation with more complex algorithms than usually attempted with serial systems. Several papers ([Lem74] and [Lem75]) give reviews of some recent special purpose image processing hardware.

1.1 Overview of this paper

Section 1 introduces the rationale on which the structure of the RTPP is based. Sections 2 through 5 discuss components: i.e. the Quantimet (QMT), buffer memories (BMs), triple line buffers, and General Picture Processor (GPP) respectively in more detail. Section 6 discusses the physical implementation of the RTPP while Section 7 comprises references. Appendix A details the GPP instruction set while Appendix B lists the GPP I/O registers and GPP I/O lights and switches. Appendix C presents the PDP8e Input Output Transfer instructions used in the RTPP. Appendix D gives several examples, including estimated running times, of several algorithms programmed for the RTPP.

1.2 RTPP design goals

There were several (sometimes conflicting) design goals used in defining what should go into a picture processing facility such as the RTPP. Long experience in picture processing on general purpose serial machines has resulted in the production of picture processing packages such as PAX [JohnE70]. Some of these operations consist of simple pixel boolean binary planar operations, 4 and 8 neighbor operations, and then more complex operations such as blob finding, etc. Obviously, as the complexity of an operation increases, so does the complexity of its hardware implementation. We decided that neighborhood processing was the upper limit of complexity that the resources availiable to our group would allow us to undertake. Several years of biological experience posed another requirement: i.e. that grayscale texture measure algorithms be easy to implement. This means doing integer arithmetic on gray scales images rather than boolean arithmetic on planes of binary images.

A general gray scale parallel processor is an unrealistic goal. The complexity of such a device, (at the current state of the hardware art and occasioned by the combinatorics of the required hardware interconnections) place it well beyond the construction abilities of a small group. The cost of such a device also would make it difficult to justify if it were to be dedicated rather than a shared resource.

As will be seen we have drawn often from the good efforts of other computer and picture processing system designs, (especially [Dec71], [Dec72a], [Dec72b], [Dec67], [Thor70]) as well as many other influences. It was hoped that some of the best (and not necessarily the most complex or costlv) features of some of the above machines could be incorporated into our design.

Before going into the design of the RTPP the definitions of some of the terms used in the rest of this paper are given.

Picture - set of gray values

A image part of a picture is a square array approximation of the corresponding real image. Each element in this sampling array is said to have an integer gray value called a pixel. This gray value is a measure of the darkness or whiteness of that pixel. In our discussion, white will be zero and black 255. This representation is said to have 256 linearly resolvable gray levels or values. The gray value is generaly given in the range of the powers of a binary number as it is usually derived from the output of an analogue to digital converter. i.e. 8, 16, 32, 64, 128, 256 etc. An alternative view (which lies at the basis of such implementations as PAX) of an image is as a set of overlying binary arrays more commonly called binary planes. Thus 256 gray values is represented by 8-bits or 8 binary planes.

Neighborhood

Within a image, one usually deals with the concept of neighborhood. A neighborhood is a set of pixels close to (usually touching) a given pixel. An example of this is all pixels touching a given pixel in a array. For a square array there are 8 such pixels and consequently this 3x3 set of pixels is called the 8 neighborhood or 8 nearest neighbors. In this paper and in the RTPP the labeling of the 3x3 neighborhood is as follows:

```
3 2 1
4 8 0
5 6 7.
```

This choice of neighborhood labeling facilitates chain coding as the pixel index corresponds to its angle with the central pixel divided by 45 degrees.

Current pixel

The current pixel is the central pixel (8) in this 3x3 array. The current neighborhood is the 3x3 array surrounding the current pixel.

Neighborhood operation

A neighborhood operator is a unary or binary operator which maps neighborhoods into pixels. One of the ways the neighborhood operator works is to do various operations on parts of the neighborhood array as specified by an ordered list of pixels to be operated on. When this list is ordered in terms of the orientation of neighborhood elements, it is called a direction list.

Pixel operator

A pixel operator is a unary or binary operator which maps pixels into pixels in a 1 to 1 (x,y) mapping.

The RTPP performs neighborhood operations by executing a program on a neighborhood for each neighborhood in the plane. This is actually done using special purpose hardware called the General Picture Processor (GPP) which will be discussed below. The GPP is designed to allow pixel operations to be performed on all pixels of a 3x3 neighborhood within a single I/O access, thus allowing 3 lines to be processed at a time.

1.2.1 Picture operations

Image data processing in particular consists of doing mostly binary argument algebraic operations. These can be enumerated in an interesting way as shown below. The following are binary algebraic picture operations where "o" is a general operator. Let I1, I2 and I3 be images. Then,

```
I1 o I2 --> I3
I1 o -->I3
I1 o I2 --> a property list
I1 o I2 --> a relational data structure.
```

The operation I1 o -->I3 may be thought of as either a unary operation or a binary operation with a null second operand.

These picture operations might include such operations as boolean: bit set, bit clear, bit complement, and, or, exclusive or, equivalence; direction list processing; maxima; minima; Fourier transforms; threshold slicing; scaling; component labeling; propagation of edges; edge finding; pixel-wise addition, subtraction, multiplication, division; rotation and shifting of images; gradient; gray level histogram. Run-length/gray-scale texture histograms should also be computable by the GPP as well a large class of other picture operations. The PAX functions mentioned above are described by Johnson [LipB70].

In Appendix D sample GPP programs are shown, which compute some of the above mentioned operations and for which timing estimates are given.

1.2.2 Picture operations as binary operations

These basic binary argument relations on picture data are typical of picture processing in systems. What varies is the complexity of the operation "o". The increasing complexity of operations in the binary image plane is shown below in five increasingly complex levels I to V. Let "x" below be a pixel in a NxN neighborhood. As will be shown, an analogous hierarchy also exists for gray scale machines.

Let Ij(k) be the k'th pixel in picture j.

(I) Pixel (picture element) operations:

I1(k) o I2(k)-->I3(k).

(II) Neighborhood operations around a pixel into a pixel:

```
xxx
xxx  o  I2(k) --> I3(k).
xxx
   I1
```

(III) Neighborhood contextual operations into a pixel:

```
xxx     xxx
xxx  o  xxx   --> I3(k).
xxx     xxx
   I1      I2
```

(IV) Neighborhood operations into a neighborhood:

```
xxx     xxx         xxx
xxx  o  xxx   -->   xxx
xxx     xxx         xxx
   I1      I2          I3
```

(V) special high level functions:
edge detection, differentiation, smoothing, propagation, etc.

With progress from level I to IV and V, the increasing complexity (number of connections, number of nodes, number of modules) for a hardware circuit to perform such an operation increases. We use this wiring complexity as an estimate of the "complexity" of the operation and thus to characterize the complexity of a given level of operation.

Most small neighborhood parallel binary machines exhibit maximum complexity of levels II or III. For an NxN binary plane image these have complexity levels between N^2 and N^4 (e.g. for a 3x3 array this is 9 to 81). The ILLIAC III is an example of a level IV binary machine.

The same five levels of operations could be applied to the class of gray scale image machines. There is only one known existing gray scale processor, the ILLIAC IV.

It is interesting to examine the complexity of gray scale parallel processing operations. Assume a K bit gray scale (for binary images K is 1) and an NxN image. A binary machine having level II complexity would be interconnected on the order of NxNx1. Then a gray scale machine (having K bits of gray scale) of level II complexity would be KxNxN interconnected. The complexity of level III operation is on the order of (KxNxN)x(KxNxN)xN or $(K^2)(N^5)$. For level IV the complexity is of the order of (KxNxN)x(KxNxN)x(KxNxN) or $(K^3)(N^6)$. Obviously, the costs of implementing a machine rise with its complexity. The cost of debugging, documentation, and maintenance follow a similar increase with increasing complexity.

1.2.3 A level IV gray scale machine - 3 address machine

A gray scale picture processor of at least level IV complexity is needed to meet the requirements of the Real Time Picture Processor delineated above. This is the least complex machine that permits full generality. The use of such a level IV machine as a design tool will, we believe, allow the proper selection of some level V functions.

Although a K bit gray scale machine can be simulated using K binary planes, the expression of algorithms is awkward and usually inefficient.

Not surprisingly, a parallel gray scale machine of level IV is beyond the resources of our laboratory. Our solution to the design/maintenance dilemma has been to design a fast serial machine which operates on neighborhoods of rectangularly tessellated gray scale picture arrays. The machine's speed and a felicity in expressing algorithms is achieved partly through the use of triple operand instructions. In this machine a neighborhood has a serial representation.

The following example illustrates serial representation. Let us consider the images I1 and I2 (one of which may be the temporal successor of the other, or they may represent members of a set of serial sections etc). It is desired to find a measure of pattern similarity (in order to spatially or temporally align the images). The problem is solved by finding a measure of pattern similarity between neighborhoods I1 and I2 by computing their product and summing this product into the central pixel of I3. All neighborhoods are 3x3. Neighborhood I1, for example, comprises pixels I1(8) (its central pixel) and boundary pixels I1(0) through I1(7).

The operation to be performed is $I3(8) = \sum_{i=0}^{8} I1(i)I2(i)$.

The neighborhood processor could perform the following sequence of operations:

```
Form the products array I3 from I1 and I2.
     I3(0)<--I1(0) MUL I2(0)  ; product of I1(0), I2(0) into I3(0)
     I3(1)<--I1(1) MUL I2(1)  ;            I1(1), I2(1) into I3(1)
     I3(2)<--I1(2) MUL I2(2)  ; etc.
     I3(3)<--I1(3) MUL I2(3)  ; naturally there is a way
     I3(4)<--I1(4) MUL I2(4)  ; to do this with a loop
     I3(5)<--I1(5) MUL I2(5)  ; structure
     I3(6)<--I1(6) MUL I2(6)
     I3(7)<--I1(7) MUL I2(7)
     I3(8)<--I1(8) MUL I2(8)

Form the sum of the pixels in the I3 neighborhood.
     I3(8)<--I3(8) ADD I3(0)  ; sum the pixel product into I3(8)
     I3(8)<--I3(8) ADD I3(1)
     I3(8)<--I3(8) ADD I3(2)
     I3(8)<--I3(8) ADD I3(3)
     I3(8)<--I3(8) ADD I3(4)
     I3(8)<--I3(8) ADD I3(5)
     I3(8)<--I3(8) ADD I3(6)
     I3(8)<--I3(8) ADD I3(7)  ; The sum of the entire neighborhood
                              ; product is now in I3(8)
```

These operations, performed over the entire set of corresponding neighborhoods and resulting in the construction of a third image, where the gray value at the point is a measure of the local similarity of the anticedent images, constitutes a solution to the problem.

These iterative neighborhood operations, prohibitively expensive in time and computer cost on standard machines are to be implemented on the General Picture Processor (GPP). This is the fast serial 3-address component of the RTPP. As will be further detained below, the 3-address machine is closest to the triple operand concept at the base of the mapping: I1oI2-->I3.

1.3 The configuration of the RTPP

The real time picture processor hardware shown in Figure 2 consists of an Imanco Quantimet 720 (QMT); a Zeiss Axiomat microscope with stepping stage, focus and light source; a galvanometer mirror precision scanner; a Dicomed model 31 64-gray level storage display; up to eight 256x256 16-bit gray level buffer memories (BM); a general picture processor (GPP); a PDP8e computer with 32K core; a four 1.5 million word disk cartridge drives; an interactive control desk shown in Figure 3, and a high speed connection to a PDP10 computer on which the modelling programs are run.

1.3.1 Axiomat microscope

A Zeiss Axiomat microscope is used to supply images to the RTPP system. It has been modified so that the focus and 4:1 zoom knob controls are under computer actuated stepping motor control. In addition, the stage X and Y positions of the slide are moved by computer driven stepping motors. The light illumination subsystem contains variable wavelength interference and variable neutral density filters. These two functions are implemented by use of continuous rotary wedge filters which are controlled by the computer via high speed stepping motors. Currently, a Quantimet plumbicon or vidicon scanner and a precision galvanometer mirror scanner are interfaced to output ports of the Axiomat. There is an additional viewing port within the viewer's reach. Thus there are three image planes accessable simultaneously.

1.3.2 Video input and output devices

The video input, display, and certain image processing functions can be performed by various I/O devices connected to the multiple output ports of a Zeiss Axiomat Microscope.

A Quantimet 720 image analyzer has been incorporated into the RTPP. The plumbicon camera video is digitized to 8-bits for sampling by the system (at a 8 MHz rate) and is then reconverted to an analogue signal for reinsertion into the Quantimet video input chain. The Quantimet includes a 10.1 frame/second television display and some minimal feature extraction and measurement hardware to be discussed below.

A precision galvanometer mirror scanner with a 1024x1024 pixel random access window and 256 gray level video output can also be used as a digitizing input device to the RTPP from the microscope. The PDP8e may route mirror scanner data to a buffer memory for display on the Quantimet or further procesing by the RTPP. In addition, a Dicomed model 31

1024x1024 picture point 64-gray level precision storage display may also be used as a storage output device in the RTPP. Presently, hardcopy images from the system are produced by photographing the Dicomed display.

The Quantimet is the preferred I/O device for real time interaction because of its rapid frame rate. The digitized picture from the plumbicon scanner video is a 256 (64 usable) gray level digitized image within a 880x680 pixel window. The QMT display may also be used to post images from buffer memory windows inserted within original scanner image data. There are two types of cameras: a plumbicon and a vidicon. The plumbicon gives better linearity for use with densitometry while the vidicon has a larger dynamic light input range. When one of these cameras is used on the Axiomat at the same time the galvanometer scanner is used the difference in dynamic range of light intensities required is a problem. This is solved by putting N.D. filters in front of the TV camera. In addition, a shutter protection circuit is used to turn off the image when no TV scanning is being done or if the light level into the camera is too great.

1.3.3 Buffer memories - BM

Although a histologic slide can be used as a random access read only memory, it is inadequate for use in processing the information it contains. It is necessary to retain images and their transforms in buffer memories, and to be able to access them and display them rapidly. In order to do this our Real Time Picture Processor needs facilities to store at least four entire images or transforms at once (e.g. to store a Fourier transform and a Fourier filter takes 4 images - 2 real and 2 imaginary), and the ability to access part of them very quickly. This rapid pixel accessing is the primary reason for the use of buffer memories.

Our approach has been to design buffer memories each of which can contain a reasonable size of working image (256x256 pixels). At a nominal maximum optical resolution of about 0.25 microns, a biologically usable picture window is approximately 50 microns or more corresponding to about 200 pixels. The RTPP uses a 256x256 pixel window.

Eight 256x256-word 16-bit/word gray scale buffer memories are being built which may be accessed by either the Quantimet video scanner/display, the GPP (general picture processor), or the PDP8e computer. Each buffer memory holds two 8-bit gray scale images, one in the low and one in the high half of the 16-bit BM word. When a BM is not reading or writing data to the GPP, it can be used to continuously recycle a picture for posting on the Quantimet display and/or as input to the Quantimet video-input chain using a fast 8-bit digital to analog converter. The PDP8e controls the scanning (into) and the posting of images (from) the buffer memories. In addition, the PDP8e computer can read/write data from/to buffer memories using rapid direct memory access (DMA) techniques. The BM can also be used as a binary data mask for Quantimet or other BM data.

A BM can acquire a selected 256x256 window of a QMT scanner image or an image loaded from the PDP8e. This image can then be accessed pixel by pixel or line by line from the GPP or PDP8e. Its contents may also be posted on the Quantimet display (using a fast D/A and digitized video multiplexing). When used in this mode, the QMT is able to process synthesized video. Using a direct memory access (DMA) channel, the PDP8e can then access the buffer memories. The GPP also can directly access the buffer memories for performing transformations on the image.

The buffer memories are organized so that it is most economical to transfer entire horizontal lines of data between them and the GPP processing unit. Vertical lines or randomly accessed pixels can also be transferred but at only 1/4 the data rate.

1.3.4 Triple line buffers for image addressing

Given the basic architecture of the GPP, an addressing mode to implement the fast triple operand processing is needed. It was decided that the NxN neighborhood should be directly addressable for all I1(i), I2(j) and I3(k) in the three current NxN neighborhoods (i,j,k), letting the neighborhood be tessilated through the entire image.

The NxN selected is a 3x3 neighborhood because of the combinatorial constraints which increase intolerably rapidly for any larger neighborhood. Three NxN neighborhoods are required for the images I1, I2 and I3; therefore 27 directly addressable pixels are needed for N=3. Forty-eight and 75 directly addressable pixels would be needed for N=4 and N=5 respectively. For any size N one could argue that some algorithm would need N+1. Since a 3x3 neighborhood is the minimum size that intrinsically handles 4 and 8 neighborhood processing (a symmetrical central pixel), at least a minimum of processing power exists with this size. If necessary N+1 can be simulated in software for any N. Therefore, it was decided to hold down the complexity of the hardware and let N=3.

Finally, to make an entire neighborhood directly addressable, the processor has three line-buffers, each capable of holding N entire lines of image data (N=3). Figure 1 illustrates an N-line buffer with K-bits/pixel and M pixels/line. In this design (N,K,M) = (3,16,256). Three triple line buffers are implemented.

The processor can transfer an entire line at a time either between a buffer memory and the line buffer or vice versa. The line buffers, being implemented with fast registers, constitute a kind of cache memory (a special type of hardware used to optimize data rates between a processor and its main data memory).

The problem of addressing a neighborhood is reduced to the problem of addressing the line-buffers. Thus, neighborhood processing of an entire image can be accomplished by a sequence of actions: first, processing is done on each 3-pixel-wide current neighborhood of an 3-line-deep line-buffer. Then a processed line (usually the oldest) is moved out of the line-buffer to a buffer memory and a new line, usually the next line in the raster source image, replaces it. The processing loop is then repeated, until the entire image has been processed. Three line buffers, providing 8 neighbor processing, maintain the full generality of level IV operations. A more complete description of the hardware is given later in this document.

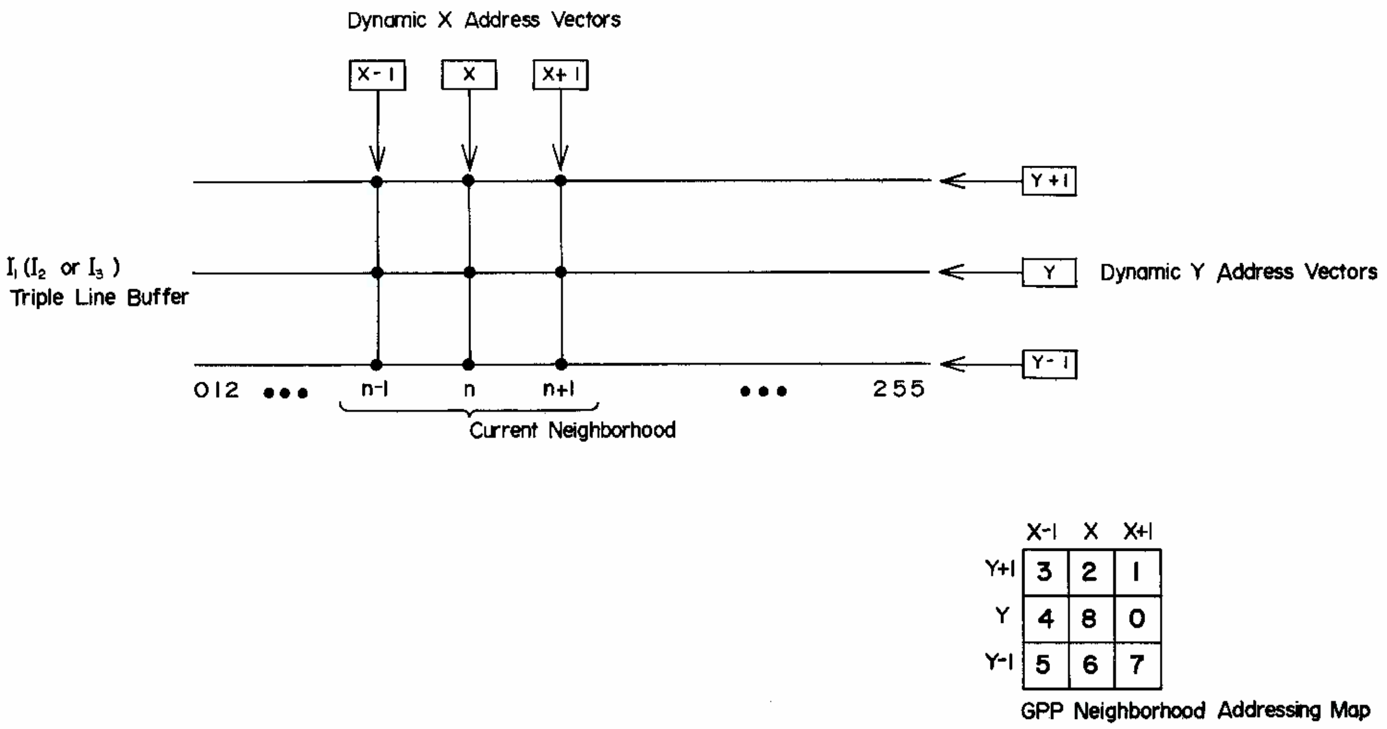
Dynamic X Address Vectors
X-1
X
X+1
Y+1
Y
Y-1
Dynamic Y Address Vectors
I_1 (I_2 or I_3) Triple Line Buffer
012 •••
n-1
n
n+1
•••
255
Current Neighborhood
X-1 X X+1
Y+1 3 2 1
Y 4 8 0
Y-1 5 6 7
GPP Neighborhood Addressing Map

Figure 1. GPP triple line buffer addressing

Associated with each line buffer is a set of three X and one Y dynamic address vectors. These dynamic address vectors define the current 3x3 neighborhood pixels in the line buffer. Therefore tessellation through different neighborhoods is easily performed by changing these address vectors. An instruction is available to selectively increment or decrement (X-1,X,X+1) in I1, I2, and I3 at one time thus effectively moving the current neighborhood to the left or right respectively.

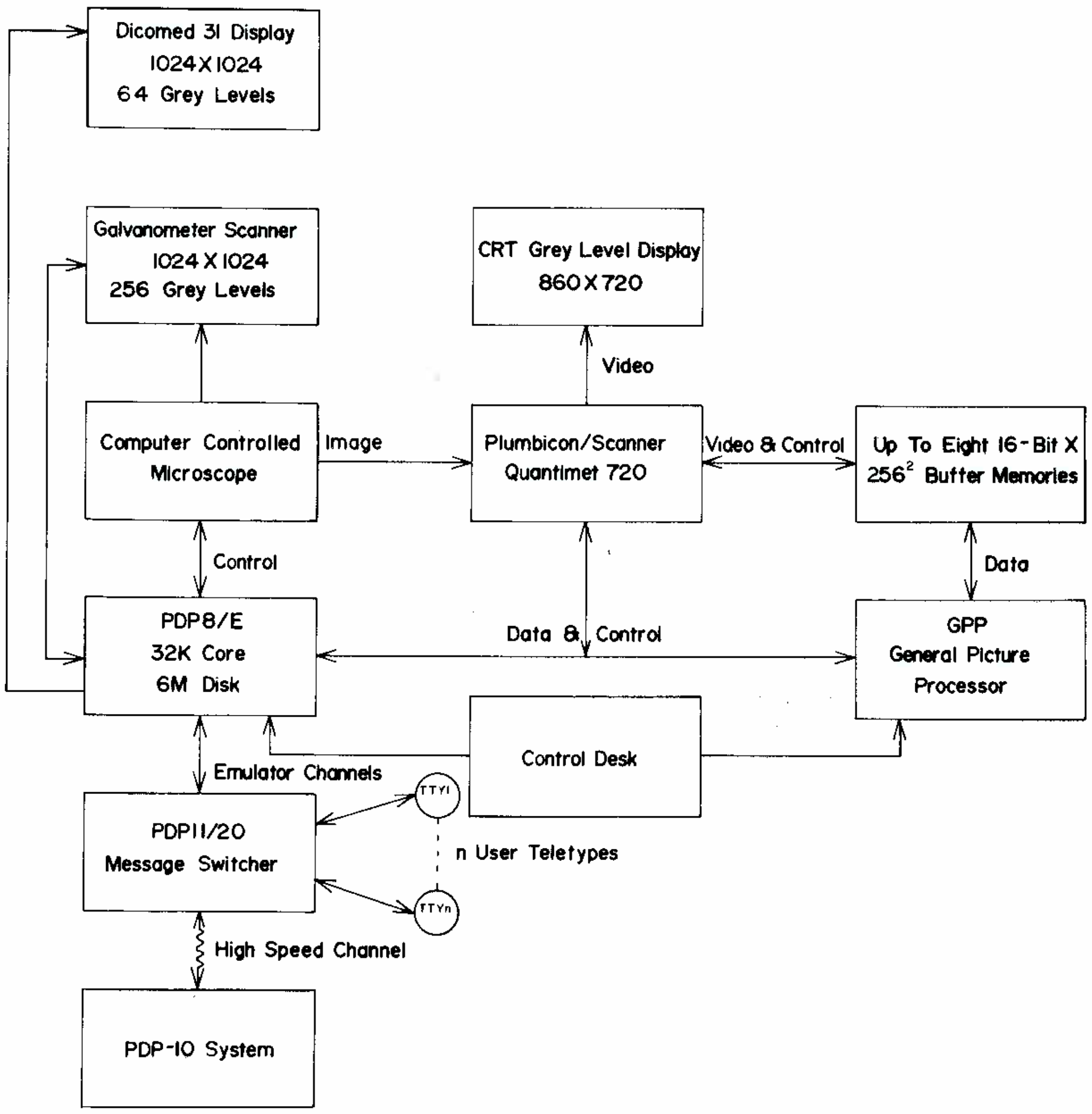

Figure 2. Block diagram of RTPP

The PDP8e computer directs the microscope stage to positions determined either manually by the operator or automatically by the PDP10 system. Images may be acquired by the buffer memories for processing by the GPP. Raw images as well as processed images may be displayed on the Quantimet 720. Precision scanning and display are implemented by the galvanometer mirror scanner and Dicomed display respectively. The user may interact with the system either through a control desk or a teletype.

Figure 3. Interactive control desk

The RTPP interactive control desk is situated next to the RTPP with the Quantimet video display to the rear of the control desk and the Axiomat microscope off to the side. A Graf-Pen spark tablet is located immediately in front of the operator with the pushbuttons and lights mounted in two large boxes to the left and right. The remote Quantimet variable frame and scale keys are located in a small box with a movable cable as is joystick for the Zeiss Axiomat (x,y) stepping stage. The latter has a long cable and may be used at the microscope for control of the stage while viewing thru the eyepiece of the microscope. The control desk controls are listed as follows going from left to right and top to bottom (for the left box first): the QSTAT lights indicate the status of the Quantimet interface; the pots are connected to A/D channels in both the PDP8e and the GPP; the GPP switches are read by the SWITCHA register; FBW2 are lighted "command" keys for the PDP8e; DISPLA/B are GPP display registers. the remove frame switch enables the remove frame and scale switches even when the PDP8e has not enabled them; the frame size switch freezes the frame and scale sizes so that a frame of fixed size may be moved around; the standby switch places the Quantimet display and system control in standby mode; the motors enable switch (also in joystick box) enables the stepping motors when the light above it is on; the scan simulation switch moves the galvanometer scanner independently of the PDP8e so that the gain/baseline controls may be setup with using the PDP8e. For the right control box, the controls are: keypad display of keypad input for the PDP8e; FBW3 "classification" keys for the PDP8e; DISP1/2 PDP8e display lights which are decoded as BCD in the top lights and as octal in the bottom lights; FBW4 PDP8e toggle switches; keypad to input 6 BCD digits to the PDP8e; FBW5/6/7 PDP8e octal digiswitches; Execute key used to execute (interpretively by the PDP8e) instructions given in the digiswitches; eight 5-position spring loaded toggle switches control various stepping motors with a fast and slow speed in both forward and reverse directions.

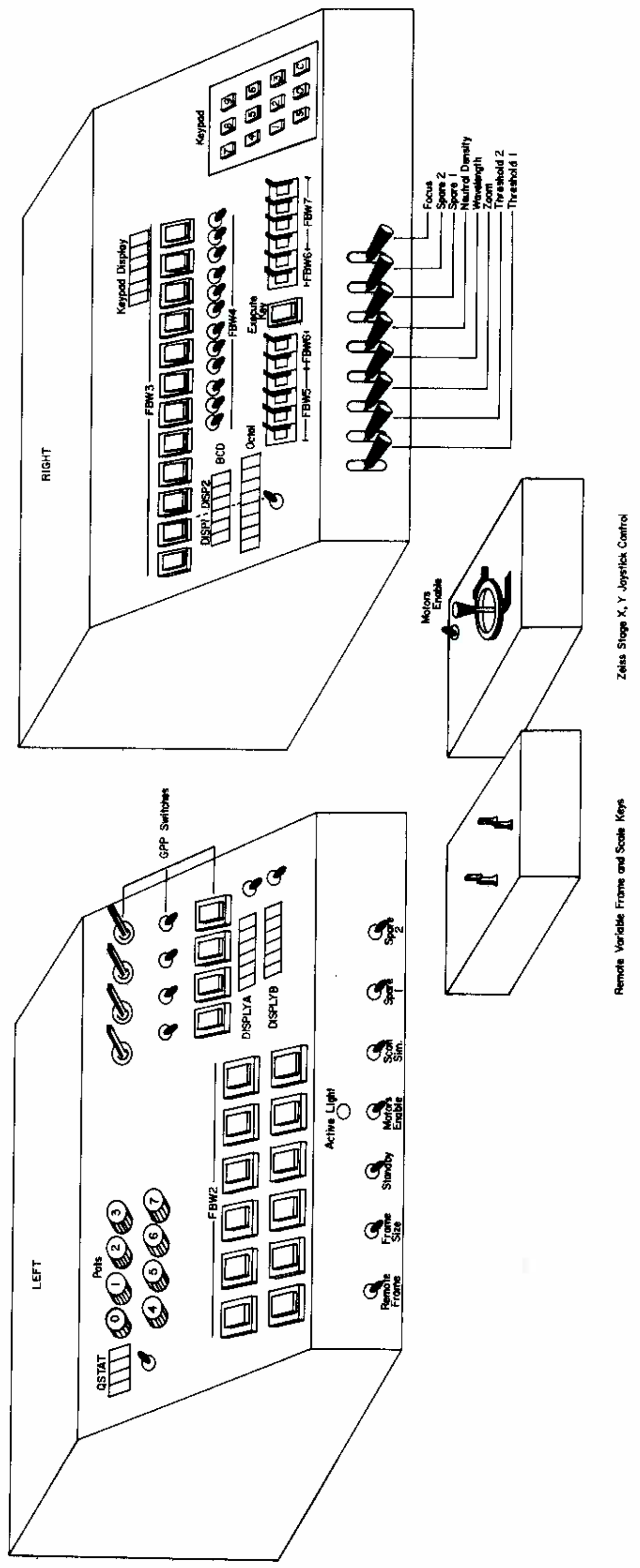
LEFT
QSTAT
Pots
0
1
2
3
4
5
6
7
FBW2
Active Light
DISPLYA
DISPLYB
GPP Switches
Remote Frame
Frame Size
Standby
Motors Enable
Scan Sim.
Spare 1
Spare 2
RIGHT
Keypad Display
FBW3
DISP1
DISP2
BCD
Octal
FBW4
Execute Key
FBW5
FBW6
FBW7
Keypad
Focus
Spare 2
Spare 1
Neutral Density
Wavelength
Zoom
Threshold 2
Threshold 1
Motors Enable
Zeiss Stage X, Y Joystick Control
Remote Variable Frame and Scale Keys

Figure 3

1.3.5 General Picture Processor

The General Picture Processor (GPP) is a stored program very fast serial processor which can rapidly access and process image data through current picture neighborhoods. It uses the concept of triple operand instructions on up to 3 different current neighborhoods. These current neighborhoods may be taken from three diffent triple line cache memory buffers which in turn are backed by the slower buffer memories.

The programs which the GPP executes are loaded into a GPP program memory (PM) by the PDP8e which controls the GPP as a peripheral device. The GPP programs will be written and compiled on the the PDP10 system and assembled on the PDP8e. The PDP8e interacting with the PDP10 will be able to load the binary compilation files into the GPP or to save (get) these files on its own local disk for later use.

The PDP10 system, which is described subsequently in other documents ([ShapB77], [Lem76c], [Lem76d]), and in sections (1.4-1.5,1.7) is written in PDP10 SAIL [VanL73] language. It consists of a high level procedural description language PRDL [ShapB77] which will be used interactively to build morphological descriptions of biological images. PRDL will cause the low level picture processing functions to be evaluated on the RTPP rather than on the PDP10. An image processing program PROC10 [Lem76d] is currently being used on the PDP10 to emulate various image processing procedures which could run on the RTPP. Various often called procedures such as fetching a picture neighborhood have been shown (using PROC10) to be useful image processing primitive operations which are time consuming using normal PDP10 instructions. Thus such operations are commited to GPP hardware instructions (both through actual hardware and through microprogram implementation).

Running RTPP programs will be accomplished by having PRDL functions for the RTPP be compiled by the MAINSAIL cross-compiler [Wil75] on the PDP10. The output of MAINSAIL is then sent to the PDP8e where it is assembled by the RTPP assembler GPPASM [Lem76b] on the PDP8e. Thus PRDL functions which run on the RTPP can be constructed and later be called by PRDL through the RTPP monitor DDTG [Lem76a] (see section 1.6). The communication between these various processors must be intimate and the ultimate responsibility for insuring this lies with the PRDL system.

BMON2 [Lem77] is a PDP8e image processing system running on the RTPP constructed to date(i.e. excluding the GPP). The RTPP subset consists of a PDP8e computer, Quantimet 720 TV image analyzer, Axiomat microscope with stage, focus and zoom stepping motor control, interactive control desk, GraphPen tablet, and image buffer memory. Control of the system is effected by interaction through the teletype or control desk. Teletype commands are entered via the OS8 command decoder (with the implication that BMON2 may be run under OS8 BATCH). Over

100 operations are available. Results are converted to microns for user convenience. Teletype commands may be dynamically assigned to the 12 command keys on the control desk which may be saved and restored from disk files. This allows individually selected subsets of BMON2 commands to be called directly from the control desk.

1.3.6 Quantimet controller

The interface between the Quantimet and the PDP8e consists of an interface to "program" the Quantimet (e.g. use thresholded video (by some criteria) to measure the total area); acquire QMT data; display numbers on the right QMT number display; load various QMT detector threshold and sizing limit registers and "live frame" frame and scale window coordinates.

In addition, special hardware is used to acquire a set of 2-properties/object for all objects in the scene (up to 1024 objects). The properties are taken from the set of object features computed by the intrinsic Quantimet Function Computer modules. These properties include: integrated density, area, perimeter, horizontal and vertial projections, and horizontal and vertical ferets. This acquisition is accomplished during a single scan of 0.1 second. The additional hardware constructed by us includes, a 1024 datum deep stack of the bottom most end points of blobs called Anti-Cooincidence-Points (X-ACP, Y-ACP); a stack of 1 bit/word detector-bit of the associated detected level; and a associated dual data stack which accepts two QMT data words from the function computers of 6 BCD digits each.

At the occurence of each ACP (which corresponds to the recognition of a discrete blob) all the above stacks have data pushed into them. The PDP8e can unload the data after it has been acquired by the Quantimet. In addition, data can be pushed on matching the (X,Y) coordinates of the QMT with those of the front of the (X-ACP, Y-ACP) stack. Since the ACP stack may be loaded by the PDP8e, this offers further possibilities for data acquisition by measuring the detection at the coordinates of the synthetic ACPs.

Other Quantimet related hardware includes a specially constructed Mask register for acquiring and, storing and using arbitrary convex shapes and a programmable display cursor. Both of these devices are described in more detail in Section 2.

1.3.7 Control of the RTPP by the PDP8e

The RTPP complex, (Figure 2), under control from the PDP8e minicomputer, may position the active microscope elements (eg. focus, stage, zoom, etc) and adjust the live frame image of the Quantimet according to program requirements or in response to the operator's directions at the control desk. The PDP8e has full control over the Quantimet to program it and acquire single data scans. Live frame video (within specified 256x256 windows) may be stored in any or all eight buffer memories, and displayed on the Quantimet at the direction of the PDP8e. To process image data in the buffer memories, the

PDP8e loads a specific program into the instruction memory of the GPP and initiates execution. Results may be passed back to the PDP8e for storage or transfer to other computers, or displayed on the Quantimet, or both.

1.4 The RTPP as a picture processing peripheral

The RTPP is really a specialized peripheral processor, a level of complexity beyond the now accepted class of display controllers. The latter usually consists of a programmable special purpose computer with a moderate amount of memory and a digital image display. The display controller is intended to remove from its large general purpose host computer a large portion of the burden of interactive display programs. In terms of the host computer and its associated facilities there can be little doubt that such a solution to the display problem is usually economically and computationally justified.

It may be noted that the RTPP, as a picture processing peripheral, removes a proportionately greater burden from the host computer's CPU and core memory than does the display controller. There is no need now to devote as much channel capacity for the transmission of raw picture data and their transforms. Instead, a system of distributed computing, made possible by the RTPP, allows the exchange of higher level data such as property lists, effectively resulting in marked information compression. This in turn allows more effective use of a wide range of facilities available on the general purpose host computer.

RTPP output would usually be images and/or lists of properties of objects contained in the images. The major component of the RTPP real time interaction is the General Picture Processor (GPP). This hardware processor allows extremely rapid serial digital processing of digitized gray scale images using special purpose hardware including image buffer memories and fast addressing schemes. Because the GPP/image buffer memories handle gray scale data (not merely binary black and white images) a larger set of useful picture functions can be quickly designed and executed than with a strictly binary image processor.

1.5 Cell Modeling System - CELMOD

A biological cell relational data modeling system will analyze the scene using RTPP generated picture primitives as in ([Lem72], [ShapB74]) and involves the use of procedural definition on a semantic data base. This modelling system is called CELMOD [Lem76c]. It consists of the three PDP10 programs (PRDL [Lem76b], PROC10 [Lem76c] and MAINSAIL [Wil75]) and the RTPP. A block diagram of CELMOD is given in Figure 4.

This model will be interactive to allow a biologist to define what he means by a particular class of cells and manipulate properties and relations of objects in the images. The biologist by means of iterative composition of processes can build up explicit connections between his semantic information and the image information represented at the individual pixel level. Being an interactive system, it will be easier for him to elicit the subconscious clues that he uses in making these decisions.

The decription language is called the Procedural Description Language (PRDL) which runs on a PDP10 and is described in [Lem76b]. The RTPP will thus also serve as a feature extractor for PRDL. Figure 4 shows a block diagram of the CELMOD system. The overall connection between the RTPP and PRDL is described in more detail in [Lem76b].

The CELMOD system will be able to perform image processing operations on the PDP10 using PROC10 while the RTPP is being constructed.

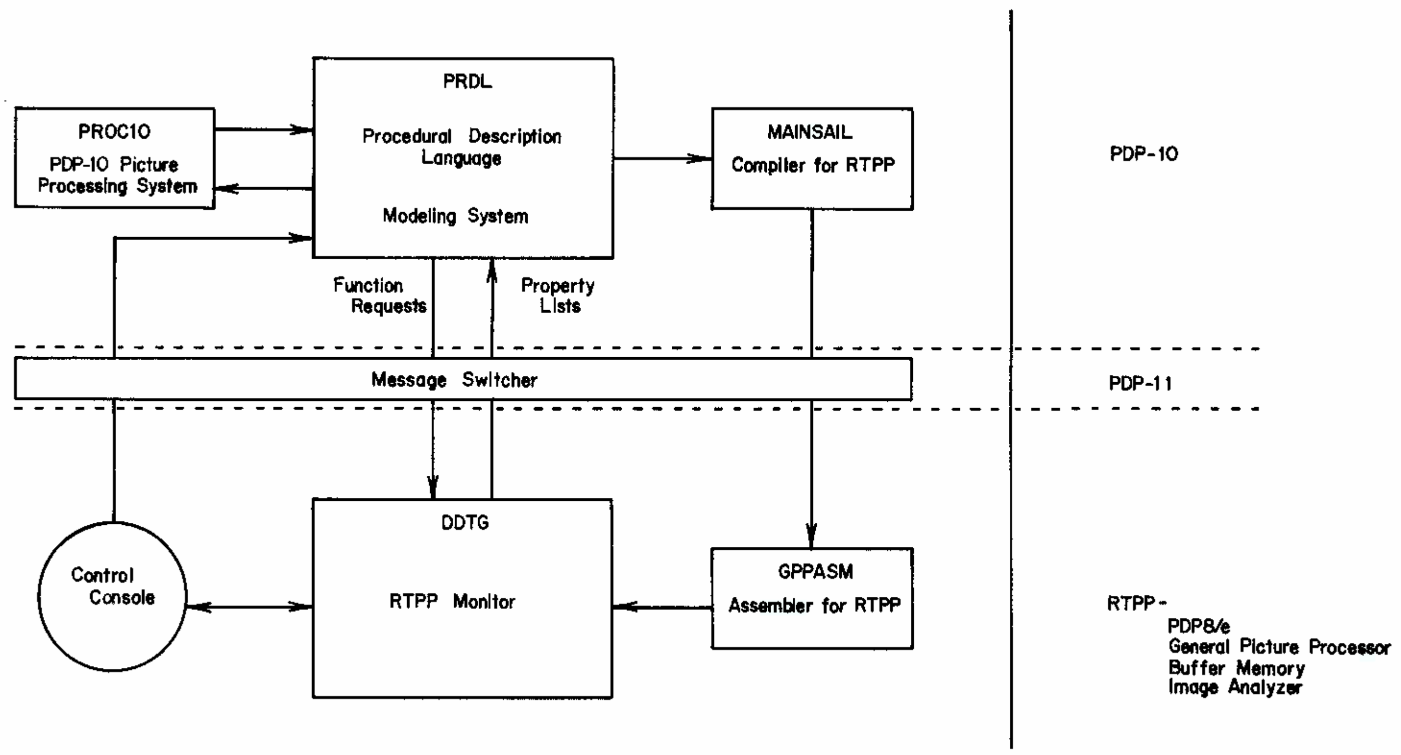

Figure 4. CELMOD system block diagram

CELMOD is composed of two major software systems running on different machines. The PRDL program runs on a DECSYSTEM-10 and is used to model cell images assuming the features are extracted. The RTPP has a resident monitor called DDTG which executes functions requested by the PRDL system and returns feature lists. The user sits at the RTPP control desk communicates with PRDL through a GT40 line graphics-teletype video terminal. The microscope data are visible on the RTPP display. The connection between the two systems is made via a PDP11/20 message switcher which is an operationally invisible high speed link between the two systems.

1.6 DDTG - the RTPP debugger/monitor

DDTG [Lem76a], a monitor/debugger is constructed for user and/or computer control of the RTPP. Functionally, DDTG is more than a simple combination of monitor and debugging facilities. As the RTPP operating system, it is required to interpret direct (i.e. via control console) user commands, or a string of commands generated by user-PDP10 interaction. In addition, it is required to provide access and control (at machine language word level) of major memory structures (PDP8e core, GPP program memory, GPP general register memory (GR) and buffer memories). It provides full control for a variety of image acquisition and low level analytic peripheral devices (e.g. Axiomat microscope stage, focus, etc. the Quantimet 720 and a variety of its plug in modules, a special mirror scanner, a sonic tablet, and a rapid scan spectrometer). Display control functions of DDTG also include the Dicomed and Quantimet displays.

DDTG has the ability to store, retrieve and execute (from PDP8e disks) PDP8e and/or GPP programs (user or PDP10 generated) (cf. Figure 4). Since in stand alone mode, DDTG is to be used as much by biologists as by computer scientists, the user interface allows many high level and seeming English command constructs. This in turn permits easy construction of understandable control programs for stand alone use, exploration and debugging at a very high level.

DDTG, written in standard PDP8e FortranII-Sabr consists of 4 major parts: an interpreter, parser, symbol table, and a large set of worker routines. The last include such features as loaders for the various component computers of the RTPP, as well as extensive and flexible disk I/O routines, a wide variety of stylized data structures.

DDTG has capabilities which allow it to load and run programs in the RTPP and to monitor their activity. An extensive set of commands are implemented to facilitate image data acquisition and display, and the running of small picture operation programs called "special segments". The General Picture Processor (GPP) portion of the RTPP will, in performing picture operations, require special segment support from DDTG. In addition, mechanisms are available for RTPP program interaction with OS/8 in order to facilitate the implementation of image processing programs (exclusive of DDTG) to process DDTG produced data.

1.7 RTPP Compiler/Assembler - MAINSAIL/GPPASM

The RTPP will have access to a very powerful cross-compiler MAINSAIL [Wil75] and a low leve assembler GPPASM [Lem76b]. MAINSAIL will run on the PDP10 and compile a dialect of PDP10 SAIL which includes teletype I/O and records but not LEAP or file structured I/O. As the initial RTPP has no floating point hardware, the initial MAINSAIL to be used has no real arithmetic constructs. Registers (such as line buffers Ij(k)) will be represented as reserved symbols (Ijk). MAINSAIL will output assembly language source code for GPPASM.

GPPASM is a low level assembler which produces non-relocatable load modules. It has labels and limited arithmetic capabilities as it was designed to run efficiently on a PDP8e. This last aspect is important for the debugging and maintenance of the RTPP. The GPPLDR which loads the absolute binary files produced by GPPASM is part of DDTG. GPPASM permits the use of REQUIRE <file> LOAD or SOURCE statements and thus a run time library can be assembled or loaded with main programs.

The GPPASM grammar was designed to be easily parsed by a simple finite state acceptor with a small auxillary stack. The assembler has either 2 or 3 passes depending on whether an assembly listing file is to be generated. The first pass incorporates declaration, PM and GR label resolution; the 2nd pass generates the PM and GR output code segments while the 3rd pass optionally generates an assembly listing. The PDP8e special (PAL8) segment is stripped out during the 1st pass and is later assembled by a modified version of PAL8 and concatinated with the PM and GR segments. The GPPLDR in DDTG is able to analyze and load such GPP binary load files. The BNF grammar for the GPPASM assembly language is given in Appendix E.

1.8 GPP Microprogram Assembler - GPPASM

The actual GPP implementation employs Microprogram control discussed in detail in [Carm77]. The microprogram is stored in a 128-bitx8K RAM by the PDP8e. Microprograms are assembled on the PDP8e also using the GPPASM assembler [Lem76b]. The microassembler is discussed in more detail in Section 5.1.1.

SECTION 2

Quantimet subsystem

The Quantimet 720 [Fish71] is a modular low level image processor which is used primarily as an I/O device rather than as an stand alone image processor. In addition to the basic scanner and display modules of the QMT, other hardware modules are used to accomplish elementary functions such as object area as determined by thresholded detected video: perimeter, number of objects in a field, integrated density under a mask, and several intercept properties. The Quantimet functions are controlled and programmed by the PDP8e minicomputer or may be used as a stand alone device.

The Quantimet part of the Real Time Picture Processor consists of the following Imanco Quantimet 720 modules:

Plumbicon (or vidicon) non-interlaced TV raster scanner

System Control module

System Display module

Variable Frame and Scale module

Standard Detector module

Digitizer/Densitometer module

1-Dimensional detector module

Amender module

Standard Computer module

MS3 Computer module

2 Function Computer modules (with Density option)

Light pen module

Classifier Collector module

The Classifier Collector module is added primarily for maintenance of the Function Computer modules.

2.1 Quantimet modules

The basic functions of these modules are enumerated here as an aid in understanding the design of the real time picture processor system. For further details see [Fish71]. The basic design for some of the Quantimet modifications was done initially for the NCI Grain Counter-I [LipL74].

2.1.1 Scanner - System Control module

The TV scanner (either a Plumbicon or vidicon camera) and system control produce a 10.1/second 880x720 digitally derived TV image. Actually, a smaller window is used which is called the big frame consisting of 860x680 lines. Inside the big frame is still a smaller frame called the live frame. Data inside the live frame is the data which is used by the rest of the Quantimet. The live frame is derived from many sources, some of which are the variable frame and scale, detected video, light pen mask output, buffer memory derived detected video and computed video of the various modules.

The TV camera is physically connected to the Zeiss Axiomat microscope through an IPU designed and constructed mechanical interface. This interface includes a Zeiss electromechanical shutter which is closed either by the computer (which can also open it) or by an automatic shut-off circuit which senses too much light entering the Quantimet camera. This latter feature prevents camera damage with too much light. The interface also has room for several neutral density filters so that the light levels can be balanced for running both the galvanometer scanner and the Quantimet camera at the same time.

2.1.2 System display module

The system display module is used to mix the system input video with the output display signals from the Standard computer, MS3 computer, Amender, Frame and Scale, 1-D detector, Digitizing detector, Function computers, Classifier Collector, and light pen modules. These display signals will either fully intensify or partially blank the system display. This mixed video signal is then displayed on a 10.1 rasters/second orange phosphor TV monitor. This phosphor is designed for slow decay times to reduce flicker. However, it does so at cost to the gray scale resolution.

The RTPP will be able to blank out the scanner video and substitute synthesized video in its place inside of buffer memory 256x256 pixel windows. This is discussed in more detail in Section 3 on the buffer memories. The Quantimet display cursor and mask register displays are added to the Quantimet display inputs (Scale mixer knob and Guard mixer knobs respectively).

2.1.3 Variable Frame and Scale module

The Variable Frame and Scale module generates a live Variable Frame by generating a rectangular computing window specified by (hor-position, hor-size, vert-position, vert-size). When in variable frame mode, any data inside this window will be enabled for detection by the Quantimet hardware; otherwise detection takes place within the standard live frame.

2.1.4 Detector modules

There are three detector modules: the Digitizer/Densitometer, the 1-D detector and the Standard detector modules. The Digitizer can be used as a detector (high speed comparator) to transform the analog video into a detected/not-detected digital signal according to the selected threshold values. The 1-D detector performs auto thresholding in 1 dimension (X-axis) to detect objects in a background phase. The Standard detector performs simple thresholding. The Digitizer and 1D detectors have a 64 gray level range while the Standard Detector maps up the white to black range into 4096 divisions. The thresholds in the Digitizer/detector and Standard Detector have been modified so as to be able to be loaded by the PDP8e.

2.1.5 Light pen

The light pen module will generate a mask of a detected object selected with the light pen. There is a restriction that the object be vertically convex so as to generate the complete mask.

2.1.6 Digitizer/densitometer

The Digitizer/densitometer also functions as a densitometer. It integrates detected gray scale data in the range of [0:63]. It does this by using a fast 125 nanosecond A/D converter and taking an optional log of the signal. It gives relative density directly. Using this module one can perform densitometry when suitably calibrated.

2.1.7 MS3 Computer

The MS3 computer module uses the detected video from the detector modules to compute global area, perimeter, intercept and count. The "pattern recognition" mode of operation activates two Function computer modules for acquiring object specific data.

2.1.8 Function Computer module

The Function computer computes for each detected object area, integrated gray-value or log-gray-value, perimeter, vertical projection, horizontal projection, horizontal feret, and vertical feret. An object is marked by the occurance of its anti-coincidence point (ACP). Data is not acquired for partially represented objects (in which the ACP falls outside of the live frame).

2.1.9 Classifier Collector module

A Classifier Collector module may be used to accept 2 Function Computer module outputs for stand alone operation. It is also able to compare function values from the function computer output against a range of values to determine whether the ACP (object label) should be used.

2.1.10 Standard Computer module

The Standard Computer module is similar to the MS3 Computer except that it does not compute perimeter and has no pattern recognition mode.

2.2 Microprogramming QMT modules

Many of the QMT modules mentioned above may be microprogrammed to select either modes of operation for functions to compute. This is done by enabling a rear status register on each of the modules called the "programmer" input. On the RTPP this is done by loading, for each QMT scan, a program word which consists of eight 12-bit status words called the Quantimet Program registers denoted QPRCG1 through QPRCG8. These registers and their allocation in the Quantimet subsystem is discussed in Appendix C.3.

2.3 PDP8e control of the Quantimet

The Quantimet may be controlled by the PDP8e in a single step data acquistion mode. When the System Control module 'Continuous/Auto' switch is put in 'Auto', it is placed under control of the PDP8e. Essentially, PDP8e control is effected by manipulation of the eight 12-bit static programming words and the QSTAT status register. The PDP8e STQMT instruction will start the Quantimet data acquistion when the Quantimet scanner reaches the start (top) of the next scan frame. The QMSKP skip instruction can be used to test when the data acquisition is done. At this time, the whole field Quantimet data is available to the PDP8e in the QDAT1, QDAT2, and QDAT3 input instructions which will read 7 BCD digits (LSD to MSD) into the PDP8e accumulator. This whole field data is displayed automatically in the left Quantimet display. The right Quantimet display may be loaded by the PDP8e load instructions LQDT1, LQDT2 and LQDT1 (MSD to LSD).

Function computer data as well as ACP (x,y) coordinates may be acquired in a single Quantimet scan through a 1024 word (69-bits/word) shift register. This is described in more detail in Appendix C.1.2.

2.3.1 Quantimet status register - QSTAT

A status register QSTAT controls operation of the Quantimet/PDP8e interface and various options on the Quantimet. This control includes enabling the Frame and Scale control desk switches, shift register data acquistion system for Function Computer data, standby and camera shutter activation. These are discussed in Appendix C.1.7.

2.4 The mask register module

Detected video is used to generate a 720 picture line Quantimet mask of entry and exit line intercept positions stored in a hardware Mask register. This Mask register can be used to supply the QMT with a detection region mask by reading the mask as the QMT goes through a scan. The mask register is a (1024 word) RAM which is loaded either from with the QMT detected video on a command from the PDP8e (GETMSK) or from the PDP8e directly. On the PDP8e issuing the GETMSK instruction, detected video is used to determine "DET-ENTRY" and "DET-EXIT" for the entire QMT live frame.

When the mask register is not being loaded, it cycles with the QMT and outputs a (QPROG selected) mask of either variable frame, mask or various logical comginations of variable frame and mask (see QPROG2[0:3]). The QPROG2 selected mask display intensity is controlled by the display "guard" knob. A display of the Mask register output alone (without the frame and scale) is also available and is selected by QPROG7[0:1]. The display intensity for this display is controlled by the display "scale and figures" knob.

The addressing in the mask register is the same as that of the frame and scale window. The QMT VTRIG, SYNC, (HTRIG), and Clock signals are used to generate the actual addresses. The entrance/exit acquisition algorithm works as follows. When detected video first goes true (enter a solid blob) the "X" coordinate at the Y'th line inside the frame is entered into the Y'th (0 to 719) "DET-ENTER" of the mask register. When the detected video first goes false again (leaving a blob), the "DET-EXIT" "X" coordinate is entered for the Y'th line. By default a line's "DET-ENTER" when "getting" a mask is assumed to , be 1023 before a line is processed. This takes care of the case where no data or only entry data is present. Note that only the first object in a line is detected and that objects with concave inward tops or bottoms will not be acquired properly because horizontal slices of such objects have multiple entrance/exit points.

The mask register can be loaded or read from the PDP8e. The main use of the mask register would be in masking QMT data with synthesized masks or in accessing actual perimeter X-Y coordinates.

Control of the mask register from the PDP8e is done with the GETMSK, MSKADR, RMASKE, RMASKX, LMASKE, and LMASKX PDP8e instructions. GETMSK enables the acquiring the next QMT detected video mask into the mask register. Now doing a STQMT command starts the actual mask acquisition from the input detected video. MSKADR loads the line number for subsequent I/O. Then RMASK- and LMASK- do I/O on that MSKADR's entrance/exit pixels.

Note that RMASK- and LMASK- are allowed at any time except during mask acqusition. the mask generated for an object will be a fairly good approximation to its boundry given that the object has no concave upward or downward regions. The following OS8 Fortran II program will get a mask from the detected video inside the current frame and save it in the PDP8e memory.

```
        DIMENSION IENTER(720),IEXIT(720)
S       GETMSK /enable a mask to be accessed
S       STQMT /start mask acquisition
S NOTDONE,      QMSKP
S       JMP NOTDONE
        DO 100 I=1,720
S       TAD \I
S       MSKADR
S       RMASKE
S       DCA \JENTER
S       RMASKX
S       DCA \JEXIT
        IENTER(I)=JENTER
100     IEXIT(I)=JEXIT
```

2.5 QMT cursor

An X-Y cursor is available which can be loaded from the PDP8e using the same coordinate system as the Frame and Scale and mask register devices. It appears as a 10 pixels line segment to the right on the actual (x,y) position on the "scale" display control. It is programmed in DDTG software to be used with the Graf-Pen to track the pen on the QMT screen whether the pen is actually using the data or not. To make the cursor disappear, load XP or YP. The PDP8e commands LDXP and LDYP load a binary coordinate pair into the cursor controller. The cursor display intensity on the Quantimet display is controlled by the "scale and figures" knob.

SECTION 3

Buffer memory

The RTPP is designed to use up to eight gray scale buffer memories (BM). The memories may be simultaneously selected by the PDP8e for posting of images (displaying on the Quantimet display) or acquiring QMT scanner data. However, there is a restriction that no two active BM windows can intersect during display or scanning as data will be lost from one of the memories.

A BM consists of 65K (256x256 pixels) 16-bit/words with gray scale data being stored in both the high and low 8-bit bytes of a 16-bit word. Binary detected video images are stored as signed 8-bit bytes. Since each BM is synchronized with the QMT independently, up to 512 X 1024 pixels "could" be posted in real-time (if the Quantimet display were large enough). Usually, a 256x256 window will be displayed and the other BMs may contain variants of that displayed window. One may select the high or low 8-bit bytes in a given buffer memory. Each 8-bit slice may then be used to store separate images or alternatively, for example the high and low 8-bit fields may be used to store the real and immaginary portions of a Fourier transform.

Synthesized Quantimet video

The QMT video (level adjusted by the system control module) is digitized using the fast A/D converter to 8 bits (using 8-bits of the 9-bit Computer Labs model 7910 10MHZ A/D). This digitized image is them multiplexed digitally with the buffer memory data to be synthesized as Quantimet input video - that is substituting buffer memory data for Quantimet TV scanner data. The digitized video is then resynthesized using a fast D/A converter (Computer Labs RDA-0815A, 15 MHZ).

Synthesized video, selected for display posting, is substituted in the window (Quantimet active frame) [[XB:XB+255],[YB:YB+255]] for the QMT scanner video input to the rest of the QMT. This results from our modification of the video train which allows the QMT to process synthesized video!

In order to synthesize QMT video, the BM first synchronizes with the QMT. That is, it waits until the QMT reaches the (XB,YB) pixel and then starts dumping 256 pixels into the digital video multiplexer. This digitized video is then converted with a fast D/A to analogue video for Quantimet video input. Successive lines are then synchronized similarly.

GPP use of BMs

The GPP can perform I/O with the BMs. That is, BMs are used for storing and retrieving intermediate pictures while doing picture processing. This is done synchronously by line after the line's row or column address is specified by the GPP LINE instruction. The GPP can also random access the BM in single word or neighborhood mode.

It is possible for two different devices to access two different BM's simultaneously under certain conditions. The eight BMs are divided into parallel memory systems (group A - BMs 0 to 3, and group B - BMs 4 to 7). Provided each device access is to a different group of 4 memories then parallel access is possible. The BM access priorities from highest to lowest are: BM input from QMT scanner, GPP I/O, PDP8e I/O, BM posting on the QMT display.

If two devices require I/O on the same BM group memories simultaneously, then the highest priority device obtains access (BM0). For example, if the GPP requests I/O during a posting on the QMT sequence, then the "would be" BM posted data becomes the actual QMT scanner data. If the scanner data were a dark object then the resulting posted image would have light holes in it representing a higher priority access conflict.

BM scan and display selection

Two types of images may be acquired: 8-bit grayscale and binary images. Binary images are stored in the buffer memories as signed 8-bit bytes. Images are stored and used by selecting a high or low byte to be accessed. PDP8e commands GETA, GETB, POSTA, and PCSTB perform these control functions.

They all load 12-bit command registers (from the PDP8e AC) which selects which byte to use, which buffer memories to enable, and whether to perform gray scale or binary (leading bit of a byte) I/O. The actual scan is started with the STQMT operation. The command register format (the same for all four GET-/POST- instructions) is as follows:

```
[0:3] - (0) use low byte for pix, high byte for binary mask
        (1) use high byte for pix, low byte for binary mask

[4:7] - (0) use gray scale data
        (1) use binary mask data

[8:11] - (0) don't select a BM
         (1) select a BM
```

BM window and scan acquisition

A buffer memory can accept a 256 X 256 x 8-bit gray scale image from the Quantimet video (digitized via Computer Labs 8-bit A/D) starting at the upper left hand corner coordinates denoted (XB,YB). This is done in less than one QMT scan cycle of 0.1 second. During this time the BM group is unavailiable to both the GPP and the synthesized video generator But other BM I/O operations, GPP or PDP8e I/C, may take place. The lesser priority operation will be disconnected only during the part of the horizontal scan line that the BM recieves data from the QMT. All other lines in that BM group during this 'GET' function are available for lower priority I/O except posting.

At the end of acquiring the QMT image, the BM is disconnected from the digitizing video channel. During a scan, the BM accepts 256 sequential pixels/line starting at the X coordinate XB of each line for each of 256 lines beginning with line YB. The BM window so defined is determined by comparing (XB,YB) with the real-time (xreal,yreal) coordinates. Each BM has a pair of window coordinates (XB(i), YB(i)) which may be loaded by the PDP8e. The eight window coordinate pairs are loaded from the PDP8e by the commands (BMXi,BMYi) where i=[0:7].

In addition, the detected Quantimet video may be acquired as a binary mask into buffer memories using the GETA/B command with the appropriate bits [4:7] set. It may be inserted into the Quantimet frame input using the POSTA/B instruction with the appropriate bits [4:7] set. The resulting mask may be used for further Quantimet data acquistion.

It should be noted to avoid confusion that BM window positions are independent of the frame and scale window position.

PDP8e accessing of BM data

The PDP8e can transfer up to 4K PDP8e 12-bit words using direct memory accessing (DMA) to/from a BM. Thus the transfer of a complete BMs contents would take several DMA transfers. Four PDP8e/BM DMA packing modes are available, all of which optimize the 12/16 bit word size differential. The first 2 modes transfer three 8-bit bytes (either high or low byte) in three BM words into two (OS8 packed format) PDP8e words. The third mode transfers three 16-bit BM words in four (OS8 packed format) PDP8e words. The fourth mode transfers 16-bit BM data in two PDP8e 12-bit words in packed sign-extended EAE (extended arithmetic element of the PDP8e) double precision format so that the PDP8e can do arithmetic on full BM words without extensive packing and unpacking.

3.1 Physical BM memory addressing

Although it is unnecessary to understand the physical implementation of the BMs to use them, the underlying structure is presented here for those who are interested. The 19-bit BM address specifies a buffer memory location and may be broken down into the concatination (&) of subaddresses: UBM[0:2]&YXADDR[0:15].

UBM[0:2] selects a BM unit.

Within a BM, YXADDR selects the data:
YXADDR[0:15]=YADDR[0:7]&XADDR[0:7].

Then the

row address = YADDR[0:7],
Column address = XADDR[0:7].

Each buffer memory is divided into four physical memory cards. YADDR[0:1] selects the memory card. The BM word size is 16-bits. The memory cards are four way interleaved so that 64-bits of data may be transfered in one 500 nsec memory cycle. The buffer memory is constructed in such a way that only four 16-bit words of data addressed modulo XADDR[6:7]=00 are obtained in one memory cycle. Therefore, random addressing of BM words requires 500 nsec/word minimum; whereas horizontal raster or line by line transfer (i.e . continuous incrementing of YXADDR) will result in an effective 125 nsec./word transfer rate.

3.1.1 BM controller accessing priorities

During a BM-QMT scan acquisition, the posting of data to the QMT is discontinued during that entire scan for that respective BM group. Posting of BM data from one BM group may take place during the same scan that acquisition of QMT video data by the other BM group. In fact, video data being posted by one BM group may be loaded into the second BM group during the acquisition scan.

The (XB(i), YB(i)) position for each buffer memory is used to select where the data shall be taken and the image posted on the display. This allows the operator to position the transform window exactly where it is needed to operate on data. The hardware priority networks resolves conflicts if two or more BM windows intersect (in case the software does not prevent their intersection).

If while posting BM video data on the QMT a conflict occurs with BM I/O other than acquiring QMT video, then during the duration of the conflict the origonal scanner video is posted rather than the BM video.

SECTION 4

GPP - General Picture Processor

4.1 GPP organization

The general picture processor, GPP, is primarily used to serially perform rapid local neighborhood processing on image data stored in the buffer memories. The GPP is organized as a 16-bit, 2s complement, pipelined, register to register transfer machine. It is dynamically microprogrammable so that real time implementation of complex register to register type machine instructions are possible. The GPP behaves as a PDP8e peripheral processor for which the PDP8e assembles, loads and starts programs, services requests for monitor operations, and stops.

The GPP is a reentrant machine, a program memory (PM) is distinct from a data memory (DM) or an extended data memory (XDM). Both the program and extended data memory may be addressed with a maximum 32-bit address. The data memory may be addressed with a 16-bit address.

A detailed description of the various data spaces and busses is given in [Carm77].

4.1.1 Microprogram memory - MPM

The GPP has six separate memory groups or "spaces". The first is the microprogram memory, MPM. Each of the addressed microprogram memory locations makes up a microinstruction. A single microinstruction theoreretically may have as many as 128 microcommands (thus a MPM word is 128-bits wide). There is a maximum of 8k microinstructions. The microprogram memory is loaded directly by the PDP8e through program register transfers.

As part of the MPM word, 32-bits are allocated to two 16-bit fields, MRA and MRB. These fields allow the microprogram to supply arguments when forming complex instructions.

4.1.2 Program memory - PM

The second memory space is the program memory, PM. Each of the addressed program memory locations makes up a GPP instruction. A GPP instruction is divided into four separate fields. These fields are called the OPR, P1, P2, and P3. The OPR contains the instruction operation. If used, P1 and P2 may contain the address of the two separate data source locations used by the operation. P3 may contain the address of the data sink location used to receive the result of the operation. The triple address instruction format allows the GPP to rapidly access two source operands, P1 and P2, in parallel, operate on them, and deposit the result in a sink operand, P3, with a single instruction. Each of the program memory fields, OPR, P1, P2, and P3, are 16-bits wide. Thus a PM instruction is 64-bits wide.

The OPR is divided into two subfields: the OPRgroup[0:8] and the OPRalu[9:15]. The most significant nine bits, OPRgroup[0:8], is used to reference the starting address of the instruction in the microprogram memory. This is done via a mapping memory (MM) discussed in the next subsection. The least significant seven bits in the OPR field, OPRalu[9:15], are used to select an arithmetic logic unit to be used as the operation of the instruction if any.

A 16-bit program field register, PFR, is provided for expansion of the program memory address space to 32-bits. There are a maximum of 64k instructions per program memory field. The GPP is initally being constructed with one 64k program memory at program field address zero. The program memory is loaded by the PDP8e (using PDP8e DMA). GPP instructions are microprogrammed not to alter data in the program memory thus guaranteeing reentrency.

4.1.3 Mapping memory - MM

The third memory space is the mapping memory, MM. This memory is used to map the instruction number, which is part of the OPR field in the program memory, to a microprogram starting address in the MPM. This starting address is used by the microprogram memory to execute the instruction requested by the program memory. The mapping memory is 13-bits by 1K words and is loaded directly by the PDP8e. The OPRgroup[0:9] can address the first 512 words of the MM. The second 512 words may be addressed by use of the APPLY GPP instruction which takes the 10-bit MM address from the P1 instruction field.

4.1.4 Data memory - DM

The forth memory space is the data memory, DM. Each data memory location is 16-bits wide. The data memory space is divided into two groups called the data memory I/O space and the data memory general register (GR) space.

All GPP input and output registers, and many internal special register are addressed via the data memory I/O space. The top 2k of the 64k data memory space is assigned to the data memory I/O space.

The data memory GR space is a fast (100nsec) interleaved memory addressed in the first 62k (63488 words) of the 64k data memory space. No other register's addresses are assigned in this 62k GR data space. The GR data space is interleaved so that each consecutively addressed location is mutually exclusive.

Both the GP and I/O data spaces are divided into mutually exclusive data memory modules and will be discussed in detail later in this section. The data memory is loaded by the PDP8e via PDP8e DMA.

4.1.5 Extended data memory - XDM

In order for the GPP to access large blocks of data used to store images or lists the fifth memory space, extended data memory, XDM, is provided. The XDM 16-bit words are addressed with 32-bits. The first 19-bits address the image buffer memories. The rest of the XDM is not initially implemented. Pointer registers enable the GPP instructions to directly address any word in the XDM. GPP instructions like GLINE, PLINE, GNBH, and PNBH are microprogrammed to transfer blocks of data between the XDM and the data memory, DM.

4.1.6 Program memory address push down list - PDL

The sixth memory space is the program memory address push down list, PDL. A 16-bit program counter is used to address the instruction in any field of the program memory. The 16-bit program field register, PFR, together with the program counter, PC, comprise the program memory address. This 32-bit address may be stored in a 32-bit by 1K word push down list (using the GPP PUSHJ instruction) for returning from a procedure call (using the GPP POPJ instruction). Normal GPP instructions can not read or write on this list. The PDL itself is not addressed in the data memory space but the stack pointer is and may be read.

4.1.7 Arithmetic logic units - ALUs

The ALU space is a set of arithmetic logic units, ALUs, and up to 32 ALU input data registers, ALUA(0-15) and ALUB(0-15). The ALUs are physically divided into groups which fit on a logic card. The ALU input data registers are duplicated on each of these logic cards as required and are addressed as part of the data memory I/O space. Each ALU logic card has a maximum of sixteen 16-bit output register ports called AOUT(0-15). Once an ALU logic card is activated by a microcommand, the cards respective output ports are available as data to be used in the data memory space. The ALU output ports are not addressed in the data memory space but are controlled by microcommands for loading on the address or data buses.

Normally the 7-bit OPRalu field of the program memory will select the ALU to be used during an instruction. It is also possible for the microprogram memory to select the ALU via 7-bits of a microcommand field, MRB[8:15]. This feature allows microprograms to make maximal use of GPP ALUs as required.

4.1.8 Buses: D1 D2 A1 A2 CB MHOLD ALU

In order to transfer information between various GPP registers two sets of 16-bit data buses, D1 and D2, and address buses, A1 and A2, are used. Any location in the data memory, DM, may be addressed by either address bus and its contents deposited onto either data bus to be stored at a second location in the data memory. The program memory 16-bit fields (P1, P2, and P3) may be deposited onto any of these four buses thereby allowing the program memory to be used as data or an address for data. Two 16-bit microcommand fields, MRA[0:15] and MRB[0:15], may drive any one of these four buses. This allows microinstructions to contain data or to address data memory locations.

A single bit conditional bus, CB, is used as a modifier to many of the microcommands. For example, to write in the data memory first requires a microcommand. The microcommand may also require the CB bus to be "false". The CB bus is usally driven by an output from one of the ALUs or from an active data memory space register.

The MHOLD bus is also a single bit which is used throughout the GPP. If asserted, the GPP does not execute the present set of microcommands last fetched from the MPM. When the MHOLD bus is unasserted, the microprogram sequencing continues. This signal allows different processing times for ALUs or memory elements than the basic GPP microprogram clock time.

A seven bit bus, ALU, is used to route the least significant seven bits of the OPR[0:15] field of the program memory or seven bits from the microprogram memory (MRB[8:15]) to the ALUs. The MFB[8:15] data allows for the selection of a particular ALU (other than the one specified by the program memory OPR field) during the current instruction.

The D1, D2, A1, A2, CB, MHOLD, and ALU buses are commonly referred to as the G-bus.
The following table outlines the GPP organization as discussed thus far in this section.

Table 4.1. GPP organization

I GPP memory space
- 1) Microprogram memory, MPM
  - a) 128-bits wide by 8K words
  - b) Each word ia a microinstruction
  - c) Each bit is a microcommand
- 2) Program memory, PM
  - a) 64-bits wide by 64K words
  - b) Each word is a GPP instruction
  - c) Divided into four fields, OPR, P1, P2, P3
  - d) OPR field divided into subfields, OPRgroup and OPRalu
- 3) Mapping memory, MM
  - a) 13-bits wide by 1K words
  - b) generates microprogram starting address
- 4) Data memory, DM
  - a) 16-bits wide by 64K words
  - b) General registers, GR = 62K words
  - c) Input-output registers and special internal registers = 2K words
- 5) Extended data memory, XDM
  - a) Addressed with 32-bits
  - b) 16-bit words
  - c) Direct addressing via pointer registers
  - d) Buffer memories addressed in first 19-bits
- 6) Program memory address push down list, PDL
  - a) 32-bits wide by 1K words
  - b) Stores the program field and program counter contents during procedure calls

II GPP ALU space
- 1) Set of ALUs (add, subtract, or, and, multiply, etc.)
- 2) Data space ALU input registers ALUA(0-15) and ALUB(0-15)
- 3) ALU space ALU output ports, AOUT(0-15)

III GPP buses
- 1) All buses referred to as G-bus
- 2) Data buses, D1 and D2
- 3) Address buses, A1, A2
- 4) Conditional bus, CB
- 5) Micro hold bus, MHOLD
- 6) ALU selection bus, ALU

4.2.3 I-buffer control: XCLK XCLKB YCLK YCLKB XRST

Several active data memory registers are used to control functions in the triple-line I-buffers. When an argument is moved into one of the XCLK, YCLK, or XRST registers an action discribed in the following table 4.2 is executed. The MOVE instruction may be used to load arguments into these registers for their respective executions. The active data memory registers, XCLKB and YCLKB, also discribed in the following table, execute their respective function as a result of the argument loaded into them and can then cause a branch to a new program memory location. The MOVEB instruction with the proper argument is used to load these registers.

Table 4.2 Triple-line I-buffer control registers

P3 Argument

IBMR

P1 argument notation is:
<64 sq pixel image> - bits 14, 15 = 00
<256 sq pixel image> -bits 14, 15 = 01
<512 sq pixel image> - bits 14, 15 = 10
<1024sq pixel image> - bits 14, 15 = 11

XRST

P1 argument notation is:
<I1,I2,I3> - bits 10,11,12
<X-1,X,X+1> - bits 13,14,15

Reset selected X counters in I1, I2 or I3 to

X-1 = -1, X = 0, X+1 = +1.

XCLK

P1 argument notation is:
<left/right> - bit 9
<I1,I2,I3> - bits 10,11,12
<X-1,X,X+1> - bits 13,14,15

Increment (bit 9 of P1 = 0) or decrement (bit 9 of P1 = 1) the specified I-buffer X dynamic address vectors.

YCLK

P1 argument notation is:
<I1,I2,I3> - bits 10,11,12
Increment the specified line buffer Y dynamic address vector register and Y line counter.

XCLKB

P1 argument notation is:
<left/right> - bit 9
<I1,I2,I3> - bits 10,11,12
<X-1,X,X+1> - bits 13,14,15

Increment (bit 9 of P2 = zero) or decrement (bit 9 of P2 = 1) the specified I-buffer X dynamic address vectors. Then, if all X .NE. 64, 256, 512, or 1024 (selected by IBMR) then PC<==P2.

YCLKB

P1 argument notation is:
<I1,I2,I3> - bits 10,11,12
Increment the specified I-buffer Y dynamic address vector register and Y line counter. If any selected Y counter .NE. 64, 256, 512, or 1024 (selected by IBMR) then PC<==P2.

4.2.4 Extended data memory interface registers

In order to interface the GPP data memory with the extended data memory several registers addressed in the I/O data memory space are provided. The XDM interface may transfer data in block form which requires a current address register of 32-bits. Registers XDMCAH and XDMCAL are used to store the XDM current address high and low 16-bits respectively. The number of words transferred per block (maximum of 1k) and control codes for the type of I/O is stored in register XDMWCC (XDM word count and control). Data writen on the XDM from the GPP D1 or D2 buses is stored in WXDM1 and WXDM2 (write XDM1,2) registers respectively. Data received from the XDM to be stored in the DM via data buses D1 and D2 is stored in RXDM1 and RXDM2 (read XDM1,2) registers respectively. An address register, XDMAR (XDM address register), may be used to store the data memory address of the transferred word. Table 4.3 summerizes these registers and their meanings.

To receive data from the XDM the XDM current address registers (XDMCAH and XDMCAL) are first loaded with the starting address of the XDM transfer data. When the word count and control are loaded into the XDMWCC register the transfer is initiated. To receive data from the interface the RXDM register is read which will stall the GPP (GPP microprogram clock disabled) until the requested data is provided by the XDM. When the last XDM data word of the block transfer is stored in the RXDM register and the read by the GPP the CB bus will be asserted to allow the GPP microprogram to branch. This signals the microprogram that the black transfer is complete.

To send data to the XDM the XDM current address registers are first loaded with the starting address of the XDM transfer data. The word count and control codes are then loaded into the XDMWCC register and the transfer is started when the WXDM register is loaded with DM data. If the XDM is busy when the WXDM is loaded then the GPP will stall. The last data word stored in the WXDM register will cause the CB to be asserted thereby allowing the microprogram to branch signaling the end of the block transfer.

Table 4.3 XDM interface registers

XDMCAH	- High order 16-bit XDM current address.
XDMCAL	- Low order 16-bit XDM current address.
XDMWCC	- Argument notation is:
	<word count> - [6:15]
	<image size> - [4:5]
	00 = 64 sq pixels
	01 = 256 sq pixels
	10 = 512 sq pixels
	11 = 1024 sq pixels
	<axis projection> - [3]
	0 = Horizontal axis
	1 = Vertical axis
	<word size> - [1:2]
	00 = 16-bit
	01 = low byte of XDM
	10 = high byte of XDM
	<input/output> - [0]
	0 = output to XDM
	1 = input from XDM
RXDM1	- Read XDM data via data bus D1.
RXDM2	- Read XDM data via data bus D2.
WXDM1	- Write XDM data via data bus D1.
WXDM2	- Write XDM data via data bus D2.
XDMAR	- XDM address register used as pointer into the data memory.

4.2.5 GLINE and PLINE execution registers - GLINER PLINER

Two registers GLINER and PLINER are used to pass needed information to the XDM interface when executing the GPP instructions GLINE (get image line from XDM) and PLINE (put image line on XDM). These registers are addressed in the I/O data memory space and are loaded with the selected I-buffer and Y line (Y+1, Y, or Y-1), the selected word size (16-bit, high byte, or low byte), and selected horizontal or vertical axis projection.

When the GLINE or PLINE GPP instruction is executed the XDMCAH and XDMCAL registers are loaded first, then, the GLINER or PLINER register is loaded which transfers its argument plus the the state of the I-buffer mode register to the XDMWCC register. This would start the XDM I/O if the GLINE instruction were executed. If the PLINE instruction were executed the I/O would start when the WXDM register is loaded.

4.3 GPP run time registers

Several run time registers for GPP status and I/O data memory space write protection are presented in the following subsections.

4.3.1 GPP run time status registers - GSTAT1 GSTAT2

Two GPP run time status registers, GSTAT1 and GSTAT2, are used to alert the PDP8e of special GPP conditions. The two registers are 12 bit and are set by special hardware in the GPP. The PDP8e may read the two registers which are not addressed in the data memory space. The status registers are defined in table 4.4.

Table 4.4 GPP status registers, GSTAT1 and GSTAT2.

```
GSTAT1 bits:
        [0] = 1 -> PM parity error.
        [1] = 1 -> GR parity error.
        [2] = 1 -> MPM parity error.
        [3] = 0 -> GPP RUN-HALT FF is set to HALT.
        [3] = 1 -> GPP RUN-HALT FF is set to RUN.
        [4] = 1 -> GR address base overflow error.

GSTAT2 bits:
        All spares.
```

4.3.2 GPP run time write protect registers

Two registers are used to protect special I/O data memory space registers from being written on by GPP instructions. The first register, write protect enable register, WPER, is used to specify which I/O data memory space registers is to be protected. The second register, write protect status register, WPSR, is used to signal the PDP8e of an error; i.e. the GPP tried to write on a protected register. The write protect status register is 12-bits and can be read by the PDP8e. The write protect enable register, WPER, is loaded by the PDP8e and is also 12-bits. Neither of these registers is addressed in the data memory space. The definition of the registers is in table 4.5.

Table 4.5 Write protect registers, WPER WPST

```
WPST bits:
        [0] = 1 -> PDLCTR written.

WPER bits:
        [0] = 1 -> Enable PDLCTR write protection.
```

4.3.3 GPP parity error registers - PMPE GRPE MPMPE

The program memory, GR data memory, and microprogram memory is checked for parity errors on each 8-bits respectively. In order to determine where a parity error occured several registers are provided. Both the microprogram memory parity error register, MPMPE, and the GR data memory parity error register are 16-bits. The program memory parity register, PMPE, is 8-bits. These registers are read by the PDP8e and are not addressed in the data memory space.

4.5 Running the GPP

To operate the GPP the microprogram memory, mapping memory, program memory, and appropreate data memory locations are loaded by the PDP8e.

To load the MPM and MM the PDP8e first executes a GMHLT (GPP microprogram halt) and a GPPCLR (GPP clear). When the MPM and MM have been loaded the PDP8e executes a GMRUN (GPP microprogram run) command which starts the MPM at starting address zero. The microprogram memory executes a "monitor" program which listens for requested DMA I/O from the PDP8e or the GPP run flip flop set by the PDP8e.

The PDP8e loads the required program and data memory and then loads the GPP program counter, PC, with the starting address of the program in the PM. The PC is loaded by first loading the EXDMA words with a 16-bit address and executing the LPC (load PC) PDP8e command.

When the program counter has been loaded the PDP8e executes the GPPRUN (GPP run) command which starts the execution of program memory instructions at the starting address defined by the PC.

The PDP8e may then wait for communications from the GPP via the GIN and GOUT GPP registers or read the GPP status register. The status registers contains, in part, the state of the GPP run flip flop which was set by the PDP8e GPPRUN command.

The GPP may turn the GPP RUN-HALT flip flop off (MOVE arg,,halt) thereby halting GPP instruction execution by returning to the microprogram monitor. The GPP RUN-HALT flip flop set to HALT may be used as a signal to the PDP8e that the GPP program halted. The PDP8e will analize the argument in the halt register to see if the program has been normally completed or the GPP requests aid from the PDP8e.

The resulting GPP data, if any, may be transfered to the PDP8e. To run additional programs, only the program memory and required data memory need be loaded.

SECTION 5

Implementation of the RTPP

This section discusses the construction of the various electronic and mechanical components of the RTPP.

5.1 The GPP control - microprogram control

The GPP control logic is implemented as a microprogrammable controller. The microprogram is loaded into the microprogram memory by the PDP8e. The various bus and register enables, function enables etc. are driven by the microprogram controller output bus. This bus is 60-bits wide so that all necessary enables may be performed in parallel. The microprogram control store is a maximum of 4K words. Note that the microprogram memory (MCPM) is distinct from the GPP program memory (PM) although both are 60-bits wide. These microprograms are loaded into the microprogram control store RAMs through the PDP8e DMA. Thus experimentation with new instruction sets is easily performed as is debugging of the initial instruction set. A microprogram assembler MICROP will assemble microprograms on the RTPP PDP8e to facilitate rapid instruction set debugging. The grammar for MICROP is given here:

5.1.1 Microprogram instruction BNF grammar

A BNF grammar specification is given for the GPP microprogram control programs. As can be seen from the MICROP grammar, assembly consists of doing the inclusive-or of all <M> symbols in a statement up to the ";". Thus each <M> symbol has an associated bit in the microprogram controller output bus.

```
<microprogram>::=<microprogram><microstatement> |
                 <microstatement> | <microprogram>$

<microstatement>::=<label> <M-list> <address>; |
                 /<comment>; | <M>=<number>; | <origin def>;

<origin def>::=ORIGIN <number>

<address>::= GOTO <label symbol> | <label symbol>
<label>::=<label symbol>: | <null>

<label symbol>::=new symbol | <number>

<M-list>::=<delim> <M-list> | <M-list><delim> <M> |
                <M-list> <delim>

<delim>::= space | ,

<M>::=P1DABE | P1DBE | P2DABE | P2DBE | P3DABE | P3DBE |
              PCDBE | DRADBE | DRBDBE | ALUDBE | DRCDBE |
              DRCDAB  |  READ  |  WRITE  |  LODCTR | INCCTR |
              DECCTR | FC0000 | ... | FC1111 | BROVF | BRUDF |
              BRGT | BRLT | BREQ | BRLE | BRGE | JUMP | PUSHJ |
              POPJ | ...
```

The microprogram instruction will have a right justified 12-bit address field which is used for jumps (JUMP, PUSHJ, POPJ) and manipulating counters (LODCTR, INCCTR, DECCTR). Microprogram subroutines are performed using a 64 word 12-bit address stack.

5.2 Internal Control Logic Design

The method of Richards [Rich73] for designing complex controllers is used in various parts of the system including the Quantimet controller, BM controller, GPP microprogram controller, DMA controllers, etc. Richards employs a method of automatically defining sequential logical states such that a sequential synchronous counter implementation is easily realized. It is thus possible to easily and rationally construct finite state machine (FSM) controllers. To ease hardware debugging and maintenance of the RTPP, the "new" state is displayed in octal LED's on the cards where the FSMs reside.

5.3 Physical construction of the RTPP

The RTPP consists of several subsections as mentioned in Section 1.3. Much of the system can be bought ready made off the shelf. Other parts such as the GPP, buffer memory, control desk and QMT I/O to the PDP8e are specially built. We would like to minimize the amount of special purpose mounting hardware needed and thus are led to the implementation of the basic RTPP control and computational hardware as a set of rack mounted card files each 19 by 10.5 inches. These card files hold 16 double height (140 pin) high density wire-wrap cards made by Cambion. The wire-wrap cards plug into a semi-automatic wire-wrapable power back plane which holds two 70-pin semi-automatic wire-wrappable card sockets for each card.

The various buses can then be implemented by wire-wraping twisted pairs on these power back planes. Interconnection between card files is by plug connection with Scotchflex (3-M Corp.) cables directly to the 70 pin card sockets. Any cabling between wire-wrap cards is avoided by all connections being done through the backplanes. The wire-wrap cards and power back planes are being constructed outside of NIH already wire-wrapped with by pass capacitors mounted.

The Quantimet and control desk logic (RQC) is housed in one standard 19" wide, 6′ high cabinet, the buffer memories in another and the GPP in a third cabinet. These cabinets are located between the control-desk/Quantimet-display and the PDP8e cabinets.

5.4 Buffer memory implementation

The hardware of the BM is divided into three sections: 1) memory cards; 2) dual memory card controller; and 3) BM interfaces.

The memories use Cambion specially constructed semi-automatic wire-wrap cards. The Texas Instruments TMS4030 (N channel MOS 4096x1 bit 22 pin dynamic RAM) is used as the main memory element. Each card contains a total of 64 TMS4030 memory chips and locations for 35 16-pin TTL support chips. Four memory cards make up one memory and four memories are housed in one 16 slot Cambion card enclosure. There are a total of two card enclosures used to house the eight BMs.

The dual memory controller logic is contained on one Cambion wirewrap card. Both controllers are totally independent and may operate in parallel. The dual contoller contains the refresh logic for the dynamic RAMs and the priority logic to service a total of five I/O requests from various BM interfaces.

Each BM interface requires a minimum of three card slots to communicate with the dual BM address, data and control buses. Five separate interface locations or a total of 15 card slots plus the dual memory controller card make up the third 16 slot Cambion card enclosure. A fourth Cambion card enclosure is used for additional BM interface logic. All I/O on the buffer memories may be in 16-bit word or 8-bit byte modes.

SECTION 6

References

Carm74. Carman G, Lemkin P, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - II hardware implementation. J. Hist. Cyto. Vol 22, 1974, 732:740.

Carm76. Carman G, Lemkin P, Schultz M:RTPP - System Documentation, Vol I: Microscope, scanner and Quantimet Controller. NCI/IP Technical Report #13. In prep.

Carm77. Carman G, Lemkin P, Lipkin L, Shapiro B, Schultz M:Micropogram Control Architecture for the General Picture Processor. NCI/IPU Technical Report #22, April 22, 1977.

Dec67. Digital Equipment coproration:LINC8 small computer handbook. Maynard, Mass, 1967.

Dec71. Digital Equipment Corporation:PDP11/20 processor handbook, Maynard, Mass, 1971.

Dec72a. Digital Equipment Corporation:PDP8e and PDP8m small computer handbook. Maynard, Mass. 1972.

Dec72b. Digital Equipment Corporation:DEC system 10 assembly language handbook. Maynard, Mass. 1972.

Fish71. Fisher C:The new QUANTIMET 720. The Microscope Vol 19 No 1, 1971, 1:20.

JohnE70. Johnston E:The PAX II picture processing system. In [LipB70].

Kir69. Kirsch R:Computer determination of the constituent structure of biological images part I. NBS report 10 173, Dec 1969.

Lem72. Lemkin P:A simplified biological cell world model for question answering using functional description. Univ. of Maryland, Scholarly Paper #75, May, 1972.

Lem74. Lemkin P, Carman G, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - I systems design. J. Hist. Cyto, 22, 1974, 725:731.

Lem75. Lemkin P:A Literature Survey of the Technological Basis for Automated Cytology. Univ. Md. TR-386, June, 1975.

Lem76a. Lemkin P:Functional specifications for the RTPP monitor - debugger DDTG. NCI/IP Technical Report #2, Feb 1976.

Lem76b. Lemkin P, Shapiro B, Schultz M, Lipkin L, Carman G:GPPASM - a PDP8e assembler for the General Picture Processor. NCI/IP Technical Report #16, Dec 15, 1976.

Lem76c. Lemkin P, Shapiro B, Lipkin L:The CELMOD Biological Image Modelling System. NCI/IP Technical Report #14. In prep.

Lem76d. Lemkin P, Gordon R, Shapiro B:PROC10 - An image processing system for the PDP10. Operation and Description. NCI/IP Technical Report #8, Dec 16, 1976.

L7m7e. Lemkin P: BMON2 - Buffer memory monitor system for interactive image processing. NCI/IP Technical Report #21a, June, 1977.

LipB70. Lipkin B, Rosenfeld A (Eds):Picture processing and Psychopictorics. Academic Press, 1970.

LipL74. Lipkin L, Lemkin P, Carman G:Automated Grain Counting in Human Determined Context. J. Hist. Cyto. Vol 22, 1974, 755.

Rich73. Richards C:An easy way to design complex program controllers. Electronics, Feb 1, 1973, 107:113.

ShapB74. Shapiro B, Lemkin P, Lipkin L:The application of artificial intelligence techniques to biological cell identification. J. Hist. Cyto. Vol 22, 1974, 741:750.

ShapB77. Shapiro B, Lemkin P, Lipkin L:PRDL - Procedural Description Language. NCI/IP Technical Report #13, June 1977.

Thor70. Thorton J E:The control data 6600 - design of a computer. Scott, Foresman and Co, 1970.

VanL73. VanLehn K:Sail User Manual. Stanford Artifical Intelligence Laboratory memo AIM-204, July 1973.

Wil75. Wilcox C:MAINSAIL - MAchine INdependent SAIL. DECUS meeting, Languages in Review Session, 1975.

APPENDIX A

GPP instruction set

The instruction set of the GPP allows for not only the usual set of operators needed in a triple address type machine, but also specific operators oriented toward neighborhood processing and I/O image transfers. These include the following five classes of instructions for which examples are given in table A.1. Reference [Carm77] discusses the microprogram control structure in detail.

Table A.1 Examples of some RTPP instructions

```
1) Register-to-register transfers,
     MOVE P1,,P3       ; P1 to P3
     INC P1,,P3        ; increment P1 by 1 into P3
     ADDSB ,P2,P3      ; add P2 and ALUA(0) register,
             store in P3
     GESA P1,,P3       ; if P1 > ALUB(0) store P1 in P3
     FLOAT P1,,P3      ; float the single precision P1 to
             a 48-bit floating point P3
     DFLOAT P1,,P3     ; float the double precision P1 to
             a 48-bit floating point P3
2) P1 operated on by P2 to be deposited in P3,
     ADD P1,P2,P3      ; sum of P1 and P2 stored in P3
     MUL P1,P2,P3      ; product of P1 and P2 stored in P3
     AND P1,P2,P3      ; Logical AND of P1 and P2 stored in P3
     MOVBIT P1,P2,P3 ; bitset P3 from P1 under mask P2
     DSUB P1,P2,P3     ; Double precision P1 subtract double
             precision P2 stored in double precision P3
     FMAX P1,P2,P3     ; Floating point maximum of P1 and p2
             stored in P3
3) Conditional branch instructions
     EQB P1,P2,P3      ; if P1=P2 then goto P3 else do next
             instr.
     GTB P1,P2,P3      ; if P1>P2 then goto P3 else do next
             instr.
     DLTB P1,P2,P3     ; if double precision P1<P2 then go to
             P3 else do next instr.
     FLTB P1,P2,P3     ; if floating point P1>P2 then go to
             P3 else do next instr.
4) Control instructions
     PUSHJ ,,P3        ; procedure entry, stack return address
     POPJ ,,           ; procedure exit to stacked return
             address
     JUMP ,,P3         ; unconditional GOTO
5) I/O instructions.
     MOVE P1,,XRST   ; reset X dynam. vectors as a
             function of argument, P1.
     MOVE P1,,XCLK   ; incr. X dynam. vec. as a
             function of the argument, P1.
     MOVE P1,,YCLK   ; advance Y dynam. vec, as a
             function of the argument, P1.
```

A

```
BMIO P1,P2,LINE  ; read or write a 256 word line
        of the BM as a function of P1 and P2.
BMIO P1,P2,GETI1  ; Fetch a neighborhood into the I1
        line buffer as a function of P1 and P2.
MOVE 'PBM2,,P3  ; read BM2 datum into P3
MOVE P1,,'PBM2  ; write P1 into  BM2
MOVE GIN,,P3  ; read word from PDP8e channel
MOVE P1,,GOUT  ; write word to PDP8e channel
MOVE TIN,,P3  ; read GPP terminal input channel
```

Among the I/O instructions are whole line, local neighborhood, and single pixel transfers from/to the buffer memories. The latter type of instruction allows random access of the buffer memories. The line transfer allows entire horizontal or vertical 256 (16-bit) pixel lines to be transfered. The local neighborhood transfers allow random accessing of neighborhoods within the buffer memories in one instruction.

A GPP instruction cycle consists of fetching an instruction from the PM, incrementing the PC instruction program counter, and then executing that instruction. The contents of the PC is the index of the next instruction to be executed in the PM. The instruction is executed by fetching the two input operands, performing the operation, and then storing the result in the output operand if so directed.

A.1 Instruction groups

In order to obtain maximum speed, in-line microprograms are used to control all instructions. This requires redundant addressing of arguments but results in considerably greater speed that using microprogram subroutines. Therefore all instructions are divided into groups that have similar characteristics required by the microprogram control structure. Each of the instruction groups are divided into subsections in order to explain their properties.

A.1.1 Group subsections

Under the subsection, "NOTATION", is a representation of what registers are filled with what arguments and where the results are stored if any. A text description is presented under the subsection, "DESCRIPTION". The subsection, "GROUP ADDRESS MODE TABLE", presents the different types of program memory P field addressing allowed. If the P field has no prefix then normal addressing is assumed. Normal addressing is where the contants of the program memory P field is the address of the argument used in the instruction.

If the P field has a # as a prefix then immediate addressing is used. Immediate addressing is where the contants of the program memory P field is used as the argument by the instruction. If the P field has a ′ as a prefix then indirect addressing is used. Indirect addressing is where the contants of the P field is used as the address of the address for the argument used by the instruction.

Whenever the P field is used as the branch argument then normal addressing is used to denote that the contents of the program memory data will be used as the data loaded into the program counter, PC. Also whenever the P field is used as the branch argument then indirect addressing is used to denote that the contants of the program memory data will be used as the address of the data to be stored in the program counter, PC.

The table also contains the address of the mapping memory location which contains the starting address of the microprogram which will execute the instruction using the respective addressing mode.

The third column in this table gives the number of microcycles to complete only the addressing part of the instruction. The total number of microcycles used in the instruction is this number added to the number of cycles used by the ALU (if used) found under "ALU cycles" in the "OPERATIONS" subsection.

The last column is the label in the microprogram source file used to execute the instruction. The next subsection is called the "OPERATIONS". Here the program memory OPR field is specified. This opcode plus the specified P field make up the instruction which is assembled by the GPP assembler, GPPASM. Example instructions, if necessary for further clarification, may be found under the "INSTRUCTION EXAMPLES" in the group subsection.

A.1.2 Notation glossary for instruction groups

The following is a glossary of the notation used in the instruction groups.
1) P3 <== P1+P2 denotes the data represented by P1 and P2 added together and deposited into the address represented by P3.
2) , denotes parallel operations.
3) ; denotes sequential operations.
4) # prefix to the P fields denotes immediate addressing.
5) No prefix to the P fields denotes normal addressing.
6) ' prefix to the P fields denotes indirect addressing.
7) .EQ. denotes that 2s complement arithmetic is used. All these compare functions will use 2s complement arithmetic.
8) + denotes 2s complement addtion.
9) - denotes 2s complement subtraction.
10) * denotes 2s complement multiplication.
11) / denotes 2s complement division.
12) AND denotes logical bit AND. All boolean functions are used bit by bit.
13) OR denotes logical bit OR.
14) XOR denotes logical bit EXCLUSIVE OR (also called the RING SUM or SUM MODULO-2).
15) EQV denotes logical bit EQUIVALENCE (also called the BICONDITIONAL or DOT MODULO-2.
16) NOR denotes logical bit N-OR (also called NEITHER-NOR).
17) NAND denotes logical bit N-AND (also called the JOINT-DENIAL or SHEFFER-STROKE.
18) IMPLIES denotes logical bit IMPLIES (also called the CONDITIONAL).
19) BUTNOT denotes logical bit BUT-NOT (also called the INHIBIT AND).
20) P3[0:7] denotes the argument P3 bit 0 through 7.
21) Bit 0 is always the left most and is the most significant bit in a word.
22) Pj! denotes the data addressed by (Pj+1).
23) Pj!! denotes the data addressed by (Pj+2).
24) (Pj! & Pj) denotes a double precision (32-bit) word where Pj! is the most significant 16-bit word.
25) (Pj & Pj! & Pj!!) represents a floating point number.
26) f[ALUA(0)] denotes "function of ALUA(0)".
27) MM Adr denotes mapping memory address.
28) M Cyc denotes microprogram cycles.
29) MP Label denote microprogram starting address label.
30) ALU code denotes the code number of the ALU used by the instruction. This number becomes the OPR[9:15] field.
31) ALU cycles denotes the number of microprogram cycles used to perform the ALU function.
32) In general, all operation prefixes of B, D, and F refer to byte, double, and floating precision instructions respectively.
33) in general, all operation suffixes of B refer to branch instructions. An exception is the operations with suffix SB which refers to getting the P2 operator for ALUB[i].
34) All symbols in upper case are either instructions or input/output space addresses.

A.2 Summary of GPP instructions

Byte instructions:

BMOVES	BAND	BOR	BREV	BMOVLL	BMOVHH	BMOVLH
BMOVHL	BGET1	BGET2	BPUT	BSETL	BSETH	BSTLSA
BSTHSA	BSTLSB	BSTHSB				

Single precision instructions:

MOVE	MOVEB					

ADD	SUB	MUL	DIV	MAX	MIN	MOVBIT
AND	OR	XOR	EQV	NOR	NAND	IMPLIES
BUTNOT	SHFTR	SHFTL	ASR	ASL	ROTR	ROTL
REV	MAKXYA	GT	LT	GE	LE	EQ
NE	BMIO	INC	DEC	MINUS	BSWAP	

ADDSA	SUBSA	MULSA	DIVSA	MAXSA	MINSA	MVBTSA
ANDSA	ORSA	XORSA	EQVSA	NORSA	NANDSA	IMPLSA
BUTNSA	SFTRSA	SFTLSA	ASRSA	ASLSA	ROTRSA	ROTLSA
MKYXSA	GTSA	LTSA	GESA	LESA	EQSA	NESA

ADDSB	SUBSB	MULSB	DIVSB	MAXSB	MINSB	MVBTSB
ANDSB	ORSB	XORSB	EQVSB	NORSB	NANDSB	IMPLSB
BUTNSB	SFTRSB	SFTLSB	ASRSB	ASLSB	ROTRSB	ROTLSB
MKYXSB	GTSB	LTSB	GESB	LESB	EQSB	NESB

INCB	DECB	GTB	LTB	GEB	LEB	EQB
NEB						

Double precision:

DADD	DSUB	DMUL	DDIV	DMAX	DMIN	DGT
DLT	DGE	DLE	DEQ	DNE	DAND	DOR
DXOR	DEQV	DNOR	DNAND	DIMPLI	DBUTNO	DMINUS
DINC	DDEC	DSWAP	DCOMP	DREV	DSHFTR	DSHFTL
DASR	DASL	DROTR	DROTL			
DINCB	DDECB	DGTB	DLTB	DGEB	DLEB	DEQB
DNEB						

Floating point:

FADD	FSUB	FMUL	FDIV	FMINUS	FINC	FDEC
FGT	FLT	FGE	FLE	FEQ	FNE	FMAX
FMIN	FLOAT	DFLOAT	FIX	DFIX		

FINCB	FDECB	FGTB	FLTB	FGEB	FLEB	FEQB
FNEB						

Specials:

JUMP	PUSHJ	POPJ	APPLY

A.2.1 GPP instruction group %1

NOTATION: P3 <== P1;
DESCRIPTION: Data in P1 or addressed by P1 is moved to location addressed by P3. P1 and P3 may address all of the data memory space.

GROUP %1 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,,P3	:?:	.?.	G%1B1
#P1,,'P3	:?:	.?.	G%1B2
P1,,P3	:?:	.?.	G%1B3
P1,,'P3	:?:	.?.	G%1B4
'P1,,P3	:?:	.?.	G%1B5
'P1,,'P3	:?:	.?.	G%1B6

OPERATIONS:

```
MOVE                ALU code none    ALU none        ALU cycles none
----       P3 <== P1;
```

INSTRUCTION EXAMPLES:

MOVE arg,,I23
The data in location arg is stored into the local neighborhood (pixel 3) of the triple line buffer, I2.

MOVE arg,,GOUT
The data in location arg is deposited into the GOUT register. The GOUT flag is set which signals the PDP8e that fresh data is in the GOUT register. When the PDP8e reads the GOUT register the GOUT flag will be cleared.

MOVE arg,,GOUTS
The data in location arg is deposited into the GOUT register. The GOUT flag is set which signals the PDP8e that fresh data is in the GOUT register. The GPP then stalls until the PDP8e reads the GOUT register.

MOVE GIN,,data
Data from the GIN register is deposited into the location addressed by data.

MOVE GINS,,data
If the PDP8e has not loaded the GIN register since the last addressed GIN or GINS then the GPP will stall. When the PDP8e does load the GIN register, the data will be deposited into the location addressed by data.

MOVE data,,XCLK
The data addressed by the data is deposited into the XCLK register. The specified action as a function of the XCLK register is executed and the GPP continues.

MOVE data,,HALT
The data addressed by the data is deposited into the HALT register. The program execution then halts and the PDP8e is signaled by the HALT flip-flop being set.

A.2.2 GPP instruction group %2

```
NOTATION:     P3 <== P1;
              Condition <== f[P3 addressed location];
                 If condition is true
                        then PC <== P2
                        else continue;
```

DESCRIPTION: Data in P1 or addressed by P1 is moved to location addressed by P3. P1 and P3 may address all of the data memory space. The data memory location has the option of driving the conditional bus, CB. If the CB is true then the PC is loaded with data from or addressed by P2.

GROUP %2 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,P2,P3	:?:	.?.	G%2B1
#P1,P2,´P3	:?:	.?.	G%2B2
#P1,´P2,P3	:?:	.?.	G%2B3
#P1,´P2,´P3	:?:	.?.	G%2B4
P1,P2,P3	:?:	.?.	G%2B5
P1,P2,´P3	:?:	.?.	G%2B6
P1,´P2,P3	:?:	.?.	G%2B7
P1,´P2,´P3	:?:	.?.	G%2B8
´P1,P2,P3	:?:	.?.	G%2B9
´P1,P2,´P3	:?:	.?.	G%2B10
´P1,´P2,P3	:?:	.?.	G%2B11
´P1,´P2,´P3	:?:	.?.	G%2B12

OPERATIONS:

```
MOVEB              ALU code none    ALU none         ALU cycles none
-----     P3 <== P1; then if condition is true
                   then PC <== P2
                   else continue;
```

INSTRUCTION EXAMPLES:

MOVEB #0,branchaddress,GOUTB
If the GOUT flag is still set (PDP8e has not yet read the GOUT register), the next GPP instruction will be executed from program address branchaddress. If the flag was cleared, the branch will not occure. The argment #0 is ignored.

MOVEB #0,branchaddress,GIN
If the GIN register has not been loaded by the PDP8e since the last addressed GIN or GINS was used then the next GPP instruction will be executed from program address branchaddress. If the GIN register is full of fresh data then the branch will not occure. The argment, #0, is not used.

MOVEB p1,branchaddress,XCLKB
The data addressed by p1 is deposited into the XCLK register. The XCLK function takes place as specified by the XCLK register. If the condition in the function is true then the GPP program

will branch to branchaddress.

A.2.3 GPP instruction group %3

```
NOTATION:      ALUA(0)<==P1; ALUB(0)<==P2; ALUB(1)<==P3;
               P3 <== f[ALUA(0), ALUB(0), ALUB(1)];
                  where the function is one of the specified ALUs.
```

DESCRIPTION: Data from or addressed" by P1 is deposited in ALUA(0). Data from or addressed" by P2 is deposited in ALUB(0). Data from or addressed by P3 is deposited in the ALUB(1). ALUA(0), ALUB(0) and ALUB(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3.

GROUP %3 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,#P2,P3	:?:	.?.	G%3B1
#P1,#P2,´P3	:?:	.?.	G%3B2
#P1,P2,P3	:?:	.?.	G%3B3
#P1,P2,´P3	:?:	.?.	G%3B4
#P1,´P2,P3	:?:	.?.	G%3B5
#P1,´P2,´P3	:?:	.?.	G%3B6
P1,#P2,P3	:?:	.?.	G%3B7
P1,#P2,´P3	:?:	.?.	G%3B8
P1,P2,P3	:?:	.?.	G%3B9
P1,P2,P3, MEDF	:?:	.?.	G%3B10
P1,P2,´P3	:?:	.?.	G%3B11
P1,P2,´P3, MEDF	:?:	.?.	G%3B12
P1,´P2,P3	:?:	.?.	G%3B13
P1,´P2,´P3	:?:	.?.	G%3B14
´P1,#P2,P3	:?:	.?.	G%3B15
´P1,#P2,´P3	:?:	.?.	G%3B16
´P1,P2,P3	:?:	.?.	G%3B17
´P1,P2,´P3	:?:	.?.	G%3B18
´P1,´P2,P3	:?:	.?.	G%3B19
´P1,´P2,´P3	:?:	.?.	G%3B20

OPERATIONS:

```
BMOVES                ALU code #?      ALU BMOVES       ALU cycles ?#
-----     If ((P1.AND.(P2/256)).NE.0)
                      Then P3 <== (255.AND.P2).OR.P3
                      Else P3 <== P3;

BAND                  ALU code #?      ALU BAND         ALU cycles ?#
----      P3[0:7] <== P3[0:7],  P3[8:15] <==
P1[8:15].AND.P2[8:15];

BOR                   ALU code #?      ALU BOR          ALU cycles ?#
---       P3[0:7] <== P3[0:7],  P3[8:15] <== P1[8:15].OR.P2[8:15];
```

INSTRUCTION EXAMPLES:

```
BMOVES ´arg1,arg2,result
```

The data indirectly addressed by the argument, arg1, is bit logically ANDed with the addressed argument, arg2, shifted right (with zero fill) 8 bits. If this result is not equal to zero then the addressed location, result, is bit logically ORed with the argument, arg2[8:15] only. If the previous result is equal to zero then the addressed location, result, is unchanged. This is a bit-set operation for copying 8-bit PAX type image planes.

CAUTION: ALU A and B registers may not be used as P3 arguments. Check microcode author after microprograms are written.

A.2.4 GPP instruction group %4

```
NOTATION:     ALUA(0)<==P1; ALUB(1)<==P3;
              P3 <== f[ALUA(0), ALUB(0), ALUB(1)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data from or addressed" by P1 is deposited in ALUA(0). Data from or addressed by P3 is deposited in the ALUB(1). ALUA(0) and ALUB(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3.

GROUP %4 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,,P3	:?:	.?.	G%4B1
#P1,,´P3	:?:	.?.	G%4B2
P1,,P3	:?:	.?.	G%4B3
P1,,´P3	:?:	.?.	G%4B4
´P1,,P3	:?:	.?.	G%4B5
´P1,,´P3	:?:	.?.	G%4B6

OPERATIONS:

```
BREV            ALU code #?      ALU BREV         ALU cycles ?#
----     P3[0:7] <== P3[0:7]
         P3[8:15 <== bit reverse of P1[8:15];
                 (for use in FFT).

BMOVLL          ALU code #?      ALU BMOVLL       ALU cycles ?#
------   P3[0:7] <== P3[0:7],  P3[8:15] <== P1[8:15];

BMOVHH          ALU code #?      ALU BMOVHH       ALU cycles ?#
------   P3[0:7] <== P1[0:7],  P3[8:15] <== P3[8:15];

BMOVLH          ALU code #?      ALU BMOVLH       ALU cycles ?#
------   P3[0:7] <== P3[0:7],  P3[8:15] <== P1[0:7];

BMOVHL          ALU code #?      ALU BMOVHL       ALU cycles ?#
------   P3[0:7] <== P1[8:15],  P3[8:15] <== P3[8:15];

BGET1           ALU code #?      ALU BGET1        ALU cycles ?#
-----    If GBA1[17] = 0 then BMOVHL else BMOVLL;

BGET2           ALU code #?      ALU BGET2        ALU cycles ?#
-----    If GBA2[17] = 0 then BMOVHL else BMOVLL;

BPUT            ALU code #?      ALU BPUT         ALU cycles ?#
-----    If PBA1[17] = 0 then BMOVLH else BMOVLL;
```

INSTRUCTION EXAMPLES:

BGET1 ´GBA1I,,resultantbyte
The get byte auto address register, GBA1, is used as the address

of the argument for P1. This 17-bit auto address register will be incremented after the instruction is completed. Only the high order 16-bits are used for the address. If bit 17 of the GBA1 register is 0 then the BMOVHL ALU is used otherwise the BMOVLL ALU is used to create the argument deposited in data memory space location resultantbyte. This instruction is used to suck bytes from two bytes per word packed data lists. The next time this instruction is used the next packed byte will be sucked.&&&

BGET1 'GBA1,,resultanthyte
The get byte auto address register, GBA1, is used as the address of the argument for P1. This 17-bit auto address register will not be incremented after it is used as the address for the argument. Only the high order 16-bits are used for the address. If bit 17 of the GBA1 register is 0 then the BMOVHL ALU is used otherwise the BMOVLL ALU is used to create the argument deposited in data memory space location resultantbyte.

CAUTION: ALU A and B registers may not be used as P3 arguments. Check microcode author after microprograms are written.

A.2.5 GPP instruction group %5

```
NOTATION:     ALUA(0)<==P1; ALUB(0)<==P2;
              P3  <== f[ALUA(0), ALUB(0)];
                 where the function is one of the specified ALUs.
                 XARO is loaded as a function of the ALU.
```

DESCRIPTION: Data from or addressed by P1 is deposited in ALUA(0). Data from or addressed" by P2 is deposited in ALUB(0). ALUA(0) and ALUB(0) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3. The XARO register is loaded as a function of the ALU.

GROUP %5 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,#P2,P3	:?:	.?.	G%5B1
#P1,#P2,´P3	:?:	.?.	G%5B2
#P1,P2,P3	:?:	.?.	G%5B3
#P1,P2,´P3	:?:	.?.	G%5B4
#P1,´P2,P3	:?:	.?.	G%5B5
#P1,´P2,´P3	:?:	.?.	G%5B6
P1,#P2,P3	:?:	.?.	G%5B7
P1,#P2,´P3	:?:	.?.	G%5B8
P1,P2,P3	:?:	.?.	G%5B9
P1,P2,P3, MEDF	:?:	.?.	G%5B10
P1,P2,´P3	:?:	.?.	G%5B11
P1,P2,´P3, MEDF	:?:	.?.	G%5B12
P1,´P2,P3	:?:	.?.	G%5B13
P1,´P2,´P3	:?:	.?.	G%5B14
´P1,#P2,P3	:?:	.?.	G%5B15
´P1,#P2,´P3	:?:	.?.	G%5B16
´P1,P2,P3	:?:	.?.	G%5B17
´P1,P2,´P3	:?:	.?.	G%5B18
´P1,´P2,P3	:?:	.?.	G%5B19
´P1,´P2,´P3	:?:	.?.	G%5B20

OPERATIONS:

```
ADD                ALU code #?     ALU ADD          ALU cycles ?#
------  P3  <== P1+P2,   XARO  <== overflow;

SUB                ALU code #?     ALU SUB          ALU cycles ?#
------  P3  <== P1-P2,   XARO  <== underflow;

MUL                ALU code #?     ALU MUL          ALU cycles ?#
------  P3  <== P1*P2,   XARO  <== hi order of the 32 bit product;

DIV                ALU code #?     ALU DIV          ALU cycles ?#
------  P3  <== P1/P2,   XARO  <== remainder;

MAX                ALU code #?     ALU MAX          ALU cycles ?#
------  P3 <== maximum of P1 and P2,  XARO <== 0;
```

```
MIN                 ALU code #?      ALU MIN           ALU cycles ?#
------   P3  <== minimum of P1 and P2,   XAR0  <== 0;

MOVBIT              ALU code #?      ALU MOVBIT        ALU cycles ?#
------   If ((P1.AND.(P2/256)).EQ.0)
                    then P3  <== 0
                    else P3  <== (255.AND.P2);
         XAR0  <== 0;

AND                 ALU code #?      ALU AND           ALU cycles ?#
------   P3  <== bit AND of P1 and P2,   XAR0  <== 0;

OR                  ALU code #?      ALU OR            ALU cycles ?#
------   P3  <== bit OR of P1 and P2,   XAR0  <== 0;

XOR                 ALU code #?      ALU XOR           ALU cycles ?#
------   P3  <== bit XOR of P1 and P2,   XAR0  <== 0;

EQV                 ALU code #?      ALU EQV           ALU cycles ?#
------   P3  <== bit EQV of P1 and P2,   XAR0  <== 0;

NOR                 ALU code #?      ALU NOR           ALU cycles ?#
------   P3  <== bit NOR of P1 and P2,   XAR0  <== 0;

NAND                ALU code #?      ALU NAND          ALU cycles ?#
------   P3  <== bit NAND of P1 and P2,   XAR0  <== 0;

IMPLIS              ALU code #?      ALU IMPLIES       ALU cycles ?#
------   P3  <== bit IMPLIES of P1 and P2,   XAR0  <== 0;

BUTNOT              ALU code #?      ALU BUTNOT        ALU cycles ?#
------   P3  <== bit BUTNOT of P1 and P2,   XAR0  <== 0;

SHFTR               ALU code #?      ALU SHFTR         ALU cycles ?#
------   P3  <== rightshift P1 by P2 Mod 16 bits 0 fill,
         XAR0  <== 0;

SHFTL               ALU code #?      ALU SHFTL         ALU cycles ?#
------   P3  <== leftshift P1 by P2 Mod 16 bits 0 fill,
         XAR0  <== 0;

ASR                 ALU code #?      ALU ASR           ALU cycles ?#
------   P3  <== rightshift P1 by P2 Mod 16 bits bit 0 fill,
         XAR0  <== 0;

ASL                 ALU code #?      ALU ASL           ALU cycles ?#
------   P3  <== leftshift P1 by P2 Mod 16 bits bit 15 fill,
         XAR0  <== 0;

ROTR                ALU code #?      ALU ROTR          ALU cycles ?#
------   P3  <== rotate right P1 by P2 Mod 16 bits,
         XAR0  <== 0;

ROTL                ALU code #?      ALU ROTL          ALU cycles ?#
------   P3  <== rotate left P1 by P2 Mod 16 bits,
         XAR0  <== 0;
```

```
REV                 ALU code #?     ALU REV           ALU cycles ?#
------    P3 <== bit reverse of P1,   XAR0 <== 0;
                    (for use in FFT).

MAKXYA              ALU code #?     ALU MAKYXADDR     ALU cycles ?#
------    P3[0:7] <== P2[8:15],   P3[8:15] <== P1[8:15],
          XAR0 <== 0;

BSETL               ALU code #?     ALU BSETL         ALU cycles ?#
------    P3[0:7] <== P1[0:7],   P3[8:15] <== P2[8:15],
          XAR0 <== 0;

BSETH               ALU code #?     ALU BSETH         ALU cycles ?#
------    P3[8:15] <== P1[8:15],   P3[0:7] <== P2[0:7],
          XAR0 <== 0;

GT                  ALU code #?     ALU GT            ALU cycles ?#
------    If P1.GT.P2 then P3 <== P1   else continue;

LT                  ALU code #?     ALU LT            ALU cycles ?#
------    If P1.LT.P2 then P3 <== P1   else continue;

GE                  ALU code #?     ALU GE            ALU cycles ?#
------    If P1.GE.P2 then P3 <== P1   else continue;

LE                  ALU code #?     ALU LE            ALU cycles ?#
------    If P1.LE.P2 then P3 <== P1   else continue;

EQ                  ALU code #?     ALU EQ            ALU cycles ?#
------    If P1.EQ.P2 then P3 <== P1   else continue;

NE                  ALU code #?     ALU NE            ALU cycles ?#
------    If P1.NE.P2 then P3 <== P1   else continue;

BMIO                ALU code #?     ALU AND           ALU cycles ?#
------    P3 <== bit AND of P1 and P2,   XAR0 <== 0;
```

Note: BM I/O execution registers, LINE, GETI1, GETI2, and GETI3 are used as P3. The AND function is then ignored and the BM I/O function is executed. A description of the BM I/O execution registers is found in section 4 under "Data Memory Modules".

INSTRUCTION EXAMPLES:

ADD I17,I27,I37
Data from the local neighborhoods of I1 and I2 is added together and deposited in the I3 local neighborhood.

ADD I15,I25,GOUTS
Data from the local neighborhoods of I1 and I2 is added together and deposited in the GOUT register. The PDP8e must receive the data before the GPP continues with the next instruction.

BMIO arg1,arg2,LINE
The data from the addressed arguments arg1 and arg2, is deposited into the ALUA(0) and ALUB(0) registers respectively. The data memory space register, LINE, which is one of the BM I/O execution registers, is then addressed and the EM input output

transfer takes place as a result of the ALU registers. The GPP program will not procede until the I/O is complete. A description of the BM I/O execution registers is found in section 4 under "Data Memory Modules".

AND GINS,arg2,HALT
The GPP stalls until data is loaded from the PDP8e into the GIN register. The GIN register and the argument, arg2, are then bit AND together and deposited into the HALT register. The GPP instructions stop and the PDP8e is signaled that the halt flip flop is off. Only the PDP8e can restart the GPP at this point.

A.2.6 GPP instruction group %6

```
NOTATION:     ALUA(0)<==P1;
              P3 <== f[ALUA(0), ALUB(0)];
                 where the function is one of the specified ALUs.
              XARO is loaded as a function of the ALU.
```

DESCRIPTION: Data from or addressed by P1 is deposited in ALUA(0). ALUA(0) and ALUB(0) are the inputs to the selected logic unit called for by the instruction. The output of the logic unit is deposited in the data register addressed by P3. The XARO register is loaded as a function of the ALU. Note that the loading of the ALU registers is not symmetric and that only ALUA(0) is loaded.

GROUP %6 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,P3	:?:	.?.	G%6B1
#P1,´P3	:?:	.?.	G%6B2
P1,P3	:?:	.?.	G%6B3
P1,´P3	:?:	.?.	G%6B4
´P1,P3	:?:	.?.	G%6B5
´P1,´P3	:?:	.?.	G%6B6

OPERATIONS:

```
INC               ALU code #?     ALU INC           ALU cycles ?#
------   P3 <== P1+1,   XARO <== 0;

DEC               ALU code #?     ALU DEC           ALU cycles ?#
------   P3 <== P1-1,   XARO <== 0;

MINUS             ALU code #?     ALU MINUS         ALU cycles ?#
------   P3 <== -P1,   XARO <== 0;

COMP              ALU code #?     ALU COMP          ALU cycles ?#
------   P3 <== bit complement of P1,  XARO <== 0;

BSWAP             ALU code #?     ALU BSWAP         ALU cycles ?#
------   P3 <== byte swap of P1,  XARO <== 0;

ADDSA             ALU code #?     ALU ADD           ALU cycles ?#
------   P3 <== P1+ALUB(0),  XARO <== overflow;

SUBSA             ALU code #?     ALU SUB           ALU cycles ?#
------   P3 <== P1-ALUB(0),  XARO <== underflow;

MULSA             ALU code #?     ALU MUL           ALU cycles ?#
------   P3 <== P1*ALUB(0),  XARO <== hi order of the 32 bit
product;

DIVSA             ALU code #?     ALU DIV           ALU cycles ?#
------   P3 <== P1/ALUB(0),  XARO <== remainder;
```

```
MAXSA            ALU code #?     ALU MAX          ALU cycles ?#
------    P3 <== maximum of P1 and ALUB(0),  XARO <== 0;

MINSA            ALU code #?     ALU MIN          ALU cycles ?#
------    P3 <== minimum of P1 and ALUB(0),  XARO <== 0;

MVBTSA           ALU code #?     ALU MOVBIT       ALU cycles ?#
------    If ((P1.AND.(ALUB(0)/256)).EQ.0)
                 then P3 <== 0
                 else P3 <== 255.AND.ALUB(0);
          XARO <== 0;

ANDSA            ALU code #?     ALU AND          ALU cycles ?#
------    P3 <== bit AND of P1 and ALUB(0),  XARO <== 0;

ORSA             ALU code #?     ALU OR           ALU cycles ?#
------    P3 <== bit OR of P1 and ALUB(0),  XARO <== 0;

XORSA            ALU code #?     ALU XOR          ALU cycles ?#
------    P3 <== bit XOR of P1 and ALUB(0),  XARO <== 0;

EQVSA            ALU code #?     ALU EQV          ALU cycles ?#
------    P3 <== bit EQV of P1 and ALUB(0),  XARO <== 0;

NORSA            ALU code #?     ALU NOR          ALU cycles ?#
------    P3 <== bit NOR of P1 and ALUB(0),  XARO <== 0;

NANDSA           ALU code #?     ALU NAND         ALU cycles ?#
------    P3 <== bit NAND of P1 and ALUB(0),  XARO <== 0;

IMPLSA           ALU code #?     ALU IMPLIES      ALU cycles ?#
------    P3 <== bit IMPLIES of P1 and ALUB(0),  XARO <== 0;

BTNTSA           ALU code #?     ALU BUTNOT       ALU cycles ?#
------    P3 <== bit BUTNOT of P1 and ALUB(0),  XARO <== 0;

SFTRSA           ALU code #?     ALU SHFTR        ALU cycles ?#
------    P3 <== right-shift P1 by ALUB(0) Mod 16 bits 0 fill,
          XARO <== 0;

SFTLSA           ALU code #?     ALU SHFTL        ALU cycles ?#
------    P3 <== left-shift P1 by ALUB(0) Mod 16 bits 0 fill,
          XARO <== 0;

ASRSA            ALU code #?     ALU ASR          ALU cycles ?#
------    P3 <== right-shift P1 by ALUB(0) Mod 16 bits; bit 0
fill,
          XARO <== 0;

ASLSA            ALU code #?     ALU ASL          ALU cycles ?#
------    P3 <== left-shift P1 by ALUB(0) Mod 16 bits; bit 15
fill,
          XARO <== 0;

ROTRSA           ALU code #?     ALU ROTR         ALU cycles ?#
------    P3 <== rotate right P1 by ALUB(0) Mod 16 bits,
          XARO <== 0;
```

```
ROTLSA            ALU code #?     ALU ROTL        ALU cycles ?#
------   P3 <== rotate left P1 by ALUB(0) Mod 16 bits,
         XARO <== 0;

MKYXSA            ALU code #?     ALU MAKYXADDR   ALU cycles ?#
------   P3[0:7] <== ALUB(0)[8:15],  P3[8:15] <== P1[8:15],
         XARO <== 0;

BSTLSA            ALU code #?     ALU BSETL       ALU cycles ?#
------   P3[0:7] <== P1[0:7],  P3[8:15] <== ALUB(0)[8:15],
         XARO <== 0;

BSTHSA            ALU code #?     ALU BSETH       ALU cycles ?#
------   P3[8:15] <== P1[8:15],  P3[0:7] <== ALUB(0)[0:7],
         XARO <== 0;

GTSA              ALU code #?     ALU GT          ALU cycles ?#
------   If P1.GT.ALUB(0) then P3 <== P1 else continue;

LTSA              ALU code #?     ALU LT          ALU cycles ?#
------   If P1.LT.ALUB(0) then P3 <== P1 else continue;

GESA              ALU code #?     ALU GE          ALU cycles ?#
------   If P1.GE.ALUB(0) then P3 <== P1 else continue;
LESA              ALU code #?     ALU LE          ALU cycles ?#
------   If P1.LE.ALUB(0) then P3 <== P1 else continue;

EQSA              ALU code #?     ALU EQ          ALU cycles ?#
------   If P1.EQ.ALUB(0) then P3 <== P1 else continue;

NESA              ALU code #?     ALU NE          ALU cycles ?#
------   If P1.NE.ALUB(0) then P3 <== P1 else continue;
```

INSTRUCTION EXAMPLES:

A.2.7 GPP instruction group %7

```
NOTATION:       ALUB(0)<==P2;
                P3  <== f[ALUA(0), ALUB(0)];
                    where the function is one of the specified ALUs.
                XARO is loaded as a function of the ALU.
```

DESCRIPTION: Data from or addressed by P2 is deposited in ALUB(0). ALUA(0) and ALUB(0) are the inputs to the selected logic unit called for by the instruction. The output of the logic unit is deposited in the data register addressed by P3. The XARO register is loaded as a function of the ALU. Note that the loading of the ALU registers is not symmetric and that only ALUB(0) is loaded. These instructions are the arithmetic list processing instructions and take the form of "operator 'ST'ore". It is useful when a previous result can be left in the ALUA(0).

GROUP %7 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P2,P3	:?:	.?.	G%7B1
#P2,'P3	:?:	.?.	G%7B2
P2,P3	:?:	.?.	G%7B3
P2,'P3	:?:	.?.	G%7B4
'P2,P3	:?:	.?.	G%7B5
'P2,'P3	:?:	.?.	G%7B6

OPERATIONS:

```
ADDSB             ALU code #?       ALU ADD          ALU cycles ?#
------   P3  <== ALUA(0)+P2,   XARO <== overflow;

SUBSB             ALU code #?       ALU SUB          ALU cycles ?#
------   P3  <== ALUA(0)-P2,   XARO <== underflow;

MULSB             ALU code #?       ALU MUL          ALU cycles ?#
------   P3  <== ALUA(0)*P2,   XARO <== hi order of the 32 bit
         product;

DIVSB             ALU code #?       ALU DIV          ALU cycles ?#
------   P3  <== ALUA(0)/P2,   XARO <== remainder;

MAXSB             ALU code #?       ALU MAX          ALU cycles ?#
------   P3  <== maximum of ALUA(0) and P2,   XARO <== 0;

MINSB             ALU code #?       ALU MIN          ALU cycles ?#
------   P3  <== minimum of ALUA(0) and P2,   XARO <== 0;

MVBTSB            ALU code #?       ALU MOVBIT       ALU cycles ?#
------   If ((ALUA(0).AND.(P2/256)).EQ.0)
                  then P3  <== 0
                  else P3  <== 255.AND.P2;
         XARO  <== 0;

ANDSB             ALU code #?       ALU AND          ALU cycles ?#
```

```
------  P3 <== bit AND of ALUA(0) and P2,  XAR0 <== 0;

ORSB           ALU code #?     ALU OR          ALU cycles ?#
------  P3 <== bit OR of ALUA(0) and P2,  XAR0 <== 0;

XORSB          ALU code #?     ALU XOR         ALU cycles ?#
------  P3 <== bit XOR of ALUA(0) and P2,  XAR0 <== 0;

EQVSB          ALU code #?     ALU EQV         ALU cycles ?#
------  P3 <== bit EQV of ALUA(0) and P2,  XAR0 <== 0;

NORSB          ALU code #?     ALU NOR         ALU cycles ?#
------  P3 <== bit NOR of ALUA(0) and P2,  XAR0 <== 0;

NANDSB         ALU code #?     ALU NAND        ALU cycles ?#
------  P3 <== bit NAND of ALUA(0) and P2,  XAR0 <== 0;

IMPLSB         ALU code #?     ALU IMPLIES     ALU cycles ?#
------  P3 <== bit IMPLIES of ALUA(0) and P2,  XAR0 <== 0;

BTNTSB         ALU code #?     ALU BUTNOT      ALU cycles ?#
------  P3 <== bit BUTNOT of ALUA(0) and P2,  XAR0 <== 0;

SFTRSB         ALU code #?     ALU SHFTR       ALU cycles ?#
------  P3 <== right-shift ALUA(0) by P2 Mod 16 bits 0 fill,
        XAR0 <== 0;

SFTLSB         ALU code #?     ALU SHFTL       ALU cycles ?#
------  P3 <== left-shift ALUA(0) by P2 Mod 16 bits 0 fill,
        XAR0 <== 0;

ASRSB          ALU code #?     ALU ASR         ALU cycles ?#
------  P3 <== right-shift ALUA(0) by P2 Mod 16 bits; bit 0
fill,
        XAR0 <== 0;

ASLSB          ALU code #?     ALU ASL         ALU cycles ?#
------  P3 <== left-shift ALUA(0) by P2 Mod 16 bits; bit 15
fill,
        XAR0 <== 0;

ROTRSB         ALU code #?     ALU ROTR        ALU cycles ?#
------  P3 <== rotate right ALUA(0) by P2 Mod 16 bits,
        XAR0 <== 0;

ROTLSB         ALU code #?     ALU ROTL        ALU cycles ?#
------  P3 <== rotate left ALUA(0) by P2 Mod 16 bits,
        XAR0 <== 0;

MKYXSB         ALU code #?     ALU MAKYXADDR   ALU cycles ?#
------  P3[0:7] <== P2[8:15],  P3[8:15] <== ALUA(0)[8:15],
        XAR0 <== 0;

BSTLSB         ALU code #?     ALU BSETL       ALU cycles ?#
------  P3[0:7] <== ALUA(0)[0:7],  P3[8:15] <== P2[8:15],
        XAR0 <== 0;
```

```
BSTHSB              ALU code #?     ALU BSETH         ALU cycles ?#
------    P3[8:15] <== ALUA(0)[8:15],  P3[0:7] <== P2[0:7],
          XAR0 <== 0;

GTSB                ALU code #?     ALU GT            ALU cycles ?#
------    If ALUA(0).GT.P2 then P3 <== ALUA(0) else continue;

LTSB                ALU code #?     ALU LT            ALU cycles ?#
------    If ALUA(0).LT.P2 then P3 <== ALUA(0) else continue;

GESB                ALU code #?     ALU GE            ALU cycles ?#
------    If ALUA(0).GE.P2 then P3 <== ALUA(0) else continue;
LESB                ALU code #?     ALU LE            ALU cycles ?#
------    If ALUA(0).LE.P2 then P3 <== ALUA(0) else continue;

EQSB                ALU code #?     ALU EQ            ALU cycles ?#
------    If ALUA(0).EQ.P2 then P3 <== ALUA(0) else continue;

NESB                ALU code #?     ALU NE            ALU cycles ?#
------    If ALUA(0).NE.P2 then P3 <== ALUA(0) else continue;
```

INSTRUCTION EXAMPLES:

A.2.8 GPP instruction group %8

NOTATION: ALUA(0)<==P1; P3<==ALUA(0)*ALUB(0);
If P3=0 then PC<==P2 else continue;
(*=One of GPP specified ALU operations.)

DESCRIPTION: Data from or addressed by P1 is deposited in ALUA(0). ALUA(0) and ALUB(0) are the inputs to the selected logic unit called for by the instruction. The output of the logic unit is deposited in the data register addressed by P3. If P3=0 then move P2 into the PC. The INCB and the DECB are implemented in order to simplify the implemetation of for and do loops.

GROUP %8 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,P2,P3	:?:	.?.	G%8B1
#P1,P2,´P3	:?:	.?.	G%8B2
#P1,´P2,P3	:?:	.?.	G%8B3
#P1,´P2,´P3	:?:	.?.	G%8B4
P1,P2,P3	:?:	.?.	G%8B5
P1,P2,´P3	:?:	.?.	G%8B6
P1,´P2,P3	:?:	.?.	G%8B7
P1,´P2,´P3	:?:	.?.	G%8B8
´P1,P2,P3	:?:	.?.	G%8B9
´P1,P2,´P3	:?:	.?.	G%8B10
´P1,´P2,P3	:?:	.?.	G%8B11
´P1,´P2,´P3	:?:	.?.	G%8B12

OPERATIONS:

```
INCB              ALU code #?      ALU  INC           ALU cycles ?#
------  P3 <== P1+1, if P3 = 0 then PC <== P2 else continue;

DECB              ALU code #?      ALU  DEC           ALU cycles ?#
------  P3 <== P1-1, if P3 = 0 then PC <== P2 else continue;
```

INSTRUCTION EXAMPLES:

A.2.9 GPP instruction group %9

```
NOTATION:     ALUA(0)<==P1, ALUB(0)<==P2;
              Condition <== ALUA(0)*ALUB(0);
                 If condition is true then PC<==P3
                         else continue;
```

DESCRIPTION: Data from or addressed by P1 is deposited in ALUA(0). Data from or addressed by P2 is deposited in ALUB(0). ALUA(0) and ALUB(0) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output condition of the ALU is then tested. If true, data from P3 is deposited into the PC.

GROUP %9 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1,#P2,P3	:?:	.?.	G%9B1
#P1,#P2,´P3	:?:	.?.	G%9B2
#P1,P2,P3	:?:	.?.	G%9B3
#P1,P2,´P3	:?:	.?.	G%9B4
#P1,´P2,P3	:?:	.?.	G%9B5
#P1,´P2,´P3	:?:	.?.	G%9B6
P1,#P2,P3	:?:	.?.	G%9B7
P1,#P2,´P3	:?:	.?.	G%9B8
P1,P2,P3	:?:	.?.	G%9B9
P1,P2,P3, MEDF	:?:	.?.	G%9B10
P1,P2,´P3	:?:	.?.	G%9B11
P1,P2,´P3, MEDF	:?:	.?.	G%9B12
P1,´P2,P3	:?:	.?.	G%9B13
P1,´P2,´P3	:?:	.?.	G%9B14
´P1,#P2,P3	:?:	.?.	G%9B15
´P1,#P2,´P3	:?:	.?.	G%9B16
´P1,P2,P3	:?:	.?.	G%9B17
´P1,P2,´P3	:?:	.?.	G%9B18
´P1,´P2,P3	:?:	.?.	G%9B19
´P1,´P2,´P3	:?:	.?.	G%9B20

OPERATIONS:

```
GTB              ALU code #?     ALU GT          ALU cycles ?#
------  If P1.GT.P2 then PC <== P3 else continue;

LTB              ALU code #?     ALU LT          ALU cycles ?#
------  If P1.LT.P2 then PC <== P3 else continue;

GEB              ALU code #?     ALU GE          ALU cycles ?#
------  If P1.GE.P2 then PC <== P3 else continue;

LEB              ALU code #?     ALU LE          ALU cycles ?#
------  If P1.LE.P2 then PC <== P3 else continue;

EQB              ALU code #?     ALU EQ          ALU cycles ?#
------  If P1.EQ.P2 then PC <== P3 else continue;
```

```
NEB               ALU code #?      ALU NE          ALU cycles ?#
------  If P1.NE.P2 then PC <== P3 else continue;
```

INSTRUCTION EXAMPLES:

A.2.10 GPP instruction group %10

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUB(0) <== P2,   ALUB(1) <== P2!;

              P3  <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)],
              P3!  <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)];
              XAR0  <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)],
              XAR1  <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Double precision arithmetic is implemented with Group 8 by accessing groups of 2 words associated with P1, P2, and P3 effective addresses. The address Pi+1 is denoted here as Pi!.

GROUP %10 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%10B1
P1,P2,´P3	:?:	.?.	G%10B2
P1,´P2,P3	:?:	.?.	G%10B3
P1,´P2,´P3	:?:	.?.	G%10B4
´P1,P2,P3	:?:	.?.	G%10B5
´P1,P2,´P3	:?:	.?.	G%10B6
´P1,´P2,P3	:?:	.?.	G%10B7
´P1,´P2,´P3	:?:	.?.	G%10B8

OPERATIONS:

```
DADD              ALU code #?      ALU DADD          ALU cycles ?#
------   (P3! & P3) <== (P1! & P1)+(P2! & P2),
         (XAR1 & XAR0) <== overflow;

DSUB              ALU code #?      ALU DSUB          ALU cycles ?#
------   (P3! & P3) <== (P1! & P1)+(P2! & P2),
         (XAR1 & XAR0) <== underflow;

DMUL              ALU code #?      ALU DMUL          ALU cycles ?#
------   (P3! & P3) <== lower order (P1! & P1)*(P2! & P2),
         (XAR1 & XAR0) <== high order (P1! & P1)*(P2! & P2);

DDIV              ALU code #?      ALU DDIV          ALU cycles ?#
------   (P3! & P3)<==(P1! & P1)/(P2! & P2),
                 (XAR1 & XAR0) <== remainder;
```

INSTRUCTION EXAMPLES:

A.2.11 GPP instruction group %11

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUB(0) <== P2,   ALUB(1) <== P2!;

              P3  <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)],
              P3! <== f[ALUA(0), ALUA(1), ALUB(0), ALUB(1)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1). Data addressed by P2 and P2! is deposited in ALUB(0) and ALUB(1). ALUA(0), ALUA(1), and ALUB(0), ALUB(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3 and P3!.

GROUP %11 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%11B1
P1,P2,´P3	:?:	.?.	G%11B2
P1,´P2,P3	:?:	.?.	G%11B3
P1,´P2,´P3	:?:	.?.	G%11B4
´P1,P2,P3	:?:	.?.	G%11B5
´P1,P2,´P3	:?:	.?.	G%11B6
´P1,´P2,P3	:?:	.?.	G%11B7
´P1,´P2,´P3	:?:	.?.	G%11B8

OPERATIONS:

```
DMAX             ALU code #?      ALU DMAX          ALU cycles ?#
------   (P3! & P3) <== maximum of (P1! & P1) and (P2! & P2);

DMIN             ALU code #?      ALU DMIN          ALU cycles ?#
------   (P3! & P3) <== minimum of (P1! & P1) and (P2! & P2);

DGT              ALU code #?      ALU DGT           ALU cycles ?#
------   If (P1! & P1).GT.(P2! & P2)
                 then (P3! & P3) <== (P2! & P2) else continue;

DLT              ALU code #?      ALU DLT           ALU cycles ?#
------   If (P1! & P1).LT.(P2! & P2)
                 then (P3! & P3) <== (P2! & P2) else continue;

DGE              ALU code #?      ALU DGE           ALU cycles ?#
------   If (P1! & P1).GE.(P2! & P2)
                 then (P3! & P3) <== (P2! & P2) else continue;

DLE              ALU code #?      ALU DLE           ALU cycles ?#
------   If (P1! & P1).LE.(P2! & P2)
                 then (P3! & P3) <== (P2! & P2) else continue;

DEQ              ALU code #?      ALU DEQ           ALU cycles ?#
------   If (P1! & P1).EQ.(P2! & P2)
                 then (P3! & P3) <== (P2! & P2) else continue;
```

```
DNE               ALU code #?      ALU DNE           ALU cycles ?#
------   If (P1! & P1).NE.(P2! & P2)
                  then (P3! & P3) <== (P2! & P2) else continue;

DAND              ALU code #?      ALU DAND          ALU cycles ?#
------   (P3! & P3) <== bit AND of (P1! & P1) and (P2! & P2);

DOR               ALU code #?      ALU DOR           ALU cycles ?#
------   (P3! & P3) <== bit OR of (P1! & P1) and (P2! & P2);

DXOR              ALU code #?      ALU DXOR          ALU cycles ?#
------   (P3! & P3) <== bit XOR of (P1! & P1) and (P2! & P2);

DEQV              ALU code #?      ALU DEQV          ALU cycles ?#
------   (P3! & P3) <== bit EQV of (P1! & P1) and (P2! & P2);

DNOR              ALU code #?      ALU DNOR          ALU cycles ?#
------   (P3! & P3) <== bit NOR of (P1! & P1) and (P2! & P2);

DNAND             ALU code #?      ALU DNAND         ALU cycles ?#
------   (P3! & P3) <== bit NAND of (P1! & P1) and (P2! & P2);

DIMPLI            ALU code #?      ALU DIMPLIES      ALU cycles ?#
------   (P3! & P3) <== bit IMPLIES of (P1! & P1) and (P2! & P2);

DBUTNO            ALU code #?      ALU DBUTNOT       ALU cycles ?#
------   (P3! & P3) <== bit BUTNOT of (P1! & P1) and (P2! & P2);
```

INSTRUCTION EXAMPLES:

A.2.12 GPP instruction group %12

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;

              P3  <== f[ALUA(0), ALUA(1)],
              P3! <== f[ALUA(0), ALUA(1)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1). ALUA(0) and ALUA(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3 and P3!.

GROUP %12 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P3	:?:	.?.	G%12B1
P1,´P3	:?:	.?.	G%12B2
´P1,P3	:?:	.?.	G%12B3
´P1,´P3	:?:	.?.	G%12B4

OPERATIONS:

```
DMINUS           ALU code #?      ALU DMINUS       ALU cycles ?#
------   (P3! & P3) <== -(P1! & P1);

DINC             ALU code #?      ALU DINC         ALU cycles ?#
------   (P3! & P3) <== (P1! & P1)+1;

DDEC             ALU code #?      ALU DDEC         ALU cycles ?#
------   (P3! & P3) <== (P1! & P1)-1;

DSWAP            ALU code #?      ALU DSWAP        ALU cycles ?#
------   P3 <== P1!,  P3! <== P1;

DCOMP            ALU code #?      ALU DCOMP        ALU cycles ?#
------   (P3! & P3) <== bit complement of (P1! & P1);

DREV             ALU code #?      ALU DREV         ALU cycles ?#
------   (P3! & P3) <== bit reverse of (P1! & P1);
                   (for use in FFT).
```

INSTRUCTION EXAMPLES:

A.2.13 GPP instruction group %13

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUB(0) <== P2;

              P3  <== f[ALUA(0), ALUA(1), ALUB(0)],
              P3!  <== f[ALUA(0), ALUA(1), ALUB(0)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1). Data addressed by P2 is deposited in ALUB(0). ALUA(0), ALUA(1), and ALUB(0) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3 and P3!.

GROUP %13 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,#P2,P3	:?:	.?.	G%13B1
P1,#P2,´P3	:?:	.?.	G%13B2
P1,P2,P3	:?:	.?.	G%13B3
P1,P2,´P3	:?:	.?.	G%13B4
P1,´P2,P3	:?:	.?.	G%13B5
P1,´P2,´P3	:?:	.?.	G%13B6
´P1,#P2,P3	:?:	.?.	G%13B7
´P1,#P2,´P3	:?:	.?.	G%13B8
´P1,P2,P3	:?:	.?.	G%13B9
´P1,P2,´P3	:?:	.?.	G%13B10
´P1,´P2,P3	:?:	.?.	G%13B11
´P1,´P2,´P3	:?:	.?.	G%13B12

OPERATIONS:

```
DSHFTR           ALU code #?     ALU DSHFTR      ALU cycles ?#
------   (P3! & P3) <== right-shift (P1! & P1) by P2 Mod 32 bits
         0 fill;

DSHFTL           ALU code #?     ALU DSHFTL      ALU cycles ?#
------   (P3! & P3) <== Left-shift (P1! & P1) by P2 Mod 32 bits
         0 fill;

DASR             ALU code #?     ALU DASR        ALU cycles ?#
------   (P3! & P3) <== right-shift (P1! & P1) by P2 Mod 32 bits
         bit 0 of P1 fill;

DASL             ALU code #?     ALU DASL        ALU cycles ?#
------   (P3! & P3) <== right-shift (P1! & P1) by P2 Mod 32 bits
         bit 15 of P1! fill;

DROTR            ALU code #?     ALU DROTR       ALU cycles ?#
------   (P3! & P3) <== rotate right (P1! & P1) by P2
         Mod 32 bits;

DROTL            ALU code #?     ALU DROTL       ALU cycles ?#
```

```
------  (P3! & P3) <== rotate left (P1! & P1) by P2 Mod 32 bits;
```

INSTRUCTION EXAMPLES:

A.2.14 GPP instruction group %14

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;

              P3  <== f[ALUA(0), ALUA(1)],
              P3! <== f[ALUA(0), ALUA(1)];
                 where the function is one of the specified ALUs.

              If (P3! & P3) = 0 then PC <== P2 else continue;
```

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1). ALUA(0) and ALUA(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3 and P3!. If the output is zero then the PC is loaded with the argument from P2.

GROUP %14 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%14B1
P1,P2,'P3	:?:	.?.	G%14B2
P1,'P2,P3	:?:	.?.	G%14B3
P1,'P2,'P3	:?:	.?.	G%14B4
'P1,P2,P3	:?:	.?.	G%14B5
'P1,P2,'P3	:?:	.?.	G%14B6
'P1,'P2,P3	:?:	.?.	G%14B7
'P1,'P2,'P3	:?:	.?.	G%14B8

OPERATIONS:

```
DINCB          ALU code #?     ALU DINC        ALU cycles ?#
------  (P3! & P3) <== (P1! & P1)+1;
        If (P3! & P3) = 0 then PC <== P2 else continue;

DDECB          ALU code #?     ALU DDEC        ALU cycles ?#
------  (P3! & P3) <== (P1! & P1)-1;
        If (P3! & P3) = 0 then PC <== P2 else continue;
```

INSTRUCTION EXAMPLES:

A.2.15 GPP instruction group %15

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUB(0) <== P2,   ALUB(1) <== P2!;

              Condition <== f[ALUA(0),ALUA(1), ALUB(0),ALUB(1)],
                 where the function is one of the specified ALUs.
              If condition is true, then PC <== P3;
```

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1). Data addressed by P2 and P2! is deposited in ALUB(0) and ALUB(1). ALUA(0), ALUA(1), and ALUB(0), ALUB(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output condition of the ALU is then tested. If true, data from P3 is deposited into the PC.

GROUP %15 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%15B1
P1,P2,´P3	:?:	.?.	G%15B2
P1,´P2,P3	:?:	.?.	G%15B3
P1,´P2,´P3	:?:	.?.	G%15B4
´P1,P2,P3	:?:	.?.	G%15B5
´P1,P2,´P3	:?:	.?.	G%15B6
´P1,´P2,P3	:?:	.?.	G%15B7
´P1,´P2,´P3	:?:	.?.	G%15B8

OPERATIONS:

```
DGTB             ALU code #?     ALU DGT           ALU cycles ?#
------   If (P1! & P1).GT.(P2! & P2) then PC <== P3
         else continue;

DLTB             ALU code #?     ALU DLT           ALU cycles ?#
------   If (P1! & P1).LT.(P2! & P2) then PC <== P3
         else continue;

DGEB             ALU code #?     ALU DGE           ALU cycles ?#
------   If (P1! & P1).GE.(P2! & P2) then PC <== P3
         else continue;

DLEB             ALU code #?     ALU DLE           ALU cycles ?#
------   If (P1! & P1).LE.(P2! & P2) then PC <== P3
         else continue;

DEQB             ALU code #?     ALU DEQ           ALU cycles ?#
------   If (P1! & P1).EQ.(P2! & P2) then PC <== P3
         else continue;

DNEB             ALU code #?     ALU DNE           ALU cycles ?#
------   If (P1! & P1).NE.(P2! & P2) then PC <== P3
         else continue;
```

INSTRUCTION EXAMPLES:

A.2.16 GPP instruction group %16

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUA(2) <== P1!!;
              ALUB(0) <== P2,   ALUB(1) <== P2!;
              ALUB(2) <== P2!!;

              (P3 & P3! & P3!!) <== f[ALUA(0-2), ALUB(0-2)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0)< ALUA(1), and ALUA(2). Data addressed by P2, P2! and P2!! is deposited in ALUB(0), ALUB(1), and ALUB(2). ALUA(0-2) and ALUB(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3, P3!, and P3!!.

The floating point number representation uses the excess exponent code because of its interesting properties. These include the ability to test if a floating point number is >, =, or < 0 by testing the exponent word as if it were a 16-bit integer. The 32-bit 2's complement mantissa is designed so that the double precision ALUs may be used to carry out its arithmetic after the words are aleigned. Note that the sign bit of the 2's complement mantissa is duplicated in the sign bit of the exponent word. The representation is as follows:

```
word 1: |1-bit mantissa sign | 15-bit signed
          '20000 excess expon.|
word 2: |2's complement high order mantissa|
word 3: |2's complement low order mantissa|
```

```
Note: 0.0 has word 1=0 !!!
Note: + number has bit 0 of exp. word =0
Note: - number has bit 0 of exp. word =1
eg. 1.0  = (0|1000...001),(0100...000)(000...000)
eg. -1.0 = (1|1000...001),(1111...111)(111...111)
eg. 0.1  = (0|1000...000),(0100...000)(000...000)
eg. 0.01  = (0|0111...111),(0100...000)(000...000)
eg. 0.001  = (0|0111...110),(0100...000)(000...000)
eg. 0.000  = (0|0000...000),(0000...000)(000...000)
```

Various floating point arithmetic and conversion errors are reported in a GR I/O space status register.

```
FPPSTATUS[0] = exponent overflow error
FPPSTATUS[1] = exponent underflow error
FPPSTATUS[2] = mantissa overflow error
FPPSTATUS[3] = mantissa underflow error
FPPSTATUS[4] = division by zero.
```

GROUP %16 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label

```
P1,P2,P3                  :?:     .?.     G%16B1
P1,P2,'P3                 :?:     .?.     G%16B2
P1,'P2,P3                 :?:     .?.     G%16B3
P1,'P2,'P3                :?:     .?.     G%16B4
'P1,P2,P3                 :?:     .?.     G%16B5
'P1,P2,'P3                :?:     .?.     G%16B6
'P1,'P2,P3                :?:     .?.     G%16B7
'P1,'P2,'P3               :?:     .?.     G%16B8
```

OPERATIONS:

```
FADD            ALU code #?     ALU FADD        ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!) + (P2 & P2! &
        P2!!);

FSUB            ALU code #?     ALU FSUB        ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!) - (P2 & P2! &
        P2!!);

FMUL            ALU code #?     ALU FMUL        ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!) * (P2 & P2! &
        P2!!);

FDIV            ALU code #?     ALU FDIV        ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!) / (P2 & P2! &
        P2!!);

FMAX            ALU code #?     ALU FMAX        ALU cycles ?#
------  (P3 & P3! & P3!!) <==
                maximum of (P1 & P1! & P1!!) and (P2 & P2! &
        P2!!);

FMIN            ALU code #?     ALU FMIN        ALU cycles ?#
------  (P3 & P3! & P3!!) <==
                minimum of (P1 & P1! & P1!!) and (P2 & P2! &
        P2!!);

FGT             ALU code #?     ALU FGT         ALU cycles ?#
------  If (P1 & P1! & P1!!).GT.(P2 & P2! & P2!!) then
        (P3 & P3! & P3!!) <== (P2 & P2! & P2!!) else continue;

FLT             ALU code #?     ALU DLT         ALU cycles ?#
------  If (P1 & P1! & P1!!).LT.(P2 & P2! & P2!!) then
        (P3 & P3! & P3!!) <== (P2 & P2! & P2!!) else continue;

FGE             ALU code #?     ALU FGE         ALU cycles ?#
------  If (P1 & P1! & P1!!).GE.(P2 & P2! & P2!!) then
        (P3 & P3! & P3!!) <== (P2 & P2! & P2!!) else continue;

FLE             ALU code #?     ALU FLE         ALU cycles ?#
------  If (P1 & P1! & P1!!).LE.(P2 & P2! & P2!!) then
        (P3 & P3! & P3!!) <== (P2 & P2! & P2!!) else continue;

FEQ             ALU code #?     ALU FEQ         ALU cycles ?#
------  If (P1 & P1! & P1!!).EQ.(P2 & P2! & P2!!) then
        (P3 & P3! & P3!!) <== (P2 & P2! & P2!!) else continue;
```

```
FNE                ALU code #?      ALU  FNE          ALU cycles ?#
------   If ( P1  & P1! & P1!!).NE.(P2 & P2! & P2!!) then
         ( P3 & P3! & P3!! ) <== (P2 & P2! & P2!!) else continue;
```

INSTRUCTION EXAMPLES:

A.2.17 GPP instruction group %17

```
NOTATION:     ALUA(0) <== P1,  ALUA(1) <== P1!;
              ALUA(2) <== P1!!;

              (P3 & P3! & P3!!) <== f[ALUA(0-2)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0)< ALUA(1), and ALUA(2). ALUA(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3, P3!, and P3!.

GROUP %17 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,,P3	:?:	.?.	G%17B1
P1,,´P3	:?:	.?.	G%17B2
´P1,,P3	:?:	.?.	G%17B3
´P1,,´P3	:?:	.?.	G%17B4

OPERATIONS:

```
FMINUS            ALU code #?      ALU FMINUS       ALU cycles ?#
------  (P3 & P3! & P3!!) <== -(P1 & P1! & P1!!);

FINC              ALU code #?      ALU FINC         ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!)+1;

FDEC              ALU code #?      ALU FDEC         ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!)-1;
```

INSTRUCTION EXAMPLES:

A.2.18 GPP instruction group %18

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUA(2) <== P1!!;

              (P3 & P3! & P3!!) <== f[ALUA(0-2)];
                 where the function is one of the specified ALUs.
              If (P3 & P3! & P3!!) = 0
                 then PC <== P2
                 else continue;
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0)< ALUA(1), and ALUA(2). ALUA(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3, P3!, and P3!. If the output is zero then the PC is loaded with the argument from P2.

GROUP %18 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%18B1
P1,P2,´P3	:?:	.?.	G%18B2
P1,´P2,P3	:?:	.?.	G%18B3
P1,´P2,´P3	:?:	.?.	G%18B4
´P1,P2,P3	:?:	.?.	G%18B5
´P1,P2,´P3	:?:	.?.	G%18B6
´P1,´P2,P3	:?:	.?.	G%18B7
´P1,´P2,´P3	:?:	.?.	G%18B8

OPERATIONS:

```
FINCB              ALU code #?     ALU FINC          ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!)+1;
                   If P3 = 0 then PC<==P2 else continue;

FDECB              ALU code #?     ALU FDEC          ALU cycles ?#
------  (P3 & P3! & P3!!) <== (P1 & P1! & P1!!)-1;
                   If P3 = 0 then PC<=P2 else continue;
```

INSTRUCTION EXAMPLES:

A.2.19 GPP instruction group %19

```
NOTATION:      ALUA(0) <== P1,   ALUA(1) <== P1!;
               ALUA(2) <== P1!!;
               ALUB(0) <== P2,   ALUB(1) <== P2!;
               ALUB(2) <== P2!!;

               Condition <== f[ALUA(0-2), ALUB(0-2)];
                  where the function is one of the specified ALUs.
               If condition is true then PC <== P3 else continue;
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0)< ALUA(1), and ALUA(2). Data addressed by P2, P2! and P2!! is deposited in ALUB(0), ALUB(1), and ALUB(2). ALUA(0-2) and ALUB(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output condition of the ALU is then tested. If true, data from P3 is deposited into the PC.

GROUP %19 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3	:?:	.?.	G%19B1
P1,P2,'P3	:?:	.?.	G%19B2
P1,'P2,P3	:?:	.?.	G%19B3
P1,'P2,'P3	:?:	.?.	G%19B4
'P1,P2,P3	:?:	.?.	G%19B5
'P1,P2,'P3	:?:	.?.	G%19B6
'P1,'P2,P3	:?:	.?.	G%19B7
'P1,'P2,'P3	:?:	.?.	G%19B8

OPERATIONS:

```
FGTB              ALU code #?      ALU FGT           ALU cycles ?#
------    If (P1 & P1! & P1!!).GT.(P2 & P2! & P2!!)
                  then PC <== P3 else continue;

FLTB              ALU code #?      ALU DLT           ALU cycles ?#
------    If (P1 & P1! & P1!!).LT.(P2 & P2! & P2!!)
                  then PC <== P3 else continue;

FGEB              ALU code #?      ALU FGE           ALU cycles ?#
------    If (P1 & P1! & P1!!).GE.(P2 & P2! & P2!!)
                  then PC <== P3 else continue;

FLEB              ALU code #?      ALU FLE           ALU cycles ?#
------    If (P1 & P1! & P1!!).LE.(P2 & P2! & P2!!)
                  then PC <== P3 else continue;

FEQB              ALU code #?      ALU FEQ           ALU cycles ?#
------    If (P1 & P1! & P1!!).EQ.(P2 & P2! & P2!!)
                  then PC <== P3 else continue;

FNEB              ALU code #?      ALU FNE           ALU cycles ?#
------    If (P1 & P1! & P1!!).NE.(P2 & P2! & P2!!)
```

```
            then PC <== P3 else continue;
```

INSTRUCTION EXAMPLES:

A.2.20 GPP instruction group %20

```
NOTATION:     ALUA(0) <== P1;
              (P3 & P3! & P3!!) <== f[ALUA(0)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1 is deposited in ALUA(0). ALUA(0) is the input to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3, P3!, and P3!.

GROUP %20 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,,P3	:?:	.?.	G%20B1
P1,,´P3	:?:	.?.	G%20B2
´P1,,P3	:?:	.?.	G%20B3
´P1,,´P3	:?:	.?.	G%20B4

OPERATIONS:

```
FLOAT             ALU code #?      ALU FLOAT         ALU cycles ?#
------   (P3 & P3! & P3!!) <== floating point number of the
single
precision number  P1;
```

INSTRUCTION EXAMPLES:

A.2.21 GPP instruction group %21

NOTATION: ALUA(0) <== P1, ALUA(1) <== P1!;
(P3 & P3! & P3!!) <== f[ALUA(0), ALUA(1)];
where the function is one of the specified ALUs.

DESCRIPTION: Data addressed by P1 and P1! is deposited in ALUA(0) and ALUA(1) respectively. ALUA(0) and ALUA(1) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3, P3!, and P3!.

GROUP %21 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,,P3	:?:	.?.	G%21B1
P1,,´P3	:?:	.?.	G%21B2
´P1,,P3	:?:	.?.	G%21B3
´P1,,´P3	:?:	.?.	G%21B4

OPERATIONS:

DFLOAT ALU code #? ALU DFLOAT ALU cycles ?#
------ (P3 & P3! & P3!!) <== floating point number of the double precision number (P1! & P1);

INSTRUCTION EXAMPLES:

A.2.22 GPP instruction group %22

```
NOTATION:     ALUA(0) <== P1,  ALUA(1) <== P1!;
              ALUA(2) <== P1!!;

              P3 <== f[ALUA(0-2)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0) ALUA(1), and ALUA(2). ALUA(0-2) and ALUB(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by P3.

GROUP %22 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,,P3	:?:	.?.	G%22B1
P1,,´P3	:?:	.?.	G%22B2
´P1,,P3	:?:	.?.	G%22B3
´P1,,´P3	:?:	.?.	G%22B4

OPERATIONS:

```
FIX                ALU code #?      ALU FIX          ALU cycles ?#
------   P3 <== single precision number of the floating point
         number (P1 & P1! & P1!!);
```

INSTRUCTION EXAMPLES:

A.2.23 GPP instruction group %23

```
NOTATION:     ALUA(0) <== P1,   ALUA(1) <== P1!;
              ALUA(2) <== P1!!;

              (P3! & P3) <== f[ALUA(0-2)];
                 where the function is one of the specified ALUs.
```

DESCRIPTION: Data addressed by P1, P1! and P1!! is deposited in ALUA(0) ALUA(1), and ALUA(2). ALUA(0-2) and ALUB(0-2) are the inputs to the selected arithmetic unit of the GPP called for as a function of the instruction. The output of the ALU is deposited in the data register addressed by (P3! & P3).

GROUP %23 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,,P3	:?:	.?.	G%23B1
P1,,´P3	:?:	.?.	G%23B2
´P1,,P3	:?:	.?.	G%23B3
´P1,,´P3	:?:	.?.	G%23B4

OPERATIONS:

```
DFIX             ALU code #?      ALU DFIX         ALU cycles ?#
------    (P3! & P3) <== double precision number of the floating
          point number (P1 & P1! & P1!!);
```

INSTRUCTION EXAMPLES:

A.2.24 GPP instruction group %24

NOTATION:
```
NOTATION:     PC <== P3,
              PFR <== PFBR;
```

DESCRIPTION: Data from or addressed by P3 is deposited into the program conuter, PC. The contents of the program field buffer register, PFBR, is deposited into the program field register, PFR. The next GPP instruction is addressed by the new contents of the PC and PFR.

GROUP %24 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P3	:?:	.?.	G%24B1
´P3	:?:	.?.	G%24B2

OPERATIONS:

```
JUMP              ALU code #?      ALU none          ALU cycles none
----     PC <== P3; PFR <== PFRB
```

INSTRUCTION EXAMPLES:

A.2.25 GPP instruction group %25

```
NOTATION:     PDL(PDLCTR) <== PC and PFR;
              PC <== P3,
              PFR <== PFRB,
              PDLCTR <== PDLCTR + 1;
```

DESCRIPTION: The program counter, PC, and the program field register, PFR, are stored in the push down list as a funstion of the push down list address counter, PDLCTR. Then data from or addressed by P3 is deposited into the program conuter, PC, and the program field buffer register, PFBR, is deposited into the program field register, PFR. The next GPP instruction is addressed by the new contents of the PC and PFR. Finally the PDLCTR is incremented.

GROUP %25 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P3	:?:	.?.	G%25B1
´P3	:?:	.?.	G%25B2

OPERATIONS:

```
PUSHJ          ALU code #?      ALU none          ALU cycles none
------   PDL( PDLCTR) <== PC;
         PC <== P3,
         PFR <== PFRB,
         PDLCTR <== PDLCTR + 1;
```

INSTRUCTION EXAMPLES:

A.2.26 GPP instruction group %26

```
NOTATION:      PDLCTR <== PDLCTR - 1;
               PC and PFR <== PDL(PDLCTR);
```

DESCRIPTION: First the push down list address counter, PDLCTR, is decremented. Then the program counter, PC, and program field register, PFR, are loaded with the push down list data addressed by the PDLCTR. The next GPP instruction is addressed by the new contents of the PC.

GROUP %26 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
P1,P2,P3 not used	:?:	.?.	G%26B1

OPERATIONS:

```
POPJ              ALU code #?      ALU none          ALU cycles none
------  PDLCTR <== PDLCTR - 1;
        PC and PFR <== PDL(PDLCTR);
```

INSTRUCTION EXAMPLES:

A.2.27 GPP instruction group %27

NOTATION: MMADR <== P1;
execution proceeds as per the microprogram addressed.

DESCRIPTION: The contents of P1 is deposited into the mapping memory address register. The microprogram then proceeds from the addressed address of the mapping memory. If all starting address locations are used up in the OPR word then the programmer may use this instruction to define additional complex microprograms.

GROUP %27 ADDRESS MODE TABLE:

Address Mode	MM Adr	M Cyc	MP Label
#P1	:?:	.?.	G%27B1

OPERATIONS:

```
APPLY            ALU code #?      ALU ?            ALU cycles none
------   MMADR <== P1;
                 Execution proceeds as per the microprogram
                 addressed by P1. Control continues in the
                 program at the next instruction unless the
                 microprogram changes the PC.
```

INSTRUCTION EXAMPLES:

APPENDIX B

GPP I/O registers and control panel lights/switches

Appendix B contains tables of GPP GR space addresses for interface registers, lights, switches etc.

B.1 Status registers addressed by the GPP

The GPP data address space 170000:170077 is reserved for status registers. The GPP data address space 170400:170777 is reserved for I/O registers. These addresses may be used for internal I/O, external I/O, switches, display lights, X-Y coordinates, etc. The space is allocated as follows:

```
170000   A0D[4:15] - autodecrement register 0.
                   .
                   .
                   .
170007   A7D[4:15] - autodecrement register 7.

170010   A0[4:15] - auto-unmodified register 0.
                   .
                   .
                   .
170017   A7[4:15] - auto-unmodified register 7.

170020   A0I[4:15] - autoincrement register 0.
                   .
                   .
                   .
170027   A7I[4:15] - autoincrement register 7.

170030   PDLPTL[0:15] - stack pointer to 32-bit 1K PDL, read only
                   low 16-bits.

170031   PDLPTH[0:15] - stack pointer to 32-bit 1K PDL, read only
                   high 16-bits.
         Read only.

170032   SW1[0:15] - GPP front panel switch register 0:15.

170033   SW2[0:15] - GPP front panel switch register 16:31.

170034   SW3[0:15] - GPP front panel switch register 32:47.

170035   DSPLYA[0:15] - 6 digit (octal) lights, control desk.

170036   DSPLYB[0:15] - 6 digit (octal) lights, control desk.

170037   DSPLYC[0:15] - 6 digit (octal) lights, GPP front panel.
```

170040	KNOB0[6:15]	- A/D ctrl/desk knob pot 0.
170041	KNOB1[6:15]	- A/D ctrl/desk knob pot 1.
170042	KNOB2[6:15]	- A/D ctrl/desk knob pot 2.
170043	KNOB3[6:15]	- A/D ctrl/desk knob pot 3.
170044	KNOB4[6:15]	- A/D ctrl/desk knob pot 4.
170045	KNOB5[6:15]	- A/D ctrl/desk knob pot 5.
170046	KNOB6[6:15]	- A/D ctrl/desk knob pot 6.
170047	KNOB7[6:15]	- A/D ctrl/desk knob pot 7.
170050	SWA[0:15]	- control desk switch register
170051	STATUS[0:15]	- GPP/PDP8e status register (STATG1, STATG2 in PDP8e).

B.2 ALU registers

170100	ALUA0	- ALU A register 0 low order 16-bits.
170101	ALUA0	- ALU A register 0 high order 16-bits.
170102	ALUA1	- ALU A register 1 low order 16-bits.
170103	ALUA1	- ALU A register 1 high order 16-bits.
170104	ALUA2	- ALU A register 2 low order 16-bits.
170105	ALUA2	- ALU A register 2 high order 16-bits.
170106	ALUA3	- ALU A register 3 low order 16-bits.
170107	ALUA3	- ALU A register 3 high order 16-bits.
170110	ALUA4	- ALU A register 4 low order 16-bits.
170111	ALUA4	- ALU A register 4 high order 16-bits.
170112	ALUA5	- ALU A register 5 low order 16-bits.
170113	ALUA5	- ALU A register 5 high order 16-bits.
170114	ALUA6	- ALU A register 6 low order 16-bits.
170115	ALUA6	- ALU A register 6 high order 16-bits.
170116	ALUA7	- ALU A register 7 low order 16-bits.
170117	ALUA7	- ALU A register 7 high order 16-bits.

170120	ALUA10	- ALU A register 10 low order 16-bits.
170121	ALUA10	- ALU A register 10 high order 16-bits.
170122	ALUA11	- ALU A register 11 low order 16-bits.
170123	ALUA11	- ALU A register 11 high order 16-bits.
170124	ALUA12	- ALU A register 12 low order 16-bits.
170125	ALUA12	- ALU A register 12 high order 16-bits.
170126	ALUA13	- ALU A register 13 low order 16-bits.
170127	ALUA13	- ALU A register 13 high order 16-bits.
170130	ALUA14	- ALU A register 14 low order 16-bits.
170131	ALUA14	- ALU A register 14 high order 16-bits.
170132	ALUA15	- ALU A register 15 low order 16-bits.
170133	ALUA15	- ALU A register 15 high order 16-bits.
170134	ALUA16	- ALU A register 16 low order 16-bits.
170135	ALUA16	- ALU A register 16 high order 16-bits.
170136	ALUA17	- ALU A register 17 low order 16-bits.
170137	ALUA17	- ALU A register 17 high order 16-bits.

B.3 GPP line buffer and BM I/O registers

Various I/O registers are used in the program I/O subroutines for the GPP.

170400	I1XM[8:15]	- I1 X-1 dynamic address vector.
170401	I1X[8:15]	- I1 X dynamic address vector.
170402	I1XP[8:15]	- I1 X+1 dynamic address vector.
170403	I1Y[8:15]	- I1 Y line counter.

170404	I2XM[8:15]	- I2 X-1 dynamic address vector.
170405	I2X[8:15]	- I2 X dynamic address vector.
170406	I2XP[8:15]	- I2 X+1 dynamic address vector.
170407	I2Y[8:15]	- I2 Y line counter.

170410	I3XM[8:15] - I3 X-1 dynamic address vector.
170411	I3X[8:15] - I3 X dynamic address vector.
170412	I3XP[8:15] - I3 X+1 dynamic address vector.
170413	I3Y[8:15] - I3 Y line counter.

170414	GIN[0:15] - is a source P1 or P2 address which will cause a 16-bit word to be read from the previously initiated PDP8e general DMA I/O channel.
170415	GOUT[0:15] - is a P3 destination address which will cause a 16-bit word to be written out to the previously initialized PDP8e general DMA I/O channel.

170420	KRB [8:15] - read byte from the teletype input channel and clear the TTY input ready flag.
174021	KSTATUS[0] - 1 if data available, 0 if no data present.
170422	TLS[8:15] - send byte to teletype output channel and clear the TTY output not ready flag.
170423	TSTATUS[0] - 0 if TTY output channel not ready, 1 if TTY output channel ready.

170430	PBM0 - indirect pointer register to BM0.
170431	PBM1 - indirect pointer register to BM1.
170432	PBM2 - indirect pointer register to BM2.
170433	PBM3 - indirect pointer register to BM3.
170434	PBM4 - indirect pointer register to BM4.
170435	PBM5 - indirect pointer register to BM5.
170436	PBM6 - indirect pointer register to BM6.
170437	PBM7 - indirect pointer register to BM7.

173400:173777	I1[y-1]	direct line buffer address
174000:174377	I1[y]	direct line buffer address
174400:174777	I1[y+1]	direct line buffer address
175000:175377	I2[y-1]	direct line buffer address
175400:175777	I2[y]	direct line buffer address
176000:175377	I2[y+1]	direct line buffer address
176400:176777	I3[y-1]	direct line buffer address
177000:177377	I3[y]	direct line buffer address
177400:177777	I3[y+1]	direct line buffer address

B.4 GPP front panel controls

The GPP has a front control panel with the following knobs, lights, and switches to control the GPP directly rather than through the PDP8e for maintenance and hardware debugging.

B.4.1 Lights on the GPP front panel

1. PM[0:59] octal (5 groups of 4 digits) which is loaded when each instruction is fetched from the PM.

2. PC[0:15] octal (6 digits) address register is constantly displayed.

3. STATUS[0:15] - trap register. Note any trap will set the appropriate bit in the status register, and halt the GPP. The PDP8e can sense and service this condition. The bits are allocated as follows:
   STATUS[0] = PC address trap.
   STATUS[1] = data address bus trap.
   STATUS[2] = PDL overflow [>1024].
   STATUS[3] = PDL underflow (POPJ empty stack).
   STATUS[4] = GPP run FF.
   STATUS[5] = PDP8e test/GPP set.
   STATUS[6] = GPP test/set PDP8e.
   STATUS[7] = illegal GPP instr. addr. mode bits.
   STATUS[8] = illegal PM address
   STATUS[9] = illegal P1 address
   STATUS[10] = illegal P2 address
   STATUS[11] = illegal P3 address

4. DSPLYC[0:15] - 6 octal digits (DSPLYC)

5. EXAR&DRA[0:31] - data register A, 12 octal digits.

6. EDRB&DRB[0:31] - data register B, 12 octal digits.

7. Data address bus trap DABTRP[0:15] - 6 octal digits

8. PC address bus trap PCTRP[0:15] - 6 octal digits.

9. Group OPR lights GROUP[0:3] - 2 octal digits.

10. GPP microprogram control program counter MCPC[0:11] - 4 octal digits

11. GPP microprogram control bus [0:59] - 12 octal digits

APPENDIX C

PDP8e I/O transfer instructions for the RTPP

PDP8e input output transfer instructions (called IOTs) used in the RTPP are given here. They are explained in sets of IOTs associated with particular subsections of the RTPP.

C.1 PDP8e IOTs for RQC - QMT - stage - control desk

1. TB, TC - Standard Detector 12-bit threshold. These select the level of detection in the standard detector.

   DETB - load threshold detector TB.
   DETC - load threshold detector TC.

2. T1, T2 - 1-D detector 12-bit thresholds. These select the Level of detection in the 1-D detector. The data lines are allocated but the interface to the 1-D detector is not built since to do so would be redundent with programming the Digitizer/detector thresholds.

   DET1 - load threshold 1.
   DET2 - load threshold 2.

3. T1, T2 - Densitometer (Digitizer/Detector) 6-bit threshold. These select the level of detection in the densitometer.

   DETDIG - load bits 0-5 to threshold T1, and load bits 6-11 to threshold T2.

4. HP,HS,VP,VS - Frame and Scale coordinates, size is S, position is P. These four coordinates determine the variable frame and are loadable/readable. since they are on an independent up/down counter, they are enabled by the PDP8e. Note the Frame and Scale position and size switches must be set to 0 for remote (PDP8e controlled) operation.

   HPR - read BCD horizontal position.
   HSR - read BCD horizontal size.
   VPR - read BCD vertical position.
   VSR - read BCD vertical size.
   HPL - load buffer to horizontal position.
   HSL - load buffer to horizontal size.
   VPL - load buffer to vertical position.
   VSL - load buffer to vertical size.

5. (XL,YL) - light pen "touch" coordinates which may be loaded to overide the light pen or be read from the light pen module to determine where the user touched

the screen with the light pen. The pen X and Y coordinates are 3 digit BCD (12-bits). This controller has not been built since the Graf-Pen/Cursor combination with the buffer memories will achieve the same effect. NOTE: *** NOT IMPLEMENTED***.

PENST - load pen status (normal/remote) register.
RPENX - read the current normal X pen coordinate.
RPENY - read the current normal Y pen coordinate.
LPENX - load the current remote X pen coordinate.
LPENY - load the current remote Y pen coordinate.

6. DET-ENTRY[1:720], DET-EXIT[1:720]) - mask register Coordinates in the range of X (0 to 1024). This set of 2-tuples defines the run-end code of the mask.

GETMSK - enable filling the mask register with 720 ent/ext points starting at the next STQMT.
MSKADR - load the 8 bit mask relative address from acc.
RMASKE - read 10-bit binary e(MSKADR) mask reg==>ac.
RMASKX - read 10-bit binary x(MSKADR) mask reg==>ac.
LMASKE - load 10-bit binary e(MSKADR) mask reg<==ac.
LMASKX - load 10-bit binary x(MSKADR) mask reg<==ac.
LDXP - load the X QMT cursor register<==C(AC)
LDYP - load the Y QMT cursor register<==C(AC)

7. Shift register for acquiring QMT function computer data. (FC1[1:1024], FC2[1:1024], XACP[1:1024],YACP[1:1024], DET[1:1024]) - The 5 field QMT data "shift" register (With PDP8e controlled recycle) is filled with data on the occurance of an ACP. The FC1 and FC2 are Function computers 1 and 2 whose data is available at each ACP in parallel. The FC1 and FC2 are read only (PDP8e) while the XACP, YACP, DET are read/ write (PDP8e). See QSTAT enables for use with SRG.

RSRGI - read the SRG index counter (SRGI).
RFC1H - read FC1 front data high (3 digits)
RFC1L - read FC1 front data low (3 digits)
RFC2H - read FC2 front data high (3 digits)
RFC2L - read FC2 front data low (3 digits)
RSRGX - read X (10-bit binary) ACP front data.
RSRGY - read Y (10-bit binary), ACP/DET(1 bit - bit 0)
LSRGB - load ACP buffer (10-bits)
SMACP - simulate an ACP.
SMCLK - simulate a QMT clock.
SMHLD - simulate a QMT hold.
SMSYN - simulate a QMT sync.
SMVTG - simulate a QMT vtrig.
ADVSR - advance the SRG by 1.
CSRGI - clear the SRG index register.
ZSRGI - zero SRG index counter, and advance the SRG.
IZSKP - skip on SRG index register zero.

8. SIZEA - sizing on the Amender module. Used in Amending

operation. Note sizing switches must be set to 0.

SIZEA[0:11] - load 3 digit BCD into size register.

9. SIZEC - sizing on the Classifier/Collector module. Load the classifier upper/limit register. Note the mantissa is 2 left BCD digits and exponent is right BCD digit. To load the lower limit register, load the upper register then do a QPROG7[10] to transfer the upper to lower. Note: the size switches must be set to 0.

SIZEC[0:11] - load 3 digit BCD into size register.

10. SIZEM - sizing on the MS3 Computer module. Used in determining min or max chord size to be computed. Note: the sizing switches must be set to 0.

SIZEM[0:11] - load 3 digit BCD into size register.

11. SIZES - sizing on the Standard Computer module. Used in determining min or max chord size to be computed. Note: the sizing switches must be set to 0.

SIZES[0:11] - load 3 digit BCD into size register.

12. STQMT - syncronize the PDP8e with the QMT and start the QMT for 1 scan. Also used for GETMSK, GETA, and GETB.

13. QMSKP - SKIP if the QMT scan is done (used after STQMT).

14. Quantimet program word described in Appendix C.3 - QPROG1[0:11], QPROG2[0:11], QPROG3[0:11], QPROG4[0:11], QPROG5[0:11], QPROG6[0:11], QPROG7[0:11], QPROG8[0:11]

15. Load the PDP8e/QMT status register described in Appendix C.3 (** Old GC1.1 QST4**) which controls the QMT/PDP8e interface. Note that QSTAT is displayed in octal on the control desk.

QSTAT[0:11] - load the QMT status register.
RQSTAT[0:11] - reads QSTAT into the PDP8e accumulator.

16. Read the Quantimet full field data accumulator register (7 BCD digits).

QDAT1[0:11] - read least significant BCD 3 digits
QDAT2[0:11] - read middle 3 BCD digits
QDAT3[0:11] - read high 1 BCD digit

17. Load the QMT right display 28 bits of BCD data

LQDT1 - load least significant BCD 3 digits
LQDT2 - load middle 3 BCD digits.
LQDT3 - load high 1 BCD digit.

18. PDP8e 200 hz clock.

CLKACK - clear the clock
CLKSKP - skip on clock active.

19. The following are control desk switches from the PDP8e part of the control desk (Figure 3).

```
FBW1  - read z,y,x joystick/focus (bits 3 4 5 speed,
        Bits 6 7=z, 8 9=y, 10 11=x).
FBW2  - read the 12 command keys.
LFBW2 - load the 12 command key lights.
FBW3  - read the 12 momentary class keys.
FBW4  - read the 12 on/off toggle switches.
FBW5  - read digiswitch octal digits 0 1 2 3
FBW6  - read digiswitch octal digits 4 5 6 7
FBW7  - read digiswitch octal digits 8 9 10 11
FBW10 - read 3 5-position switches TH1, TH2, zoom.
        Same format as FBW1.
FBW11 - read 3 5-position switches freq, intens, spare1.
        Same format as FBW1.
FBW12 - read 5-position switch spare2 (speed bit 3,
        Motions bits 6 7), and momentary execute bit 0,
        Graf-Pen tip switch (pressed down is on) bit 11.
```

20. The following are the general purpose display lights on the right control desk. The display is decoded as both an octal 4-digit number and a 3-digit BCD number.

```
DISP1[0:11] - load left display leds which appear in 4 digit
        Octal as well octal as 3 digit BCD.
DISP2[0:11] - load right display leds which appear in 4 digit
        Octal as well octal as 3 digit BCD.
```

21. Load the galvanometer motor mirror scanner (x,y) registers. These commands use thee normal signed Cartesian coordinate system with (X0,Y0) =(0,0). Negative numbers are 2's complement. The scanner is set to scan a 1024 x 1024 pixel array. This is actually (-512 to +512) by (-512 to +512). Note that a transformation of the coordinates is necessary to map them to those of the QMT. The D/A may be used in either 10-bit or 11-bit mode by changing a hardwired jumper.

```
LGALX[1:11]  - C(AC)==>GALX, 0==>C(AC).
GALSKP          - skip when the galvanometer scanner is ready
RGAL[1:11]  - C(AC)<== integrated scanner data.
LGALY[1:11]  - C(AC)==>GALY, 0==>C(AC).
```

22. Zeiss stage, focus and zoom stepping motors.

MSTAG - load the 6 stepping motor direction word as 2-bit pairs (down,up) or (-,+) (wavelength, intens, zoom, focus, y, x)

Bits	function
0	+ wavelength
1	- wavelength
2	+ neutral density
3	- neutral density
4	+ zoom
5	- zoom
6	+ focus
7	- focus
8	+ y stage
9	- y stage
10	+ x stage
11	- x stage

SMOTR - load the spare stepping motor direction word as 2-bit pairs (down,up) or (-,+) pairs.

Bits	function
4	+ spare 1
5	- spare 1
6	+ spare 2
7	- spare 2

STEP - strobe the stepping motor sequence generator from the MSTAG and SMOTR registers.

23. GRAF-PEN interface supplied for model GP-2. The interface data sheet is for model 1353 which is presumed to be GP-2. There are 4 instructions of which 2 are microcoded to give the 4th.

GRFSKP	= 6141	skip when pen ready with data.
GRFRXY	= 6142	read the next x or y datum.
GRFINC	= 6144	increment the datum pointer to next x or y. to y if it was x, to x if it was y.
GRFRI	= 6146	read next datum, then increment.

A program example shows how these instructions are used.

```
GRFSKP   /skip when data ready.
JMP .-1
GRFRI    /read x and increment pointer
DCA XDATA /save data
GRFRI    /read y and increment pointer
DCA YDATA /save data
```

24. Analogue to Digital 16 channel converter (DEC AD8-ea/AM8-ea) with an input voltage range of +/- 1 volt, 0-30 KHZ bandwidth, 10-bit resolution, 200 nsec. apperature time.

ADCL 6530 Clear AD done and timing error flags.
Clear enable, mux and status register.

```
ADLM 6531 Load mux register from AC[811], clear AC.
ADST 6532 Clear AD done and timing error flags.
                Start AD converter. Channel to
                be converted is to be determined by
                mux register.
ADRB 6533 Clear AD done flag. Contents of
                AD buffer ==>ac[0:11].
ADSK 6534 Skip next instruction if AD done=1.
                Do not clear flag.
ADSE 6535 Skip next instruction if timing error=1.
                Do not clear flag.
ADLE 6536 Load enable register from AC[2:5].
ADRS 6537 Read AD status/enable register and
                mux into AC[0:11].
```

Status register

Bit	function
0	Conversion done
1	error
2	done interrupt enable
3	error interrupt enable
4	external start enable
5	auto increment enable
6-7	not used
8-11	contents of mux register

Input channel Allocation
----

0 - galvanometer scanner data
1 - galvanometer scanner ref. beam (when implemented)
[8:15] - control desk knobs [0:7]

25. Control desk key-pad input. The control desk has a 6 digit BCD key pad with clear C and send S keys. Pressing the send key is sensed with the skip instruction.

RKYPDH[0:11] - read high 3 BCD digits.
RKYPDL[0:11] - read low 3 BCD digits.
SKPKPD - skip on S key and clear flag. Note that key
        C being pressed clears the S key as well.

26. Dicomed 31 interface for the PDP8e/Dicomed. (C.f. subroutine DICMED.FT for handler specifications).

DICSKP = 6101 - skip on Dicomed read for next input
DICLR = 6102 - clear the Dicomed to make it ready
DICO[0,3:11] = 6106 - send command in the AC. Command
        bit in AC[0] and command/data in AC[3:11].

27. External I/O interface which adds groups of 8 input and 8 output 12-bit TTL channels to the RTPP PDP8e.

EXADR[0:11] - load channel select register with channels 0:4095.
EXIN[0:11] - C(AC)<== C(selected channel)
EXOUT[0:11] - C(AC)==>C(selected channel)

Input channel allocation
----

0 - day of year in BCD (100,10,1)
1 - hour (10,1), minutes (10) in BCD
2 - minutes (1), seconds (10,1) in BCD
3 - patch pannel 12-bit input.

Output channel allocation
----

0 - 12-bit relay register
3 - patch pannel 12-bit output

C.1.1 Allocation of Quantimet program QPROG words

The RTPP Quantimet has been modified so as to microprogrammable from the PDP8e computer using a set of eight program words QPROG[1:8]. The following table of QPROG words describes the control facilities. Features not yet implemented are marked with a *.

```
*QPROG1[0:4] = 6-bits detector module (normally leave in
        slice mode select 3 (slice B/C))
        11000= select 1
        10100= select 2 threshold A
        10010= select 3 threshold B
        10001= select 4 threshold C

QPROG1[6:9] = 4 bits Standard computer
        1000 = area
        0100 = intercept
        0010 = count
        0001 = ????
        0000 = off

*QPROG1[10:11] = 2 bits Digitizer/Detector module
        01 = density
        10 = area
        11 = intercept
        00 = off

QPROG2[0:3] = FRAME 2 output select as a function of
        (Mask register output, FRAME 1 output)
        The mask register select performs all 16 of the
        SN74S181  ALU  logic  operations.
        1111 = NOT FRAME 1
        1110 = NOT (FRAME 1 OR MR)
        1101 = (NOT FRAME 1) AND MR
        1100 = 1 (FALSE)
        1011 = NOT (FRAME 1 AND MR)
        1010 = NOT MR
        1001 = FRAME 1 XOR MR
        1000 = FRAME 1 AND (NOT MR)
        0111 = (NOT FRAME 1) OR MR
        0110 = NOT (FRAME 1 XOR MR)
        0101 = MR
        0100 = FRAME 1 AND MR
        0011 = 0 (TRUE)
        0010 = FRAME 1 AND (NOT MR)
        0001 = FRAME 1 OR MR
        0000 = FRAME 1

QPROG2[4] = 1 bit digitizer module.
        0 = log mode
        1 = linear mode

QPROG2[6:9] = FRAME 1 output select as a function of
        (Variable frame and Mask Buffer memory Frame)
        The mask register select performs all 16 of the
        SN74S181   ALU  logic  operations.
        1111 = NOT VAR FRAME
        1110 = NOT (VAR FRAME OR MASK REG)
        1101 = (NOT VAR FRAME) AND MASK REG
        1100 = 1 (FALSE)
        1011 = NOT (VAR FRAME AND MASK REG)
        1010 = NOT MASK REG
        1001 = VAR FRAME XOR MASK REG
        1000 = VAR FRAME AND (NOT MASK REG)
        0111 = (NOT VAR FRAME) OR MASK REG
        0110 = NOT (VAR FRAME XOR MASK REG)
        0101 = MASK REG
        0100 = VAR FRAME AND MASK REG
        0011 = 0 (TRUE)
        0010 = VAR FRAME AND (NOT MASK REG)
        0001 = VAR FRAME OR MASK REG
        0000 = VAR FRAME

QPROG2[10:11] = not used...

QPROG3[0:3] = 4 bits amender module.
        1011 = excess
        1111 = isolate
        0111 = cluster
        0101 = fill
        1101 = agglomerate
        1001 = smear
        0001 = unmodified

QPROG3[4:7] = 4 bits MS3 computer
        1000 = area
        0100 = intercept
        0010 = count
        0001 = perimeter
        0000 = off

QPROG3[8:11] = not used

QPROG4[0:6] = not used

QPROG4[7:9] = MS3 computer keys
        001 = key 1
        010 = key 2
        100 = key 3

*QPROG5[0:2] = Modified/unmodified switch (1/0)
        001 = MS3 modified/unmodified (1/0)
        010 = Funct. Comp. 1 modified/unmodified (1/0)
        100 = Funct. Comp. 2 modified/unmodified (1/0)

QPROG5[3:5] = Classifier/collector
        100 = function 1 select
        010 = function 2 select
        001 = count number of acceptable ACPs as select

*QPROG5[6:7] = MS3 display toggles
        00 = MS3 computer display
        01 = MS3 computer display and paralysis
        10 = MS3 computer paralysis display

*QPROG5[8:11] = other display toggles
        0001 = Amender display
        0010 = Standard detector display
        0100 = Digitizer detector display
        1000 = Standard Computer display

*QPROG6[0:5] = Display pushbuttons on modules
        000001 = classifier top display
        000010 = classifier bottom display
        000100 = Standard computer display
        001000 = MS3 computer display
        010000 = Function computer 1 display
        100000 = Function computer 2 display

*QPROG6[6:11] = 1-D detector module
        000001 = 1
        000010 = 2
        000100 = 3
        001000 = field standard
        010000 = specimen standard
        100000 = auto delineator
        000000 = off

QPROG7[0:1] = Mask register display enables
        00 = off
        01 = perimeter display
        10 = area display
        11 = live frame (generated by QPROG2[0:3]).

QPROG7[2:5] = Function computer 1 program word
        0001 = volume (integrated density)
        0010 = area
        0011 = perimeter
        0100 = vertical projection
        0101 = horizontal projection
        0110 = horizontal Feret
        0111 = vertical Feret

QPROG7[6:9] = Function computer 2 program word
        0001 = volume (integrated density)
        0010 = area
        0011 = perimeter
        0100 = vertical projection
        0101 = horizontal projection
        0110 = horizontal Feret
        0111 = vertical Feret

QPROG7[10] - Classifier move limit transfer.
        when this bit is set to 1 (0 to 1) it moves
        the contents of the classifier upper limit register
        to the contents of the lower limit register when
        the Classifier "Transfer" switch is "up".
```

C.1.2 Quantimet shift register control

The actual counting of objects is done in hardware by a pair of special purpose 1024 word (69 bits/word) hardware static shift registers (SRG). A word has 5 fields (x, y, detected, Function Computer 1 data, Function Computer 2 data) with (10, 10, 1, 24, 24) bits respectively. The SRG can accept data at a 2 mhz rate. This rate was used so that there would be no timing problem with the detected objects coming from the quantimet at a 8 mhz scan rate (at a minimum of 5 picture points apart or 1.6 mhz). One bit of the y coordinate data is designated the density bit. In addition to the SRG there is an index (counter) register, I/O buffers, x and y coordinate counters, and associated control gates and flip flops with which to control the data paths of the SRG complex.

Another part of the interface allows data to be read from the front of the SRG and data to be loaded into the rear of the SRG. The QMT can load the SRG as described in 1 and 2 below. The SRG may be operated as a circular queue utilizing a recycle mode (to move data from the front of the SRG to the rear of the SRG) or as a regular queue not using recycle. Data may be advanced to the front of the SRG one (x,y,density) triple at a time, or valid data may be automatically advanced to the front of the SRG by advancing the SRG (1024 less (SRG-index-register mod 1024)).

The SRG can accept data under three conditions.

1. It is enabled to accept anti-coincidence points (blobs) (ACP's) from the Quantimet in which case it will push the running QMT display (x,y) coordinates for that blob into the rear of the SRG and increment the SRG index register by one.

2. It compares the front (x,y) coordinate of the SRG (the SRG is now thought of as a queue) with the running QMT (x,y). If they are equal, it pushes the (x,y) coordinate again into the rear of the SRG except that an additional bit in the y SRG is set to 1 if the detector is on (i.e. in blob) or to 0 if the detector is off (in background).

3. The PDP8e can read (x,y,density) data from the front of the queue and it can load (x,y,density) data into the rear of the SRG it can read and clear the SRG index register as well.

C.1.3 The QMT shift register commands

The SRG is a 69 bit wide 1024 word shift register with the ability to recycle data. This recycle control is governed by bit 1 of QSTAT. Binary data may enter the rear of the SRG from either the QMT running (x,y) coordinate counters or from the PDP8e. Binary data may be read from the front of the SRG by the PDP8e by advancing the data to be read to the front of the SRG and then reading it.

A SRG index register is used to count (in binary) the number of (x,y) pairs in the SRG. The index register may be read and cleared by the PDP8e. Therefore, bit 0 of the y coordinate (11 bits come into the PDP8e) is assigned to be the density bit. If it is a 0, then the QMT detector was active at the time (x,y) was pushed into the SRG. If it is 1, then the detector was inactive (undetected). If a (x,y) pair was pushed on an ACP, it will be high by definition of ACP.

ZSRGI

ZSRGI zeros the shift register index counter. This "advances" the contents of the shift register to the front output buffers and sets up the data in the SRG either for removal or for the xy background compare 2nd pass with the Quantimet. This is accomplished by advancing the SRG and incrementing the SRG index register to 0 modulo 1024. If there are n points in the rear of the SRG, then the SRG data need be advanced (1024-n) times. Note that the content of the SRG index register after a ZSRGI will be multiples of 1024. So generally a CSRGI would be used after doing a ZSRGI. The following example shows how one would normally use the ZSRGI:

```
RSRGI /read and save the index counter
CMA
DCA saveindexcounter
ZSRGI /advance the SRG data to the front.
IZSKP /done advancing?
JMP .-1  /no
CSRGI /clear out the high order (multiples of 1024) bits.
```

CSRGI

CSRGI clears the shift register index register bits [0:11]. This clears only the SRG "index register".

ADVSR

ADVSR advances the shift register (SRG) one position and increment the SRG "index register" by one. ADVSR is used to advance SRG (x,y,density) data to the front of the SRG so that it can be read, and to enter it into the rear of the SRG.

IZSKP

IZSKP skips if the SRG "index register" is zero -. It is used to test whether bits 4:11 of the shift register index register are 0. The register is 12-bits long.

STQMT

STQMT starts the Quantimet with the program previously loaded into the QMT program words QPROG1-8. It initiates the Quantimet by issuing the "AUTO" signal to the QMT. It also SYNChronizes the PDP8e with the QMT VTRIG and QMT HOLD signals.

QMSKP

QMSKP skips if the Quantimet is not busy. i.e. the Quantimet has finished a complete scan initiated by the STQMT instruction and executed the program word given it. Note that QMT data (QDAT1, QDAT2, QDAT3) is active for 60 microseconds after the skip (from QMSKP) occurs.

C.1.4 Quantimet control signals

Signals generated by the QMT are used by the QMT/PDP8E RQC controller to control the operation of the "blob-catcher" shift register. These Quantimet signals are enabled at the RQC controller if the status register QSTAT[5] is 0. Otherwise the simulated instructions can be used.

VTRIG

Is generated by the QMT at the beginning of each scan. Data from the last scan is available 60 microseconds after this pulse. Note that VTRIG also is used in the RQC controller to clear the y coordinate counter. (By loading LSRGB ==> y coordinate counter).

SYNC

Is generated by the QMT at the end of each horizontal scan line. It is used by the RQC controller to load zero into the x coordinate counter and increment the y coordinate counter by 1.

CLOCK

Is generated by the QMT every 125 nanoseconds and is used by the RQC controller to upcount the x coordinate register.

ACP (gated count)

Is generated by the QMT MS3 standard computer or Classifier-collector/Function computers and results from an object being detected in full feature mode. It is used by the RQC controller to strobe (x, y, density, Function computer data 1, Function computer data 2) coordinate data into the shift register. The "count out" signal is to be prefered to using

"ACP" as it uses the MS3 standard computer sizing whereas "ACP" does not. Note that the ACP signal occurs (x+8,y+1) picture points after the last detected point of an object. This vector is subtracted from the blue blob stack to give the "detected" blob stack. This is used in the xy compare pass.

Detected video

Is a level signal generated by the detector module and is used during the 2nd "background compare" pass of the 3-D "grain counting" to signal whether a previous (x,y) pair (ACP) is detected now (in a particular wavelength light) or not.

Live frame

Is a signal from the frame and scale module. It is used to "and" the ACP's coming in so that they are inside of the live frame.

AUTO

Is a signal generated by the PDP8E RQC controller to signal the QMT to begin accumulating data during the upcomming scan. The PDP8E RQC controller synthesizes "AUTO" from "VTRIG" and "HOLD".

C.1.5 Shift register simulated operation

The QMT/shift-register interface can be simulated from the PDP8e by four instructions. The procedure is to disable the QMT signals CLOCK, BV FRAME, VTRIG, HOLD, ACP (GATED COUNT), SYNC, and detected video. Then, PDP8e signals are used to simulate the Quantimet. QSTAT[5] is an enable(0) or disable(1) for the Quantimet signals to control the shift register logic. That is, disabling the QMT enables the simulated signals. This enables the operation of a test program to simulate the ACP/coordinate catcher.

SMCLK

SMCLK simulates the Quantimet CLOCK. The CLOCK is used to countup the x coordinate register.

SMSYC

SMSYC loads the LSRGB register into the x coordinate register, and increments the y coordinate register by 1. Normally, the LSRGB contains zero, so that doing a SMSYC will load zero into the x-coordinate register.

SMVTG

SMVTG simulates the Quantimet vertical trigger to load the y coordinate register with the contents of the LSRGB register. Normally the LSRGB contains zero. So effectively, it will zero y-coord.

SMACP

SMACP simulates the ACP signal for use as input to the shift register interface.

SMHLD

SMHLD simulates a QMT hold signal. This instruction is only used in maintenance programs.

The detector level is simulated by QSTAT[4] status register: 1 indicates no detection, 0 indicates detection. This only applies if QSTAT[4]=1 (i.e. in simulation mode).

C.1.6 Reading and loading the SRG

The SRG is read from the front and loaded from the rear.

Reading the SRG

The three SRG registers index, x front of the SRG, and y front of the SRG may be read directly into the PDP8E accumulator.

RSRGI

RSRGI reads the 12-bit shift register index register used to count the number of ACP's collected so far in the shift register. Note that successive data acquisitions of ACP's will count up RSRGI unless it is cleared after it is read each time with a CSRGI.

RSRGX

RSRGX reads the front of the "x" shift register for x ACP coordinates. It is 10-bits of x coordinate right justified binary.

RSRGY

RSRGY reads the front of the "y" shift register for y ACP coordinates. It is 10-bits of y coordinate right justified binary. The detected bit is bit 0 and is 1 if the ACP was in the detected region otherwise bit 0 is 0.

RFC1H

RFC1H reads the high order 3 digits of BCD Function computer 1 data into the PDP8e AC.

RFC1L

RFC1L reads the low order 3 digits of BCD Function computer 1 data into the PDP8e AC.

RFC2H

RFC2H reads the high order 3 digits of BCD Function computer 2 data into the PDP8e AC.

RFC2L

RFC2L reads the low order 3 digits of BCD Function computer 2 data into the PDP8e AC.

Loading the SRG

The SRG may be loaded by using the following algorithm:

[0] clear the SRG index register with a CSRGI.

[1] disable the QMT CLOCK, SYNC, VTRIG by setting QSTAT
Bit 5 to 1 (disable). This enables the PDP8E to
Run the SRG control logic.

[2] for each (x,y) coordinate pair, load the x and y
Coordinate counters with the following commands:

LSRGB

LSRGB loads the SRG binary input buffer from the PDP8E
accumulator.

[2.1] the x coordinate binary counter is loaded by the
Following sequence of PDP8e code:

```
CLA     /clear the ACC
TAD x   /load x coord into the AC
LSRGB    /put it into the SRGb buffer
SMSYN    /load x-coord register from buffer
         /(And increment y-coord register
         /By 1. The latter is irrelevant)
```

[2.2] the y coordinate binary counter is loaded by the
Following sequence of PDP8e code:

```
CLA      /clear the ACC
TAD y    /load y coord into the AC
LSRGB    /put it into the SRGb buffer
SMVTG    /load y-coord register from buffer.
ADVSR    /shift the (x,y) data ==> SRG.
```

[2.3] put zero back into the SRGb buffer so that the QMT
Signals VTRIG and SYNC will operate properly.
(VTRIG, SYNC loads 0). The following code
Will do this:

```
CLA /0 TO AC
LSRGB /load c(AC) into the SRGb buffer.
```

C.1.7 Allocation of status register QSTAT

The shift register (SRG) operation is controlled by the status register QSTAT which enables and disables various states of the system. QSTAT is constantly displayed on the RTPP left side of the control desk. Usually, QSTAT is '4000' octal.

bit 0

Frame and scale switch enable for up/down counters. 0 to disable, 1 to enable. There is a manual enable overide switch on the RTPP left control desk.

bit 1

Shift register recycle enable. If enabled, then doing a "ADVSR" will recycle the data in the SRG, as will the "ZSRGI". When doing the green pass during classification using the QMT xy compare, recycle should be on so that data is not lost if the points are too close to each other. 0 to disable, 1 to enable recycle.

bit 2

Select SRG QMT data acquisition method: push data into SRG on ACP's (0), or xy comparator (1). This is used to push data into the SRG during the first pass (on any ACP), and to push the density condition on the second pass (on xy real QMT = xy of the front of the SRG). Note the computer control "end/full feature" on the QMT must be on full feature to get ACP's. The "gated count" output of the MS3 standard computer module is used so as to apply the sizing condition to all ACPs. (Sizing set to 1 will get rid of some of the noise.)

bit 3

SRG data "push" enable for the SRG. It is used to enable the shift register ACP/xy compare data to be pushed automatically. It is disabled while reading the the shift register): 0 to disable, 1 to enable. It enables the storage of data in the shift register. Use "ADVSR" to advance the shift register and (RSRGX, RSRGY, RFC1H/L, RFC2H/L) to read the detected data from the shift register. While disabled, the user may single shot the QMT and collect data from the QMT system control (via QDAT1, QDAT2 QDAT3) without affecting the contents of the SRG.

bit 4

Is used to simulate the detector true/false line. It is used primarily during maintenance. 0 indicates not detected, 1 indicates detected. This line should be active only while the QMT/PDP8E SRG control is set to PDP8E control (i.e. QSTAT[5]=1).

bit 5

QMT/PDP8E shift register control selects the QMT SRG control (0) or PDP8E SRG control (1). This allows PDP8E control of the QMT VTRIG (SMVTG), SYNC (SMSYC), CLOCK (SMCLK), ACP (SMACP) (or count out) and detected video (QSTAT[4]). These signals are used to drive the SRG complex including the x,y coordinate generator and coordinate shift register.

bit 6

Standby disable status. If a 1, then the QMT display is put in standby. This is accomplished by use of a relay which completes the comdition of the QMT display standby switch. Therefore the QMT display standby switch must always be in on (up) position. Bit 6 set to 0 is the non-standby status.

bit 7

Frame and scale remote size keys counters enable/disable (0/1). It is useful for moving a fixed size frame around with the control desk keys without disturbing the size.

bit 8

Open the QMT shutter by loading bit 8. Normally, this bit is clear which means that the shutter is open. To close the shutter after it has been opened, clear the bit in QSTAT[8]. A control desk toggle switch is used to overide the computer control with manual intervention from the shutter control electronics.

bit summary for QSTAT
=====================

0	- Frame and scale enable
1	- SRG recycle enable
2	- SRG xy comparator enable (disable ACP)
3	- SRG "push" enable
4	- Simulated detector
5	- Simulated QMT enable
6	- Standby disable
7	- Frame and scale size counter disable.
8	- QMT camera/Axiomat shutter open[0]/close[1]

C.2 PDP8e IOT Instructions for RTPP-PDP8e DMA

Various data transfers must be set up and carried out between the 8e and the GPP and the PDP8e and BM. This section discusses the DMA channel implementation as well as GPP control

DMA from the PDP8e is distributed from one general DMA card which plugs into the "Master" (RTPP) PDP8e. Several commands are implemented for the DMA channel control which enable the selection and specification of DMA activity. As the DMA channel selected by the PDP8e DMAGO[9:11] only one channel can be active at a time! This section describes the DMA channel.

1. DMAGO instruction to start the specified DMA channel. A command word in the AC is loaded by the DMAGO.

bit	function
0	(0) Read device ==>8e
	(1) Write device <==8e
1	(0) Wait for GPP I/O instruction
	(1) Direct I/O (forced by the PDP8e)
3:4	packing mode for BM data
	00 two 8e-words/1 16-bit packed EAE format (word count = Mod 2)
	01 4 8e-words/6 low 8-bit BM bytes (OS8 packed) (word count = Mod 4)
	10 4 8e-words/6 high 8-bit BM bytes (OS8 packed) (word count = Mod 4)
	11 4 8e-words/3 16-bit BM bytes (OS8 packed) (word count = Mod 4)
5	(0) normal operation
	(1) disable DMA (for maintenance)
6:8	PDP8e extended current address.
9:11	DMA channel select.
	000 BM I/O.
	001 X8e I/O.
	010 GPP general (GR) I/O.
	011 GPP program memory (PM) I/O.
	100 GPP microprogram memory (MPM) I/O is allocated (even if not eventually used).
	101 spare
	110 spare
	111 spare

2. DMASKP skip on DMA channel done.

3. DMAWC load the DMA PDP8e word count from the PDP8e AC, i.e. binary number of PDP8e words to be transfered. NOTE: DMAWC = 0000 will transfer 4096 words. See DMAGO[3:4] for BM word count.

4. DMACA load the DMA PDP8e current address from the PDP8e

AC, i.e. address of first transfer.

5. DMACLR clear the DMA channels. Note that a PDP8e CLF (6007) or PDP8e front panel lear also causes a DMACLR.

6. Each DMA peripheral device (BM, GR, etc.) requires an additional address at which to perform the DMA in the peripheral device address space.

EXDMA1 - loads the high 12-bits of I/O device address.
EXDMA2 - loads low 12-bits of I/O device address.

C.3 PDP8e IOT Instructions for BM controller

1. Eight pairs of BM (XB(i),YB(i)) coordinate registers (1:11 bit decimal each).

BMX0	-	load the BM0 X coord. Reg<==8e acc.
BMY0	-	load the BM0 Y coord. Reg<==8e acc.
BMX1	-	load the BM1 X coord. Reg<==8e acc.
BMY1	-	load the BM1 Y coord. Reg<==8e acc.
BMX2	-	load the BM2 X coord. Reg<==8e acc.
BMY2	-	load the BM2 Y coord. Reg<==8e acc.
BMX3	-	load the BM3 X coord. Reg<==8e acc.
BMY3	-	load the BM3 Y coord. Reg<==8e acc.
BMX4	-	load the BM4 X coord. Reg<==8e acc.
BMY4	-	load the BM4 Y coord. Reg<==8e acc.
BMX5	-	load the BM5 X coord. Reg<==8e acc.
BMY5	-	load the BM5 Y coord. Reg<==8e acc.
BMX6	-	load the BM6 X coord. Reg<==8e acc.
BMY6	-	load the BM6 Y coord. Reg<==8e acc.
BMX7	-	load the BM7 X coord. Reg<==8e acc.
BMY7	-	load the BM7 Y coord. Reg<==8e acc.

2. Two commands (GETA, GETB) are required to enable the acquiring of gray scale or detected (binary mask) video data into buffer memories. Issuing a STCMT instruction after one of these commands will cause video scan data to be acquired on the next scan. The BMs are divided into two groups (A and B) each with their own sub-controller. Thus one group can be posting or acquiring data while the other is being used by the GPP or PDP8e to compute in. Group A includes BMs (0,1,2,3) and group B includes (4,5,6,7). A status word is loaded by the PDP8e from the AC to specify these enables.

Bit	Function	BMi
0	1 ==>sel. high byte pix/low byte bin mask	0 (A), 4 (B)
	0 ==>sel. low byte pix/high byte bin mask	0 (A), 4 (B)
1		1 (A), 5 (B)
2		2 (A), 6 (B)
3		3 (A), 7 (B)
4	1 ==>enab. bin mask acquisiton in BM	0 (A), 4 (B)
	0 ==>disable bin mask acquisiton in BM	0 (A), 4 (B)
5		1 (A), 5 (B)
6		2 (A), 6 (B)
7		3 (A), 7 (B)
8	1 ==> select BM	0 (A), 4 (B)
	0 ==> deselect BM	0 (A), 4 (B)
9		1 (A), 5 (B)
10		2 (A), 6 (B)
11		3 (A), 7 (B)

Note that one may acquire the mask byte of a BM while storing the gray scale data in the other byte. The status of the two gets may be tested by reading in the GET-DONE bits for the two groups

RGETA - read done bits for 0,1,2,3 into [0:3].
RGETB - read done bits for 4,5,6,7 into [0:3].

3. Two commands POSTA and POSTB are used to specify which BMs to post on the Quantimet display (at the corresponding coordinate register specified windows). Either the high or low BM byte may be displayed according to bits [0:3]. In addition the BM binary mask may be generated if the corresponding bits are set [4:7]. The BMs to be displayed are selected from bits [8:11]. So the table given for GETA/GETB also applies to POSTA/POSTB.

C.4 PDP8e IOT Instructions for GPP controller

1. GPP STATUS[0:15] is read/write by the PDP8e.

   STATG2[8:11] - read GPP status register bits 0:3.
   STATG1[0:11] - read GPP status register bits 4:15

2. GPPCLR - does a GPP clear to clear GPP registers including all those done by a GPP IOCLR and also clears the GPP PDLCNT, PC and DAB trap enables.

3. GPPCONT - does a GPP continue to turn on the GPP run flip flop.

4. GPPHLT - does a GPP halt to turn off the GPP run flip flop.

4. Load the PC trap register. The trap will halt the GPP if the trap address appears in the GPP PC with the trap found bit set in the GPP status register. The panel switch must be up to enable.

   PCTRP - load EXDMA1:EXDMA2 into the GPP PC trap
   register and enable the trap.
   PCTDS - disable PC trap.

5. Load the GPP PC register.

   GPPLAD - load EXDMA1:EXDMA2 into the GPP PC register.
   RGPPCH[8:11] - read GPP PC high
   RGPPCL[0:11] - read GPP PC low.

6. Load the DAB trap register. The trap will halt the GPP if the trap address appears in the GPP DAB with the trap found bit set in the GPP status register. The panel switch must be up to enable.

   DABTRP - load EXDMA1:EXDMA2 into the GPP DAB trap
   register and enable the trap.
   DABTDS - disable DAB trap.

C.5 PDP8e IOT Instructions for X8E controller

In controlling the X8E auxillary PDP8e processor, two output words are used by the X8e Controller.

1. X8ECTL - load the X8e control word from the PDP8e AC. When a bit is on in any one of these control functions the function will be executed. There is no need to clear the bit.

Bits	function
0	Halt X8e
1	Clear X8e
2	Cont X8e
3	Addr load X8e
4	Extd addr load X8e
5	Dep X8e
9-11	Extended current address for X8e.

2. X8ECA - X8e current address and switch register. Used for X8e current address on DMA and the X8e switch register when the X8ECTL instruction is used.

C.6 PDP8e IOT Instructions for PDP8e core line buffer controller

The triple line buffer may be simulated in PDP8e core until the GPP is built using a special core line buffer controler interface. The interface works as follows: various control registers are setup to specify mode, BM number, high or low byte, read or write, 3x3 neighborhood or length n (n=1 to 512) pixel line, horizontal or vertial line. The x and y pixel addresses are specified as PDP8e addresses. The controller then gets the current (x,y) values from PDP8e core when I/O is requested.

The controller uses a microprocessor (uP) (loaded with a control program previous to use) to control FIFO buffers between the PDP8e and the uP and the buffer memory controller and the uP. The controller is illustrated in the following figure.

```
PDP8e Omnibus          Microprocessor           Buffer Memory
                             ||
                 -------     ||    -------
DMA 15-bit  <----| FIFO |<--||--> | FIFO |---->BM 19-bit addr.
 Address         -------     ||    -------
                             ||
                 -------     ||    -------
DMA 12-bit  <----| FIFO |<--||--> | FIFO |---->BM 16-bit data
 Read            -------     ||    -------
                             ||
                 -------     ||    -------
DMA 12-bit  ---->| FIFO |-->||<--| FIFO |<----BM 16-bit data
 Write           -------     ||    -------
                             ||
                 -------     ||    -------
PDP8e 12-bit--->| REGs |-->||--> | FIFO |----->BM 16-bit I/O mode
 Write           -------     ||    -------          control
                             |
                 -------     |
PDP8e 12-bit<---| REGs |<--|
 Read status     -------     |
  and uP mem.                |
                             |
                 -------     |
PDP8e 12-bit--->| REGs |-->|
 Load uP mem.    -------     |
    and control
```

The following IOTS control the interface.

1. LBSZR - load the line buffer size register from AC[1:11]. This is used with line I/O only.

2. LBBAR (LBBXAR) - load the core line buffer address low 12 bits (high 3 bits).

3. LBXPR - load x variable pointer address (low 12-bits)

4. LBYPR - load y variable pointer address (low 12-bits)

5. LBEXPR - load extended x,y pointer address registers x[6:8], y[9:11].

6. LBCTLR - load the BM# and byte control register as follows:

bit	function
0	read(0), write(1)
3	horiz line (0), vertical line(1)
4	3X3 neighborhood(0), line(1)
5	low byte (0), high byte (1)
9:11	buffer memory #

7. LBGO - start the transfer.

8. LBSKP - skip if transfer finished.

C.6.1 Microprocessor IOTs

Several IOTs are used to load and read the uP memory as well as start and test it.

1. LUADRL (LUADRH) - load the uP address register.

2. LUMEML (LUMEMH) - load the uP memory at LUADRL:H.

3. LUCR - load 12-bit uP control register.

4. RUMEML (RUMEMH) - read the uP memory at LUADRL:H.

5. RUCR - read 12-bit uP control register.

C.7 Allocation of PDP8e IOTs for the RTPP

The following lists are the PDP8e device codes allocated for the QMT, control desk, GPP, BM, parts of the RTPP.

C.7.1 Alphabetic listing of PDP8e IOTs

DEVICE	CODE	CARD	FUNCTION
ADVSR	6314	RQCA10	ADVANCE SHIFT REG. ONE
BMIN	6542	BMD25	BM maintenance (spare input)
BMOUT	6526	BMD25	BM maintenance (spare output)
BMX0	6500	BMD29	load the BM0 X coord reg<==8e Acc.
BMX1	6501	BMD29	load the BM1 X coord reg<==8e Acc.
BMX2	6502	BMD29	load the BM2 X coord reg<==8e Acc.
BMX3	6503	BMD29	load the BM3 X coord reg<==8e Acc.
BMX4	6510	BMD27	load the BM4 X coord reg<==8e Acc.
BMX5	6511	BMD27	load the BM5 X coord reg<==8e Acc.
BMX6	6512	BMD27	load the BM6 X coord reg<==8e Acc.
BMX7	6513	BMD27	load the BM7 X coord reg<==8e Acc.
BMY0	6504	BMD29	load the BM0 Y coord reg<==8e Acc.
BMY1	6505	BMD29	load the BM1 Y coord reg<==8e Acc.
BMY2	6506	BMD29	load the BM2 Y coord reg<==8e Acc.
BMY3	6507	BMD29	load the BM3 Y coord reg<==8e Acc.
BMY4	6514	BMD27	load the BM4 Y coord reg<==8e Acc.
BMY5	6515	BMD27	load the BM5 Y coord reg<==8e Acc.
BMY6	6516	BMD27	load the BM6 Y coord reg<==8e Acc.
BMY7	6517	BMD27	load the BM7 Y coord reg<==8e Acc.
CLKACK	6302	RQCA4	CLEAR 200 HZ CLOCK FLAG
CLKSKP	6303	RQCA4	SKIP ON 200 HZ CLOCK FLAG SET
CSRGI	6315	RQCA10	CLEAR SHIFT REG. INDEX REG.
DABTDS	-	-	disable GPP DAB trap.
DABTRP	-	-	load EXDMA1:EXDMA2 into the GPP DAB trap
DET1	6422	RQCA8	1-D DETECTOR THRESHOLD 1 <== C(AC) (*NOT USED)
DET2	6423	RQCA8	1-D DETECTOR THRESHOLD 2 <== C(AC) (*NOT USED)
DETDIG	6424	RQCA8	DENSITOMETER 6-bit T1 (LEFT BYTE) AND T2 (RIGHT BYTE) THRESH <== C(AC)
DETB	6420	RQCA8	DETECTOR THRESHOLD B <== C(AC)
DETC	6421	RQCA8	DETECTOR THRESHOLD C <== C(AC)
DISP1	6435	ROCA22	CONTROL DESK DISPLAY 1 <== C(AC)
DISP2	6436	RQCA22	CONTROL DESK DISPLAY 2 <== C(AC)
DMACA	6073	RDMA	load RTPP DMA current address register
DMACLR	6074	RDMA,BMB31	clear the RTPP DMA interface
DMAGO	6070	RDMA,BMB31	start the RTPP DMA PDP8e<==>DMA
DMASKP	6071	RDMA	skip on RTPP DMA done
DMAWC	6072	RDMA	load the DMA word count register
EXADR	6450	RQCB/EXIN/EXOUT	C(AC)==>C(XADR)
EXDMA1	6524	EBUS2	load high RTPP DMA address bus (also BMD31,BMD25,BMB31)
EXDMA2	6525	EBUS2	load low RTPP DMA address bus (also BMD31,BMD25,BMB31)

EXIN	6333	RQCB/EXIN	C(AC)<==C(C(XADR))
EXOUT	6451	RQCB/EXOUT	C(AC)==>C(C(XADR))
FBW1	6341	RQCB2	C(AC) <== CONTROL DESK DATA WORD 1
FBW2	6342	RQCB2	C(AC) <== CONTROL DESK DATA WORD 2
FBW3	6343	RQCB2	C(AC) <== CONTROL DESK DATA WORD 3
FBW4	6344	RQCB2	C(AC) <== CONTROL DESK DATA WORD 4
FBW5	6345	RQCB2	C(AC) <== CONTROL DESK DATA WORD 5
FBW6	6346	RQCB2	C(AC) <== CONTROL DESK DATA WORD 6
FBW7	6347	RQCB2	C(AC) <== CONTROL DESK DATA WORD 7
FBW10	6350	RQCB4	C(AC) <== CONTROL DESK DATA WORD 10
FBW11	6351	RQCB4	C(AC) <== CONTROL DESK DATA WORD 11
FBW12	6352	RQCB4	C(AC) <== CONTROL DESK DATA WORD 12
GALSKP	6400	-----	skip when the galvanometer scanner is ready.
GETA	6522	BMD27	enable acquiring group A BM pix or binary masks
GETB	6523	-	enable acquiring group B BM pix or binary masks
GETMSK	6304	RQCA14	LOAD MASK REG FROM QMT ON NEXT STQMT
GPPCLR	-	-	clear the GPP registers
GPPCONT	-	-	continue the GPP
GPPHLT	-	-	HLT the GPP
GPPLAD	-	-	load GPP PC from EXDMA1,2
HPL	6360	RQCA6	FRAME HOR. POSITION COUNTER <== C(AC)
HPR	6320	RQCA6	C(AC) <== FRAME HOR. POSITION COUNTER
HSL	6361	RQCA6	FRAME HOR. SIZE COUNTER <== C(AC)
HSR	6321	RQCA6	C(AC) <== FRAME HOR. SIZE COUNTER
IZSKP	6317	RQCA10	SKIP IF INDEX 10-bits = ZERO.
LBBAR	6601		LOAD CORE LINE BUFFER ADDRESS LOW 12-BITS
LBBXAR	6602		LOAD CORE LINE BUFFER ADDRESS HIGH 3-BITS
LBCTLR	6606		LOAD CORE LINE BUFFER CONTROL REGISTER
LBEXPR	6605		LOAD LINE CORE BUFF. X,Y VAR. PTR EXTD ADDR.
LBGO	6620		START CORE LINE BUFFER TRANSFER
LBSKP	6621	SKIP WHEN LINE CORE BUFFER XFER DONE.	
LBSZR	6600	LOAD LINE BUFFER SIZE REGISTER	
LBXPR	6603		LOAD X LINE CORE BUFF VAR PTR REG.
LBYPR	6604		LOAD Y LINE CORE BUFF VAR PTR REG.
LDXP	6443	RQCA14	X COORDINATE CURSOR REG. <== C(AC)
LDYP	6444	RQCA14	Y COORDINATE CURSOR REG. <== C(AC)
LFBW2	6437	RQCA22	CONTROL DESK SWITCH LIGHTS <== C(AC)
LGALX	6456	RQCB8	MIRROR SCANNER X COORDINATE <== C(AC)
LGALY	6457	RQCB8	MIRROR SCANNER Y COORDINATE <== C(AC)
LMASKE	6441	RQCA14	MASK ENTRANCE REG. <== C(AC)
LMASKX	6442	RQCA14	MASK EXIT REG. <== C(AC)
LPENX	6445	RQCA16	LIGHT PEN X COORDINATE <== C(AC)
LPENY	6446	RQCA16	LIGHT PEN Y COORDINATE <== C(AC)
LQDT1	6375	RQCA4	QMT RIGHT DISPLAY LSW <== C(AC)
LQDT2	6376	RQCA4	QMT RIGHT DISPLAY MIDDLE WORD <== C(AC)
LQDT3	6377	RQCA4	QMT RIGHT DISPLAY MSW <== C(AC)
LSRGB	6367	RQCA10	SHIFT REG. LOADING REG. <== C(AC)
6607	LUADRL		LOAD uP ADDRESS REGISTER LOW 12-BITS
6610	LUADRH		LOAD uP ADDRESS REGISTER HIGH 12-BITS
6613	LUCR		LOAD uP STATUS REGISTER 12-BITS
6611	LUMEML		LOAD uP MEMORY LOW 12-BITS
6612	LUMEMH		LOAD uP MEMORY HIGH 12-BITS
MSKADR	6440	RQCA14	MASK ADDRESS REG. <== C(AC)
MSTAG	6366	RQCB10	STAGE DIRECTION REG. <== C(AC)

PCTDS	-	-	disable the PC address trap
PCTRP	-	-	load EXDMA1:EXDMA2 into the GPP PC trap
PENST	6447	RQCA16	LIGHT PEN STATUS REG. <== C(AC)
POSTA	6520	BMD29	post selected group A BMs
POSTB	6521	BMD27	post selected group B BMs
QDAT1	6324	RQCA4	C(AC) <== QMT BCD DATA LSW
QDAT2	6325	RQCA4	C(AC) <== QMT BCD DATA MIDDLE WORD
QDAT3	6326	RQCA4	C(AC) <== QMT BCD DATA MSW
QMSKP	6301	RQCA4	SKIP WHEN QMT DATA SCAN DONE
QPROG1	6370	RQCA4	QMT PROGRAM WORD 1 <== C(AC)
QPROG2	6371	RQCA4	QMT PROGRAM WORD 2 <== C(AC)
QPROG3	6372	RQCA4	QMT PROGRAM WORD 3 <== C(AC)
QPROG4	6373	RQCA4	QMT PROGRAM WORD 4 <== C(AC)
QPROG5	6431	RQCA22	QMT PROGRAM WORD 5 <== C(AC)
QPROG6	6432	RQCA22	QMT PROGRAM WORD 6 <== C(AC)
QPROG7	6433	RQCA22	QMT PROGRAM WORD 7 <== C(AC)
QPROG8	6434	RQCA22	QMT PROGRAM WORD 8 <== C(AC)
QSTAT	6374	RQCA4	RQC PROGRAM WORD <== C(AC)
RBMSP1	6543	-	spare input (not implemented)
RBMSP2	6544	-	spare input (not implemented)
RBMSP3	6545	-	spare input (not implemented)
RBMSP4	6546	-	spare input (not implemented)
RBMSP5	6547	-	spare input (not implemented)
RFC1H	6334	RQCA12	C(AC) <== FUNCTION COMPUTER 1 MSW
RFC1L	6335	RQCA12	C(AC) <== FUNCTION COMPUTER 1 LSW
RFC2H	6336	RQCA12	C(AC) <== FUNCTION COMPUTER 2 MSW
RFC2L	6337	RQCA12	C(AC) <== FUNCTION COMPUTER 2 LSW
RGAL	6410	------	C(AC) <== integrated galvanometer scanner data.
RGETA	6540	BMD29	Read the status of done bits for BM GETA
RGETB	6541	BMD27	Read the status of done bits for BM GETB
RGPPCH	-	-	read GPP PC high
RGPPCL	-	-	read GPP PC low
RKYPDH	6340	RQCB6	C(AC) <== CONTROL DESK KEY PAD MSW
RKYPDL	6353	RQCB6	C(AC) <== CONTROL DESK KEY PAD LSW
RMASKE	6354	RQCA14	C(AC) <== MASK ENTRANCE DATA AS F(MSKADR)
RMASKX	6355	RQCA14	C(AC) <== MASK EXIT DATA AS F(MSKADR)
RPENX	6356	RQCA16	C(AC) <== LIGHT PEN X COORDINATE
RPENY	6357	RQCA16	C(AC) <== LIGHT PEN Y COORDINATE
RQSTAT	6327	RQCA4	C(AC) <== QSTAT
RUCR	6672		READ uP STATUS 12-BITS
RUMEML	6670		READ uP MEMORY LOW 12-BITS
RUMEMH	6671		READ uP MEMORY HIGH 12-BITS
RSRGI	6332	RQCA10	C(AC) <== SHIFT REG. INDEX REG.
RSRGX	6330	RQCA10	C(AC) <== X CORDINATE SHIFT REG. DATA
RSRGY	6331	RQCA10	C(AC) <== Y CORDINATE SHIFT REG. DATA
SIZEA	6425	RQCA8	AMENDER SIZE REG. <== C(AC) BCD
SIZEC	6426	RQCA8	CLASS/COLLEC SIZE REG. <== C(AC) BCD
SIZEM	6427	RQCA8	MS3 COMPUTER SIZE REG. <== C(AC) BCD
SIZES	6430	RQCA22	STD COMP. SIZE REG. <== C(AC) BCD
SKPKPD	6313	RQCB6	SKIP ON CONTROL DESK KEYPAD, CLEAR FLAG ON SKIP
SMACP	6310	RQCA10	SIMULATE QMT ACP AS F(QSTAT BIT 5)
SMCLK	6311	RQCA10	SIMULATE QMT CLOCK AS F(QSTAT BIT 5)
SMHLD	6307	RQCA4	SIMULATE QMT HOLD AS F(QSTAT BIT 5)

SMOTR	6365	RQCB10	SPARE MOTOR REGISTER<==C(AC)
SMSYN	6312	RQCA10	SIMULATE QMT SYNC AS F(QSTAT BIT 5)
SMVTG	6306	RQCA4	SIMULATE QMT VERT. TRIG AS F(QSTAT BIT 5)
STEP	6305	RQCB10	MOVE STAGE AS F(MSTAG)
STQMT	6300	RQCA4	START QMT DATA SCAN
VPL	6362	RQCA6	FRAME VERT. POSITION COUNTER <== C(AC)
VPR	6322	RQCA6	C(AC) <== FRAME VERT. POSITION COUNTER
VSL	6363	RQCA6	FRAME VERT. SIZE COUNTER <== C(AC)
VSR	6323	RQCA6	C(AC) <== FRAME VERT. SIZE COUNTER
X8ECA	-	-	LOAD X8E CURRENT ADDRESS
X8ECTL	-	-	LOAD X8E CONTROL REGISTER
ZSRGI	6316	RQCA10	SEND SHIFT REG. DATA TO FRONT OF SHIFT REG.

C.8 PDP8e Device code allocation by decade

The actual device codes for particular DEC devices may be found in the various versions of the "Small Computer Handbook".

00	- PDP8e CPU
01	- paper tape reader
02	- paper tape punch
03	- Decwriter keyboard (or PDP11/20 emulator)
04	- Decwriter printer (or PDP11/20 emulator)
07	- RTPP DMA I/O channel
10	- Dicomed
11;12	- free
12	- DC02 serial interface printer
14	- Graf-Pen
16	- HSP: input channel (PDP11/20 emulator)
17	- HSP: output channel (PDP11/20 emulator)
20:27	- PDP8e extended memory control
30:37	- RQC - EBUS #1
40:45	- RQC - EBUS #1
50;52	- EBUS #2
53	- PDP8e A/D multiplexor AD8e/AM8e
54	- EBUS #2
55:56	- PDP8e Floating point processor FPP
57	- EBUS #2
60:62	- EBUS #2
63	- (Decwriter KBD - not used since PTR used as LPT)
64:65	- EBUS #2
66	- PDP8e line printer LP08 (Decwriter PTR) or PDP11/20 emulator
67	- EBUS #2
70:72	- PDP8e TC58 magtape control
74	- PDP8e RK8e disk control
76:77	- PDP8e TC08 Dectape control

C.8.1 Numerical listing of PDP8e IOTs

RDMA "OUTPUT" DEVICE CODES - C(PDP8e ACC)<==C(channel)

DEVICE CODE		CARD	FUNCTION
6073	DMACA	RDMA	load RTPP DMA current address register
6074	DMACLR	RDMA,BMB31	clear the RTPP DMA interface
6070	DMAGO	RDMA,BMB31	start the RTPP DMA PDP8e<=>DMA
6071	DMASKP	RDMA	skip on RTPP DMA done
6072	DMAWC	RDMA	load the DMA word count register

PDP8e auxillary devices

DEVICE CODE		CARD	FUNCTION (on DEC 1709 card)
6101	DICSKP	PFL1	skip on Dicomed ready for next command
6102	DICLR	PFL1	set Dicomed ready
6106	DICO	PFL1	send Dicomed command <==AC[0,3:11]

RQC "PULSE" DEVICE CODES - note: does not affect the PDP8e ACC

DEVICE CODE		CARD	FUNCTION
6300	STQMT	RQCA4	START QMT DATA SCAN
6301	QMSKP	RQCA4	SKIP WHEN QMT DATA SCAN DONE
6302	CLKACK	RQCA4	CLEAR 200 HZ CLOCK FLAG
6303	CLKSKP	RQCA4	SKIP ON 200 HZ CLOCK FLAG SET
6304	GETMSK	RQCA14	LOAD MASK REG FROM QMT ON NEXT STQMT
6305	STEP	ROCB10	MOVE STAGE AS F(MSTAG)
6306	SMVTG	RQCA4	SIMULATE QMT VERT. TRIG AS F(QSTAT BIT 5)
6307	SMHLD	RQCA4	SIMULATE QMT HOLD AS F(QSTAT BIT 5)
6310	SMACP	RQCA10	SIMULATE QMT ACP AS F(QSTAT BIT 5)
6311	SMCLK	RQCA10	SIMULATE QMT CLOCK AS F(QSTAT BIT 5)
6312	SMSYN	RQCA10	SIMULATE QMT SYNC AS F(QSTAT BIT 5)
6313	SKPKPD	RQCB6	SKIP ON CONTROL DESK KEYPAD, CLEAR FLAG ON SKIP
6314	ADVSR	RQCA10	ADVANCE SHIFT REG. ONE
6315	CSRGI	RQCA10	CLEAR SHIFT REG. INDEX REG.
6316	ZSRGI	RQCA10	SEND SHIFT REG. DATA TO FRONT OF SHIFT REG.
6317	IZSKP	RQCA10	SKIP IF INDEX 10-bits = ZERO.
6400	GALSKP	-----	skip when the galvanometer scanner is ready.
6401			
6402			
6403			
6404			
6405			
6406			
6407			

RQC "INPUT" DEVICE CODES - C(channel)==>C(PDP8e ACC)

DEVICE CODE		CARD	FUNCTION
6320	HPR	RQCA6	C(AC) <== FRAME HOR. POSITION COUNTER
6321	HSR	RQCA6	C(AC) <== FRAME HOR. SIZE COUNTER
6322	VPR	RQCA6	C(AC) <== FRAME VERT. POSITION COUNTER
6323	VSR	RQCA6	C(AC) <== FRAME VERT. SIZE COUNTER
6324	QDAT1	RQCA4	C(AC) <== QMT BCD DATA LSW
6325	QDAT2	RQCA4	C(AC) <== QMT BCD DATA MIDDLE WORD
6326	QDAT3	RQCA4	C(AC) <== QMT BCD DATA MSW
6327	RQSTAT	RQCA4	C(AC) <== QSTAT
6330	RSRGX	RQCA10	C(AC) <== X CORDINATE SHIFT REG. DATA
6331	RSRGY	RQCA10	C(AC) <== Y CORDINATE SHIFT REG. DATA
6332	RSRGI	RQCA10	C(AC) <== SHIFT REG. INDEX REG.
6333	EXIN	RQCB/EXIN	C(C(XADR)==>C(AC)
6334	RFC1H	RQCA12	C(AC) <== FUNCTION COMPUTER 1 MSW
6335	RFC1L	RQCA12	C(AC) <== FUNCTION COMPUTER 1 LSW
6336	RFC2H	RQCA12	C(AC) <== FUNCTION COMPUTER 2 MSW
6337	RFC2L	RQCA12	C(AC) <== FUNCTION COMPUTER 2 LSW
6340	RKYPDH	RQCB6	C(AC) <== CONTROL DESK KEY PAD MSW
6341	FBW1	RQCB2	C(AC) <== CONTROL DESK DATA WORD 1
6342	FBW2	RQCB2	C(AC) <== CONTROL DESK DATA WORD 2
6343	FBW3	RQCB2	C(AC) <== CONTROL DESK DATA WORD 3
6344	FBW4	RQCB2	C(AC) <== CONTROL DESK DATA WORD 4
6345	FBW5	RQCB2	C(AC) <== CONTROL DESK DATA WORD 5
6346	FBW6	RQCB2	C(AC) <== CONTROL DESK DATA WORD 6
6347	FBW7	RQCB2	C(AC) <== CONTROL DESK DATA WORD 7
6350	FBW10	RQCB4	C(AC) <== CONTROL DESK DATA WORD 10
6351	FBW11	RQCB4	C(AC) <== CONTROL DESK DATA WORD 11
6352	FBW12	RQCB4	C(AC) <== CONTROL DESK DATA WORD 12
6353	RKYPDL	RQCB6	C(AC) <== CONTROL DESK KEY PAD LSW
6354	RMASKE	RQCA14	C(AC) <== MASK ENTRANCE DATA AS F(MSKADR)
6355	RMASKX	RQCA14	C(AC) <== MASK EXIT DATA AS F(MSKADR)
6356	RPENX	RQCA16	C(AC) <== LIGHT PEN X COORDINATE
6357	RPENY	RQCA16	C(AC) <== LIGHT PEN Y COORDINATE
6410	RGAL	------	C(AC) <== integrated galvanometer scanner data.
6411			
6412			
6413			
6414			
6415			
6416			
6417			

RQC "OUTPUT" DEVICE CODES - C(PDP8e ACC)<==C(channel)

DEVICE CODE		CARD	FUNCTION
6360	HPL	RQCA6	FRAME HOR. POSITION COUNTER <== C(AC)

6361	HSL	RQCA6	FRAME HOR. SIZE COUNTER <== C(AC)
6362	VPL	RQCA6	FRAME VERT. POSITION COUNTER <== C(AC)
6363	VSL	RQCA6	FRAME VERT. SIZE COUNTER <== C(AC)
6364			
6365	SMOTR	RQCB10	SPARE MOTOR REGISTER<==C(AC)
6366	MSTAG	RQCB10	STAGE DIRECTION REG. <== C(AC)
6367	LSRGB	RQCA10	SHIFT REG. LOADING REG. <== C(AC)
6370	QPROG1	RQCA4	QMT PROGRAM WORD 1 <== C(AC)
6371	QPROG2	RQCA4	QMT PROGRAM WORD 2 <== C(AC)
6372	QPROG3	RQCA4	QMT PROGRAM WORD 3 <== C(AC)
6373	QPROG4	RQCA4	QMT PROGRAM WORD 4 <== C(AC)
6374	QSTAT	RQCA4	ROC PROGRAM WORD <== C(AC)
6375	LQDT1	RQCA4	QMT RIGHT DISPLAY LSW <== C(AC)
6376	LQDT2	RQCA4	QMT RIGHT DISPLAY MIDDLE WORD <== C(AC)
6377	LQDT3	RQCA4	QMT RIGHT DISPLAY MSW <== C(AC)
6420	DETB	RQCA8	DETECTOR THRESHOLD B <== C(AC)
6421	DETC	RQCA8	DETECTOR THRESHOLD C <== C(AC)
6422	DET1	RQCA8	1-D DETECTOR THRESHOLD 1 <== C(AC) (*NOT USED)
6423	DET2	RQCA8	1-D DETECTOR THRESHOLD 2 <== C(AC) (*NOT USED)
6424	DETDIG	RQCA8	DENSITOMETER 6-bit T1 AND T2 THRESHOLDS <== C(AC)
6425	SIZEA	RQCA8	AMENDER SIZE REG. <== C(AC)
6426	SIZEC	RQCA8	CLASSIFIER COLLECTOR SIZE REG. <== C(AC)
6427	SIZEM	RQCA8	MS3 COMPUTER SIZE REG. <== C(AC)
6430	SIZES	RQCA22	STANDARD COMPUTER SIZE REG. <== C(AC)
6431	QPROG5	RQCA22	QMT PROGRAM WORD 5 <== C(AC)
6432	QPROG6	RQCA22	QMT PROGRAM WORD 6 <== C(AC)
6433	QPROG7	RQCA22	QMT PROGRAM WORD 7 <== C(AC)
6434	QPROG8	RQCA22	QMT PROGRAM WORD 8 <== C(AC)
6435	DISP1	RQCA22	CONTROL DESK DISPLAY 1 <== C(AC)
6436	DISP2	RQCA22	CONTROL DESK DISPLAY 2 <== C(AC)
6437	LFBW2	RQCA22	CONTROL DESK SWITCH LIGHTS <== C(AC)
6440	MSKADR	RQCA14	MASK ADDRESS REG. <== C(AC)
6441	LMASKE	RQCA14	MASK ENTRANCE REG. <== C(AC)
6442	LMASKX	RQCA14	MASK EXIT REG. <== C(AC)
6443	LDXP	RQCA14	X COORDINATE CURSOR REG. <== C(AC)
6444	LDYP	RQCA14	Y COORDINATE CURSOR REG. <== C(AC)
6445	LPENX	RQCA16	LIGHT PEN X COORDINATE <== C(AC)
6446	LPENY	RQCA16	LIGHT PEN Y COORDINATE <== C(AC)
6447	PENST	RQCA16	LIGHT PEN STATUS REG. <== C(AC)
6450	EXADR	RQCB/EXIN/EXOUT	C(AC)==>C(XADR)
6451	EXOUT	RQCB/EXOUT	C(AC)==>C(C(XADR))
6452			
6453			
6455			
6456	LGALX	RQCB8	MIRROR SCANNER X COORDINATE <== C(AC)
6457	LGALY	RQCB8	MIRROR SCANNER Y COORDINATE <== C(AC)

BM "OUTPUT" DEVICE CODES - C(PDP8e ACC)<==C(channel)

DEVICE	CODE	CARD	FUNCTION
6500	BMX0	-	load BM0 X coord reg<==8e ACC, GROUP A
6501	BMX1	-	load BM1 X coord reg<==8e ACC, GROUP A
6502	BMX2	BMD29	load BM2 X coord reg<==8e ACC, GROUP A
6503	BMX3	BMD29	load BM3 X coord reg<==8e ACC, GROUP A
6504	BMX0	BMD29	load BM0 Y coord reg<==8e ACC, GROUP A
6505	BMY1	BMD29	load BM1 Y coord reg<==8e ACC, GROUP A
6506	BMY2	BMD29	load BM2 Y coord reg<==8e ACC, GROUP A
6507	BMY3	BMD29	load BM3 Y coord reg<==8e ACC, GROUP A
6510	BMX4	BMD27	load BM4 X coord reg<==8e ACC, GROUP B
6511	BMX5	BMD27	load BM5 X coord reg<==8e ACC, GROUP B
6512	BMX6	BMD27	load BM6 X coord reg<==8e ACC, GROUP B
6513	BMX7	BMD27	load BM7 X coord reg<==8e ACC, GROUP B
6514	BMY4	BMD27	load BM4 Y coord reg<==8e ACC, GROUP B
6515	BMY5	BMD27	load BM5 Y coord reg<==8e ACC, GROUP B
6516	BMY6	BMD27	load BM6 Y coord reg<==8e ACC, GROUP B
6517	BMY7	BMD27	load BM7 Y coord reg<==8e ACC, GROUP B
6520	POSTA	BMD29	post selected group A BMs
6521	POSTB	BMD27	post selected group B BMs
6522	GETA	BMD29	enable acquiring group A BM pix or binary masks
6523	GETB	BMD27	enable acquiring group B BM pix or binary masks
6524	EXDMA1	EBUS2 (also BMD31, BMD25, BMB31)	load high RTPP DMA address bus
6525	EXDMA2	EBUS2 (also BMD31, BMD25, BMB31)	load low RTPP DMA address bus
6526	BMOUT	BMD25	BM maintenance output (spare)

BM "INPUT" DEVICE CODES - C(channel)==>C(PDP8e ACC)

DEVICE	CODE	CARD	FUNCTION
6540	RGETA	BMD29	Read the status of done bits for BM GETA
6541	RGETB	BMD27	Read the status of done bits for BM GETB
6542	BMIN	BMD25	BM maintenance input (spare)
6543	RBMSP1	-	spare input (not implemented)
6544	RBMSP2	-	spare input (not implemented)
6545	RBMSP3	-	spare input (not implemented)
6546	RBMSP4	-	spare input (not implemented)
6547	RBMSP5	-	spare input (not implemented)

LINE BUFFER "OUTPUT" DEVICE CODES - C(channel)<==C(PDP8e ACC)

DEVICE	CODE	CARD	FUNCTION
6600	LBSZR		LOAD LINE BUFFER SIZE REGISTER
6601	LBBAR		LOAD CORE LINE BUFFER ADDRESS LOW 12-BITS
6602	LBBXAR		LOAD CORE LINE BUFFER ADDRESS HIGH 3-BITS
6603	LBXPR		LOAD X LINE CORE BUFF VAR PTR REG.
6604	LBYPR		LOAD Y LINE CORE BUFF VAR PTR REG.
6605	LBEXPR		LOAD LINE CORE BUFF. X,Y VAR. PTR EXTD ADDR.
6606	LBCTLR		LOAD CORE LINE BUFFER CONTROL REGISTER
6607	LUADRL		LOAD uP ADDRESS REGISTER LOW 12-BITS

```
6610     LUADRH           LOAD uP ADDRESS REGISTER HIGH 12-BITS
6611     LUMEML           LOAD uP MEMORY LOW 12-BITS
6612     LUMEMH           LOAD uP MEMORY HIGH 12-BITS
6613     LUCR             LOAD uP STATUS REGISTER 12-BITS

LINE BUFFER "PULSE" DEVICE CODES
--------------------------------
DEVICE CODE      CARD     FUNCTION
***********      ****     ********
6620     LBGO             START CORE LINE BUFFER TRANSFER
6621     LBSKP    SKIP WHEN LINE CORE BUFFER XFER DONE.

LINE BUFFER "INPUT" DEVICE CODES - C(channel)==>C(PDP8e ACC)
------------------------------------------------------------
DEVICE CODE      CARD     FUNCTION
***********      ****     ********
6670     RUMEML           READ uP MEMORY LOW 12-BITS
6671     RUMEMH           READ uP MEMORY HIGH 12-BITS
6672     RUCR             READ uP STATUS 12-BITS
```

APPENDIX D

Examples of programming the GPP

Several examples of programming the GPP are given here in order to impart the flavor of the type of assembly language code which would typically be used on the GPP. Note that the MAINSAIL compiler would produce GPP assembly souce code to be assembled by the PDP8e using GPPASM.

A "communications" assembly language is used in the following examples. It mimics the GPPASM assembly language but is easier to read since the code is in infix notation rather than prefix. That is sink and source operands are denoted here by being on the left and right side of an arrow "<==".

Let comments be denoted by text enclosed in quotes "...". Let an instruction consist of 4 fields: the P3 field followed by "<==", followed by the P1 field, followed by the opr code, followed by the P2 field. Non-existant fields may be ignored. Thus,

```
        <P3> <== <P1> <OPR> <P2>;
or
        <P3> <== <OPR> <P1>;
```

D.1 Gradient used in Kirsch algorithm

The following gradient algorithm is given by Kirsch [Kir69]. Given a 3x3 I1 subarray, the following GPP program will compute the 2-D gradient by the formula "MAX [(3*SUM a(i - (5*SUM b(i)]". This PM program is written by repeating the code instead of using loops. It could be written with loops instead. The 8 permutations of the 8 neighbors are:

```
a1 a2 a3          b5 a1 a2            a2 a3 b1
b5 .  b1          b4 .  a3            a1 .  b2
b4 b3 b2          b3 b2 b1            b5 b4 b3
--------          ---------           ---------
Permutation 1     permutation 2 ...   Permutation 8
```

The following GPP program will compute the gradient at the current pixel. The neighborhood indexing scheme is restated for clarity.

```
                3 2 1
                4 8 0
                5 6 7
```

```
let P1 be (3*SUM a(i)-5*SUM b[i]);
    R2 be MAX variable;
```

R10 to R17 the local permutation values;

The computation for permutation 1 is given and the other 7 permutations are similar.

```
Step    Instruction
----    -----------
[1]     ALUA  <== I1(0) ADD I1(1)
[2]     R1  <== ADDSB I1(2)
[3]     R1  <== #3 MUL R1
[4]     ALUA  <== I1(3) ADD I1(4)
[5]     ALUA  <== ADDSB I1(5)
[6]     ALUA  <== ADDSB I1(6)
[7]     ALUA  <== ADDSB I1(7)
[8]     DRB  <== MULST #5
[9]     R10  <== R1 SUB DRB
        .
        .
        .
```

These 9 steps give the value for 1 permutation. The 8 permutations take 72 steps. The code for the 8 permutations is just concatinated. Although it is not elegant, repeated code is obviously faster. We are able to avoid loops and increase processing speed at the expense of program memory.

To compute the maximum in R2 the following algorithm is used. Again, the PM implementation is repeated code.

```
max<==0;
offset<==9;
FOR i<=1 TO 8
  DO IF R(i+offset).GT.max
        THEN max <==R(i+offset);
```

```
Step    Instruction
----    -----------
[1]     ALUA  <== MOVE R10
[2]     ALUA  <== MAXSB R10
[3]     ALUA  <== MAXSB R12
[4]     ALUA  <== MAXSB R13
[5]     ALUA  <== MAXSB R14
[6]     ALUA  <== MAXSB R15
[7]     ALUA  <== MAXSB R16
[8]     ALUA  <== MAXSB R17
[9]     R2  <== MOVE ALUA
[10]    Haltpoint.
```

Therefore, it takes 72+10=82 instructions/pp to compute the gradient related function. At approximately 300 nsec/GPP instruction, it would take about 30 usec/pp or 256 X 256 x 30usec. or 1.9 seconds to do the entire picture exclusive of I/O. A 128 X 128 picture would take on the order of .3 seconds.

D.2 Eight neighbor filter processing - example

This is an example of 8 neighbor direction list processing. Since all 8+ (9) neighbors are directly addressable, various functions may be computed by iterating in a FOR loop construction using a filter in the general purpose registers R(i).

For example a simple angle finder might consist of the following algorithm:

```
I3(8)<==0;
For i<==0 Step 1 Until 8 Do
  If [SUM a(i)*R(i)] > threshold
        Then I3(8)<==1
        Else I3(8)<==0;
```

This will give a first order approximation to a 135 degree line finder.

A neighborhood filter R(i) might look like

```
+1 -1 -2
-1 +1 -1
-2 -1 +1.
```

Let R1 through R9 be the filter in the GR using
        The same addressing scheme as the current
        Neighborhood.
Let R10 be a 9 counter and pointer to R1:R9.
Let A0 auto-index register point to I1.
Let R12 be a temporary register
Let R13 be the threshold.

```
Step    Instruction
----    -----------
[1]     I3(8) <== MOVE #0
[2]     R10 <== MOVE #9  "form the filter list pointer"
[3]     R12 <== MOVE #0
[4]     A0 <== INC #I1(8) "form the I1 list pointer"
[5]     ALUA <== 'R10 MUL 'A0D  "process the lists"
[6]     R12 <== ADDSB R12  "sum the result"
[7]     R10 <== R10 DECB 9  "test if done"
[8]     [5] <== JUMP
[9]     [11] <== R12 EQB R13
[10]    I3(8) <== MOVE #1
[11]    Waltpoint.
```

D.3 Edge and curve detection - example

The following is an example of edge and curve detection using the algorithm in Rosenfeld, Lee and Thomas [JohnF70]. In their article, differences of averages are used as measures of texture differences. They discuss a 2-D texture measure D(rs,hk).

```
D(rs,hk)=ABS(
        [a(h+r,k+s)+...+a(h+r,k-s)+...+
         a(h+1,k+s)+...+a(h+1),k-s)]
       -[a(h,k+s)+...+a(h,k-s)+...+
         a(h-r+1,k+s)+...+a(h-r+1,k-s)])/(r*(2s+1.
```

They suggest using the measure

```
D(23,hk)*D(43,hk)*D(83,hk)
```

to separate regions by horizontal edges. For edges in other orientations, analogous operations would be optained by rotating D(rs,hk).

For use as an example to measure the complexity of coding these functions in the GPP. Let us look at D(23,hk). For each (h,k) in the picture, transform its position to (0,0). Then D(23,hk) reduces to D(23,00).

```
D(23,00)=ABS(
        [a(2,3)+a(2,2)+a(2,1)+a(2,0)+
         a(2,-1)+a(2,-2)+a(2,-3)+
         a(1,3)+a(1,2)+a(1,1)+a(1,0)+
         a(1,-1)+a(1,-2)+a(1,-3)]
       -[a(0,3)+a(0,2)+a(0,1)+a(0,0)+
         a(0,-1)+a(0,-2)+a(0,-3)+
         a(-1,3)+a(-1,2)+a(-1,1)+a(-1,0)+
         a(-1,-1)+a(-1,-2)+a(-1,-3)])/(2(2*3+1.
```

This corresponds to the 4x7 array:

-	-	+	+
-1,3	0,3	1,3	2,3
-1,2	0,2	1,2	2,2
-1,1	0,1	1,1	2,1
-1,0	0,0	1,0	2,0
-1,-1	0,-1	1,-1	2,-1
-1,-2	0,-2	1,-2	2,-2
-1,-3	0,-3	1,-3	2,-3

The program to compute D23 is given in terms of macros which the compiler/assembler for the RTPP might have implemented.

```
Define D23 = <  RY <== MOVE #256;
        A:      DOLINE (y);
                RY <==RY DECB B;
                A <== JUMP;
        B:      HLT;>

Define DOLINE (y) = < "get data";
                GETLINE(y,#3,R0);
                GETLINE(y,#2,R258);
                GETLINE(y,#1,R516);
                GETLINE(y,#0,R774);
                GETLINE(y,#-1,R1032);
                GETLINE(y,#-2,R1290);
                GETLINE(y,#-3,R1548);
        "Set up line pointers - note buffers are 258"
                A1 <== MOVE #table;
                'A1I <== MOVE #R0;
                'A1I <== MOVE #R258;
                'A1I <== MOVE #R516;
                'A1I <== MOVE #R774;
                'A1I <== MOVE #R1032;
                'A1I <== MOVE #R1290;
                'A1I <== MOVE #R1548;
        "Set up output buffer"
                'A1I <== MOVE #R1806;
        "Do 256 direction list operations/line"
                rcount <== MOVE #256;
        A:      TXT47(table)
                rcount <== rcount DECB B;
                A <== JUMP;
        B:      SAVELINE(y,R1807);>

Define SAVELINE (sline,r) = < "copy line buffer "r" in gr to line
        'Line' in buffer memory BM3"
                IOCLR;
                XRST <== MOVE #(<I3>,<x>);
                A1 <== MOVE #R;
        A:      I3(5) <== MOVE 'A1I;
                MOVEB #(<I3>,<X>), A, XCLKB;
                BMIO #(256-lines,16-bit,out,horiz,BM3,I3),vsline,LINE;
                >

Define GETLINE (sline,offset,r) = < "copy BM1 line 'line' into gr
        Buffer at 'r'.
                Gline <==line ADD offset;
                IOCLR;
                XRST <== MOVE #(<I1>,<X-1,X,X+1>);
                BMIO #(256-lines,low,in,horiz,BM1,I1,y),sline,LINE;
                A1I <== MOVE #R;
        " Get the x=-1 boundrypoint "
                'A1I <== MOVE I1(4);
        A:      'A1I <== MOVE I1(0);
                MOVEB #(<I1>,<X-1,X,X+1>), A, XCLKB;
        " Get pixels x=256,257 boundries "
                'A1I <== MOVE I1(4);
                'A1I <== MOVE I1(0); >

Define TXT47 (table) = < "do a 4 X 7 local texture operation"
                rcount <== MOVE #7;
                A2 <== MOVE #table; "buffer pointer"
                rsum <== MOVE #0;
        A:      A3 <== MOVE ´A2I; "buffer data"
                ALUA <== MOVEN ´A3I;
                ALUA <== SUBST ´A3I;
                ALUA <== ADDSB ´A3I;
                ALUA <== ADDSB ´A3I;
                rsum <==ADDSB rsum;
                rcount <== rcount DECB B;
                A <== JUMP;
        B:      A3 <== INC ´A2I;
                ´A3I <== MOVE rsum;>
```

The timing for the complete algorithm may be determined as follows: each output line has 7 inputs or a total of 8 BM random I/O accesses/line. Since the total maximum 1 way access and transfer time for a BM is 147 msec, the I/O time is 8*0.147 seconds = 1.2 seconds. Assuming an average instruction time of 300 nsec/instruction, exclusive of I/O, the following times for the above macros are computed:

Macro	#Instr. Execs./Call	#Times called	Total # instr
TXT47	5 + 8*7 = 61	256x256=65536	4.00xE+5
GETLINE	8 + 2*256 = 520	7*256 = 1792	9.31xE+5
SAVELINE	4 + 2*256 = 516	256	1.32xE+5
DOLINE	10 + 2*256=522	256	1.22xE+5
D23	1 + 2*256 = 513	1	513

The total number of instructions is on the order of 1.59 million and takes (at 300 nsec/instruction) about 0.47 seconds. Therefore, the total D23 macro execution takes on the order of 1.7 seconds it should be noted that the D23 algorithm could be done more directly from the line buffers for any size d(rs,00) and would run faster because data would not have to be copies into the GR.

D.4 Histogram computation - example

The following program will compute the gray scale histogram of the I1 picture and leave the results in general registers R0 (address 0) to R63 (address 63). It is assumed that R[0:63] are initialized to 0 by the PDP8e. After the GPP is finished, the PDP8e may read the general registers to access the resultant histogram. The algorithm is as follows:

```
        For all picture pixels
                DO R[I1(8)]<==R[I1(8)]+1;
```

```
step    Instruction
----    -----------
1       'I1(8) <== INC  'I1(8)
2       Haltpoint.
```

D.5 Haltpoint I/O - example

This example shows roughly how "haltpoint" I/O would be done for I1==>I3. The "haltpoint" procedure first obtains buffer memory data line by line and computes some function on a 3x3 neighborhood. It then outputs line I3 into BM3.

Raster I/O algorithm

```
[1.0] Load I1 buffer Y-1, Y, and Y+1 with the first 3 lines
      of BM data.

      IOCLR;
      ycounter <== MOVE #0;
      ycounter <== INC ycounter;
      [2.1] <== ycounter NEB #256;
      YRST <== MOVE <I1,I3>;
      BMIO #(256-lines,low,in,horiz,BM1,I1,Y+1),ycounter,LINE;

      YCLK <== MOVE <I1>;
      BMIO #(256-lines,low,in,horiz,BM1,I1,Y+1),ycounter,LINE;

      YCLK <== MOVE <I1>;
      BMIO #(256-lines,low,in,horiz,BM1,I1,Y+1),ycounter,LINE;
[1.1] Reset the X counters to the front of the line.

      XRST <== MOVE #( <I1,I3>, <X-1,X,X+1> );

[1.2] Process first line and store in I3 buffer. That is
      Perform I1*==>I3.

      [Neighborhood procedure];
      MOVEB #( <I1,I3>, <X-1,X,X+1> ), [1.2], XCLKB;

[2.0] Write out the I3 buffer and then load next BM line in I1
      buffer.

      BMIO #(256-lines,low,out,horiz,BM3,I3,Y),ycounter,LINE;
      YCLK <== MOVE #( <I1,I3> );
      [3.0] <== JUMP;

[2.1] BMIO #(256-lines,low,in,horiz,BM1,I3,Y),ycounter,LINE;
      [1.1] <== JUMP;

[3.0] End. Stop GPP and notify PDP8e.
      HLT
```

D.6 Random Neighborhood Accessing - example

In the case where one whishes to random access a neighborhood, quite a savings may be realized by using the neighborhood fetch instruction to fetch BM data into the current neighborhood of line buffer j. The following sequence will process neighborhood (x,y) of BM3 and store it into pixel (x,y) of BM7.

```
yxaddress <== x MAKYXADDR y;
BMIO #($low ! $BM7), yxaddress, GETI1;

[Process the current I1 neighborhood==> I3(8)]

PBM7 <== MOVE yxaddress;
'PBM7 <== MOVE I3(8);
```

D.7 Area and perimeter computation - example

Example 7 shows how one might compute total area (sum of pixels of an object > threshold) and total perimeter (total count of entrance and exit pixels of detected regions) of all objects in the scene.

```
        " Compute area "
        START:  area <== MOVE #0;
        A:      C <== I1(8) GEB th;
        B:      Haltpoint;
                A <== JUMP;
        C:      area <== INC  area;
                B <== JUMP;



       "Compute perimeter of BMi under mask BMj.
         note that switch is true inside of a blob".
        Define TRUE=1, FALSE=0;
        Variable MASKEDI1, perimeter, INBLOB, THRESHOLD,

                perimeter <==MOVE #0;
                IOCLR;
                ycounter <== MOVE #0;

        "Get the next line for the image and its mask"
       GETLINE: BMIO #(256-lines,low,in,horiz,BMi,I1,Y),sline,LINE;
                BMIO #(256-lines,low,in,horiz,BMj,I2,Y),I2Y,LINE;
                switch <== MOVE FALSE;

        "Position the line buffers' dynamic address vectors at left"
                XRST <== MOVE #(<I1,I2>,<X>);
        TEST:   MASKEDI1 <==I1(8) AND I2(8);

        "Test if in a blob"
                INBLOB <== MASKEDI1 GEB THRESHOLD;

        "Test if leaving a blob"
                YES <== switch BNE #0;
        NEXTX:  MOVEB #(<I1,I2>,<X>), TEST, XCLKB;

        "Reset switch after each line"
                switch <== MOVE FALSE;
                YCLK <== MOVE <I1,I2>;
                ycounter <==INC ycounter;
                GETLINE <== ycounter NEB #256;

        "Send data back to PDP8e which sends it to PRDL"
                GOUT <== MOVE perimeter;

        "Signal DDTG that the function is done"
                HALT;

        "Test if just entered an object"
        INBLOB: YES <== switch EQB #0;
```

```
            NEXTX <== JUMP;

        "Yes, a perimeter point, increment perimeter
            and reverse the switch"
        YES:    perimeter <==INC perimeter;
            switch <==  MOVE TRUE;
            NEXTX <== JUMP;
```

Each pixel in computing the perimeter takes at most 8 instructions in addition to 256 input instructions at 300 nsec/avg-instruction (exclusive of I/O). This gives a computation time of about 0.15 seconds and 0.147 seconds I/O. The total time is about 0.3 seconds. As the I/O cost in time is comparable to the computation time, it would be more economical to combine several simple primitives together such as area, perimeter, etc. while doing a single I/O traversal through the BM.

APPENDIX E

GPPASM BNF Grammar Specification

The BNF grammar specification is given for the GPPASM assembler [Lem76b] to be used to assemble RTPP programs. Note that MAINSAIL will generate GPPASM assembly language output. The MICROMODE source programs on the other hand are coded manually.

Note that ID is any identifier which is not a keyword in the grammar, INT is any integer, and <text> is any text not including the symbols \\, ", or ;. EPSILON is the null string.

Various terminal symbols whose meaning is not apparent are defined (including the semantics) in [Lem76a].

```
<program>::= <GPPsegment> ^Z

<GPPsegment>::= <GPPsegment> <statement> | <statement> |
                EPSILON

<statement>::= <compiler-mode-statement> crlf |
                <section-statement> crlf |
                <expunge-statement> crlf |
                <number-mode> crlf |
                <comment> |
                <require-statement> crlf |
                <PM-label> <PM-statement> crlf |
                <GR-label> <GR-statement> crlf |
                <MCPM-label> <microinstruction> |
                <MM-statement>

<compiler-mode-statement>::= GPPMODE | MICROMODE

<section-statement>::= SECTION <file>

<expunge-statement>::= EXPUNGE ID | EXPUNGE <class type>
<class type>::= <PM-class> | <GR-class> | <MCPM-class>
<PM-class>::= 1
<GR-class>::= 2
<MCPM-class>::= 3

<number-mode>::= DECIMAL | OCTAL

<require-statement>::= REQUIRE <file> SOURCETIME |
                REQUIRE <file> LOADTIME |
                REQUIRE <file> RUNTIME

<PM-statement>::= <PM-instr.> |
                PMDEF ID = <value> |
                PMORIGIN <value> |
                GPPSTART <PM-label> |
                EPSILON
```

```
<PM-instr.>::= <GPP-opcode> <P1> <delim> <P2> <delim> <P3>

<GR-statement>::= GRBLOCK <GR-list>
                  GRDEF ID = <value> crlf |
                  GRORIGIN <value> crlf |
                  EPSILON

<macroinst.def.>::= <macroinst.def.> <microinstruction> |
                  <microinstruction> crlf

<microinstruction>::= / <MCPM-statement> \ |
                  <MCPM-statement> |
                  <MCPM-statement> crlf
<MCPM-statement>::= <MCPM-instr.> |
                  MCPMDEF ID = <value> |
                  MCPMCRIGIN <value> |
                  EPSILON

<MCPM-instr.>::= <MCPM-instr.> <delim> <MCPM-opcode> |
                  <MCPM-opcode>

<MM-statement>::= OPRMAP <GPP-opcode-value> = <MCPM-label>
<GPP-opcode-value>::= <value (i.e. base value+instance value )>

<file>::= <device> <fname> . <ename>
<device>::= SYS: | DSK: | DSKB: | DSKC: | DSKD: | DSKE: |
                DSKF: | DSKG: | DSKH: | DTA0: | DTA1:
<fname>::= ID
<ename>::= ID

<comment>::= Comment text ; | " text "

<P1>::= ' <GR-address> | <GR-address> | # <value>
<P2>::= ' <GR-address> | <GR-address> | # <value> | <I/O list>
<P3>::= ' <GR-address> | <GR-address> | <PM-address>

<value>::= <land> | <value> ! <value1> | <ae>
<value1>::= <ae> | <land>
<land>::= <value> & <value1>

<ae>::= <sae>
<sae>::= <term> | <sae> + <term> | <sae> - <term>
<term>::= <factor> | <term> * <factor> | <term> % <factor>
<factor>::= <primary>
<primary>::= <PM-label> | <GR-label> | <MCPM-label> |
                ( <value> ) | + <primary> | - <primary>

<PM-label>::= <label> | EPSILON
<GR-label>::= <label> | EPSILON
<MCPM-label>::= <label> | EPSILON
<label>::= ID :

<PM-address>::= <PM-label>

<GPP-opcode>::= | BMOVES | BAND | BOR | BREV | BMOVLL | BMOVHH |
              BMOVLH | BMOVHL | BGET1 | BGET2 | BPUT | BSETL | BSETH
              | BSTLSA | BSTHSA | BSTLSB | BSTHSB | MOVE | MOVEB |
                                  E
```

```
                    ADD  | SUB | MUL | DIV | MAX | MIN | MOVBIT | AND | OR
                    | XOR | EQV | NOR | NAND | IMPLIES | BUTNOT | SHFTR  |
                    SHFTL  | ASR | ASL | ROTR | ROTL | REV | MAKXYA | GT |
                    LT | GE | LE | EQ | NE | BMIO | INC | DEC  |  MINUS  |
                    BSWAP  | ADDSA | SUBSA | MULSA | DIVSA | MAXSA | MINSA
                    | MVBTSA | ANDSA | ORSA | XORSA  |  EQVSA  |  NORSA  |
                    NANDSA  |  IMPLSA | BUTNSA | SFTRSA | SFTLSA | ASRSA |
                    ASLSA | ROTRSA | ROTLSA | MKYXSA | GTSA | LTSA |  GESA
                    | LESA | EQSA | NESA | ADDSB | SUBSB | MULSB | DIVSB |
                    MAXSB | MINSB | MVBTSB | ANDSB | ORSB | XORSB |  EQVSB
                    | NORSB | NANDSB | IMPLSB | BUTNSB | SFTRSB | SFTLSB |
                    ASRSB | ASLSB | ROTRSB | ROTLSB | MKYXSB | GTSB | LTSB
                    |  GESB | LESB | EQSB | NESB | INCB | DECB | GTB | LTB
                    | GEB | LEB | EQB | NEB | DADD | DSUB | DMUL | DDIV  |
                    DMAX | DMIN | DGT | DLT | DGE | DLE | DEQ | DNE | DAND
                    | DOR | DXOR | DEQV | DNOR | DNAND | DIMPLI | DBUTNO |
                    DMINUS | DINC | DDEC | DSWAP | DCOMP | DREV | DSHFTR |
                    DSHFTL | DASR | DASL | DROTR | DROTL | DINCB | DDECB |
                    DGTB  | DLTB | DGEB | DLEB | DEQB | DNEB | FADD | FSUB
                    | FMUL | FDIV | FMINUS | FINC | FDEC | FGT | FLT | FGE
                    | FLE | FEQ | FNE | FMAX | FMIN | FLOAT | DFLOAT | FIX
                    | DFIX | FINCB | FDECB | FGTB | FLTB | FGEB |  FLEB  |
                    FEQB | FNEB | JUMP | PUSHJ | POPJ | APPLY

<GR-address>::=  <value> | <GR-I/O-address>

<GR-I/O-address>::= <neighborhood-pixels> | <auto-index> |
                     <indirect-BM-addresses> | <TTY-I/O> |
                     <byte-pointer> | <control-desk> |
                     <status-registers> | <dynamic-address-vectors> |
                     <GR-I/C-registers>

<neighborhood-pixels>::=  I10  |  I11 | I12 | I13 | I14 | I15 |
                    I16 | I17 | I18 | I20 | I21 | I22 | I23 | I24  |
                    I25  | I26 | I27 | I28 | I30 | I31 | I32 | I33 |
                    I34 | I35 | I36 | I37 | I38 <auto-index>::=  A0D
                    |  A0  | A0I | A1D | A1 | A1I | A2D | A2 | A2I |
                    A3D | A3 | A3I | A4D | A4 | A4I | A5D | A5 | A5I
                    |   A6D   |   A6   |  A6I  |   A7D   |  A7  |  A7I

<indirect-BM-addresses>::=  PBM0  | PBM1 | PBM2 | PBM3 | PBM4 |
                    PBM5 | PBM6 | PBM7 |

<TTY-I/O>::= KRB | KSTATUS | TLS | TSTATUS

<byte-pointer>::= PPOINT | GPNT1 | GPNT2

<control-desk>::=  SW1  |  SW2  | SW3 | SWA | DSPLYA | DSPLYB |
                    DSPLYC | KNOB01 | KNOB23 | KNOB45 | KNOB67 |

<status-regists>::= PDLCNT | PDL | FXAR | DRA | DRB  |  EXAR  |
                    EEXAR | EDRB | STATUS

<dynamic-address-vectors>::= I1XM | I1X | I1XP | I1Y |  I2XM  |
                    I2X | I2XP | I2Y | I3XM | I3X | I3XP | I3Y

<GR-I/O-registers>::= GIN | GOUT
```

```
<I/O list>::= ( <list> ) | <I/O list> ! <I/O symbol> |
                <I/O symbol>

<list>::= <list> ! <I/O symbol> | <list> ! INT | <I/O symbol> |
            INT

<I/O symbol>::= $I1 | $I2 | $I3 | $XP | $X | $XM | $YP | $Y |
                $YM | $RIGHT |$LEFT | $VERTICAL | $HORIZONTAL |
                $BM0 | $BM1 | $BM2 | $BM3 | $BM4 | $BM5 |
                $BM6 | $BM7 | $DOUBLEBUFFER

<GR-list>::= <GR-allocation-size> | 0 <preload>
<GR-allocation-size>::= INT
<preload>::= <list-of-values> | \\<text>\\
<list-of-values>::= <list-of-values> , <value> | <value> |
            [ <repeat-times> <delim.> <list-of-values> ]
<repeat-times>::= <value>
<text>::= text string containing no \\, ", or ;

<MCPM-opcode>::= <Mreg> | <Oreg> | <MCPM-ALUs> | <MCPM-bits>
<Mreg>::= [ <16-bit value> ]
<Oreg>::= ( <7-bit value> )
<MCPM-ALUs>::= AL1$ <ALU-value> | AL2$ <ALU-value>
<ALU-value>::= 00| 01 | 02 | 03 | 04 | 05 |06 | 07 | 10 | 11 |
                12 | 13 | 14 | 15 | 16 | 17
<MCPM-bits>::=  P1>A1 | P2>A1 | P3>A1 | CA1>A1 | ALA>A1 |
            ALB>A1 | ALC>A1 | PDA>A1 | PA2>A1 | MR>A1 |
            GTC>A1 | P1>A2 | P2>A2 | P3>A2 | CA2>A2 | ALA>A2
            | ALB>A2 | ALC>A2 | PDA>A2 | MR>A2 | A1P>A2 |
            P1>D1 | P2>D1 | P3>D1 | CA1>D1 | ALA>D1 | ALB>D1
            | ALC>D1 | PC>D1 | MR>D1 | P1>D2 | P2>D2 | P3>D2
            | CA2>D2 | ALA>D2 | ALB>D2 | ALC>D2 | PC>D2 |
            MR>D2 | W1HD1T | W1LD1T | W1HD1F | W1LD1F |
            W2HD2T | W2LD2T | W2HD2F | W2LD2F | READA1 |
            READA2 | D1>ALA | D1>ALB | D1>PCT | D1>PCF |
            D2>ALA | D2>ALB | D2>PCT | D2>PCF | ALUSET |
            ALUCLR | ALU10 | ALU11 | ALU12 | ALU13 | ALU20 |
            ALU21 | ALU22 | ALU23 | MR0 | MR1 | MR2 | MR3 |
            MR4 | MR5 | MR6 | MR7 | MR8 | MR9 | MR10 | MR11
            | MR12 | MR13 | MR14 | MR15 | MOP0 | MOP1 | MOP2
            | MOP3 | MOP4 | MOP5 | MOP6 | MOP7 | MOP>OP |
            MR>MC | DECMC | JMPMC0 | JMPT | JMPF | PUSHJT |
            PUSHJF | POPJT | POPJF | INCPDL | DECPDL |
            SAVEPT | SAVEPF | ISALDT | ISALDF | INCGTC |
            INCPCT | INCPCF | MHALT | TEST | SET

<delim>::= , | space | tab
```

INDEX