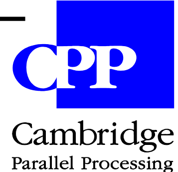


Gamma II Plus Technical Overview

Reference: man.gen.01 Version: 1.0

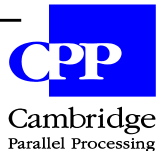
Cambridge Parallel Processing (CPP) endeavors to ensure that the information in this document is correct, but does not accept responsibility for any error or omission. Any procedure described in this document for operating CPP equipment should be read and understood by the operator before the equipment is used. To ensure that CPP equipment functions without risk to safety or health, such procedures must be strictly observed by the operator. The development of CPP products and services is continuous, and published information may not be up to date. Any particular issue of a product may contain part only of the facilities described in this document, or may contain facilities not described here. It is important to check the current status with CPP. Specifications and statements made as to performance in this document are CPP estimates intended for general guidance. They may require adjustment in particular circumstances, and are therefore not formal offers or undertakings.

Statements in this document are not part of a contract or program product license, save insofar as they are incorporated into a contract or license by express reference. Issue of this document does not entitle the recipient to access or to use the products described, and such access or use may be subject to separate contracts or licenses.

The DAP is a registered trademark of Cambridge Parallel Processing. Cambridge Parallel Processing acknowledges all other trademarks to be the property of their respective owners.

No part of this publication may be reproduced in any form without written permission from Cambridge Parallel Processing.

© Cambridge Management Corporation, Cambridge Parallel Processing and
Cambridge Parallel Processing, Ltd.



About this Overview

What this document is about: This document gives a technical overview of the architecture, programming languages and supporting software of the Gamma II^{Plus} series of computers. It serves as an introduction to the more detailed information available in more specialized Gamma II^{Plus} product manuals.

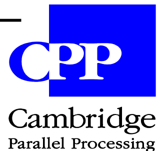
Who this document is for: This overview is aimed at the technical systems staff and programming staff of potential customers who wish to gain an understanding of the Gamma II^{Plus}.

How this document is organized: Apart from Section 1, which is a general introduction, this document contains two sections:

- Section 2, "Gamma II^{Plus} Hardware"
- Section 3, "Gamma II^{Plus} Software"

These sections are designed to be easily readable independently of one another.

Related documents: All CPP Gamma II^{Plus} documentation can be considered to be related to this overview. The "Documentation Map" at the back of this manual enables users to view a complete schema of all CPP manuals, and enables online users to access these manual interactively.



Contents

1 Introduction 1-1

About CPP 1-1
CPP's Product Line 1-1
Application Areas for the Gamma II^{Plus} 1-1
Applications Support 1-2
For More Information 1-2

2 Gamma IIPlus Hardware 2-1

In this Section 2-1
Overview 2-2
Processor Elements 2-4
1-Bit Processors 2-4
PE Connections 2-5
Assembly Language for 1-Bit Processors 2-6
8-Bit Processor 2-7
Assembly Language for 8-Bit Processors 2-9
Array Memory 2-10
Master Control Unit 2-11
Master Control Chip 2-11
Scalar Coprocessor 2-12
Array Support Unit 2-12
CPAL Sequencer Unit 2-13
Array Interface 2-13
VMEbus Interfaces 2-13

Diagnostic Control Unit 2-14
VMEbus and Connections 2-15
VMEbus Slave-Interface 2-15
Gamma II^{Plus} Backplane 2-15
Host Interface 2-16
Fast Input/Output 2-18
D-Plane 2-18
Gamma/Input Output Controller 2-20
GIOC Control 2-22
Software for the GIOC 2-22
External Adaptor Units 2-22

3 Gamma II^{Plus} Software 3-1

In this Section 3-1
Overview 3-2
Summary 3-3
C++ 3-4
AP Class Library 3-4
Object Declarations 3-5
Operations and Functions 3-6
Components 3-7
Examples 3-8
Fortran-Plus 3-11
Data Modes 3-11
Data Type and Length 3-12
Expressions 3-12
Indexing and Assignment 3-13

Subroutines and Functions 3-15
Error Handling 3-16
Examples 3-17
Assembly Languages 3-19
Program Development 3-20
C++ Environment 3-20
Fortran-Plus Environment 3-20
Application Support Libraries 3-22
Parallel Data Transforms 3-24

1 Introduction

About CPP

Cambridge Parallel Processing (CPP) is a privately held, American-owned company, and is a leading international supplier of advanced, fine grained, massively parallel computer systems.

CPP's Product Line

CPP's product line comprises many different configurations of the Gamma II^{Plus} series of computers. This series offers:

- 1024 and 4096 processor variations
- exceptional price/performance ratio — both for workstations and as network servers
- stand-alone computer system configuration for dedicated applications
- power of up to 2.4 GFLOPS (32-bit) and 123 GOPS (8-bit integer addition)

Based on over twenty years of proven technology, the Gamma II^{Plus} incorporates the fourth generation of CPP's proprietary DAP (Distributed Array of Processors) technology, and employs an SIMD (Single Instruction Multiple Data) architecture.

Application Areas for the Gamma II^{Plus}

The SIMD architecture makes the Gamma II^{Plus} ideal for image and signal processing. It also provides a natural and efficient solution to many problems in banking, insurance, transportation, engineering, medicine, manufacturing and communications — in fact, any application where large data sets are subject to multiple operations.

Application areas include:

- Image Processing
- Signal Processing
- Data Mining
- Neural Networks
- Bio-Sequencing
- Medical Imaging
- Robotic Vision
- 3D Graphics
- C^3 (Command, Communication and Control)
- Data Compression
- Statistical Analysis
- Optical Character Recognition
- Free Text Database Searching
- Computational Fluid Dynamics
- Dynamic Linear Programming
- Minefield Detection

Applications Support

CPP provides an extensive range of powerful software development tools and application support libraries to ensure all software development on the Gamma II^{Plus} fully exploits the system's exceptional power and versatility.

Application developers can select either C++ or Fortran-Plus (Fortran 77 with parallel processing extensions).

For More Information

If you would like further information, please contact one of the CPP offices shown below:

Worldwide Headquarters
16841 Armstrong Avenue
Irvine, CA 92606
USA

Tel: (949) 724 8300
Fax: (949) 724 8399
email: info@cppus.com

European Headquarters
Centennial Court
Easthampstead Road
Bracknell, Berks RG12 1YQ
UK
Tel: (01344) 861024
Fax: (01344) 305544
email: info@cppuk.co.uk

2 Gamma II^{Plus} Hardware

In this Section

This section comprises the following units:

Unit	What is Covered	Page
Overview	An overview of the architecture of the Gamma II^{Plus}: • Processor Elements • Master Control Unit • VMEbus • Fast I/O	2-2
Processor Elements	Details of the 1-bit processor, 8-bit processor and array memory that make up each Processor Element.	2-4
Master Control Unit	Details of the MCU — the processor that controls the Gamma II^{Plus} array.	2-11
VMEbus and Connections	Details of the VMEbus connecting the microprocessor controller, the MCU, Fast Input/Output controllers, and optional links to remote host workstations.	2-15
Fast Input/ Output	Details of the architecture of the FIO and of the Gamma Input/Output Controller.	2-18

Overview

Main Components

The Gamma II^{Plus} has four main components:

- Processor Elements (PEs) with array memory.
- Master Control Unit (MCU) with code store.
- VMEbus with microprocessor controller, host interface (optional) and VME slots.
- Fast Input/Output Controllers (optional).

The relationship between these components is displayed on the following diagram:

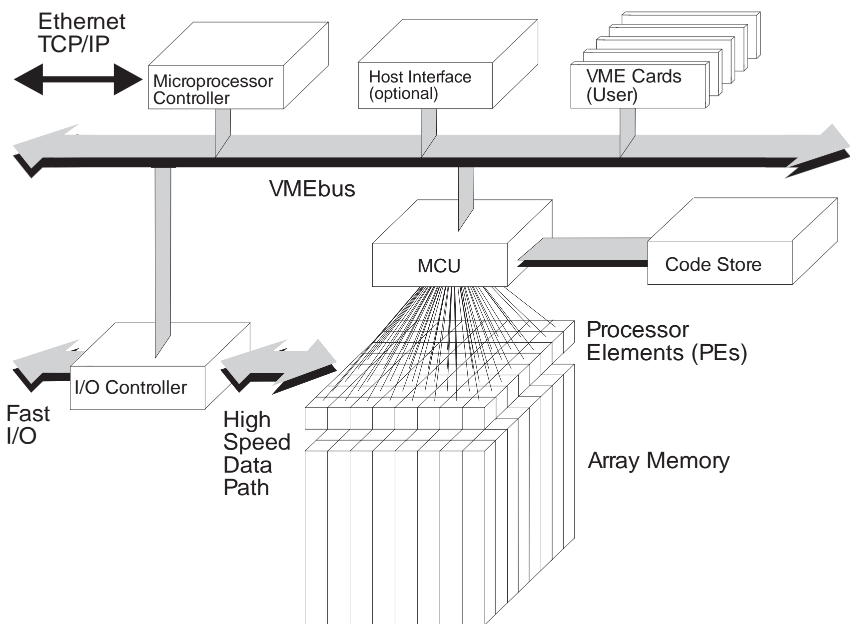


Figure 2.1 *The Main Components of the Gamma II^{Plus}*

Processor Elements

The computational core of the Gamma II^{Plus} is an array of Processor Elements (PEs). The Gamma II^{Plus} achieves its performance through the power of its PEs and their high connectivity.

Master Control Unit

Central control of the Gamma II^{Plus} array is vested in the MCU, which includes a control processor and code store holding DAP instructions. The MCU communicates with the processor array via dedicated address, data and control paths. The PEs operate synchronously under control of the MCU.

VMEbus

The VMEbus interconnects the main components of the system, and provides a link to an optional remote host workstation. It is used for both data transfer and control messages, and may have standard VMEbus cards connected to it.

FIO Controllers

The Fast Input/Output Controllers connect external devices to a high speed data path and so to the array memory of the Gamma II^{Plus}.

Processor Elements

Each PE in the array contains:

- a 1-bit processor
- an 8-bit processor
- array memory

The 1-bit and 8-bit processors work together, enabling each PE to perform tasks ranging from single-bit operations through to multi-byte floating point arithmetic.

Connections between PEs

The PEs are connected to each other in a two-dimensional array; for current Gamma II^{Plus} models this can be either a 64×64 array (Gamma II^{Plus} 4000) or a 32×32 array (Gamma II^{Plus} 1000). (See also “PE Connections” on page 2-5.)

Processor Selection

The Gamma II^{Plus} compilers select which processor to use in a way that is entirely transparent to the user.

1-Bit Processors

Each 1-bit processor (Figure 2.2) has three 1-bit registers called Q, C and A. Each of these registers is used for a variety of purposes.

Q and C Registers

It is convenient to think of the Q register as being an accumulator and the C register as a carry register. Arithmetic and logical functions are performed within the PE by the adder, which adds the contents of the Q register, the C register and a third single-bit input (typically a bit read from array memory), producing a sum and a carry-out. Addition of multiple-bit data is achieved by a sequence of such additions operating on successive data bits, starting at the least significant bit.

A Register

The main purpose of the A register is to provide *activity control*. Some instructions that write a result from the 1-bit processor to array memory are conditional on the value of the A register, thus permitting data to be written or not depending on some local condition.

Q-A-C Register Planes

As all of the PEs run in lock-step, it is convenient to think of the Q, C, and A registers of every 1-bit processor as forming three *register planes*: the Q-plane, the C-plane and the A-plane. These are shown in the 1-bit processor array in the upper part of Figure 2.3 on page 2-6. (Other parts of the figure show the array memory and 8-bit processor array, which are discussed later.)

D Register

The D register shown in Figure 2.2 is used for input or output under control of the Fast Input/Output unit (page 2-18). Data to be output is transferred one plane at a time out of the array memory to the D-plane. The data is then transferred from the D-plane through the

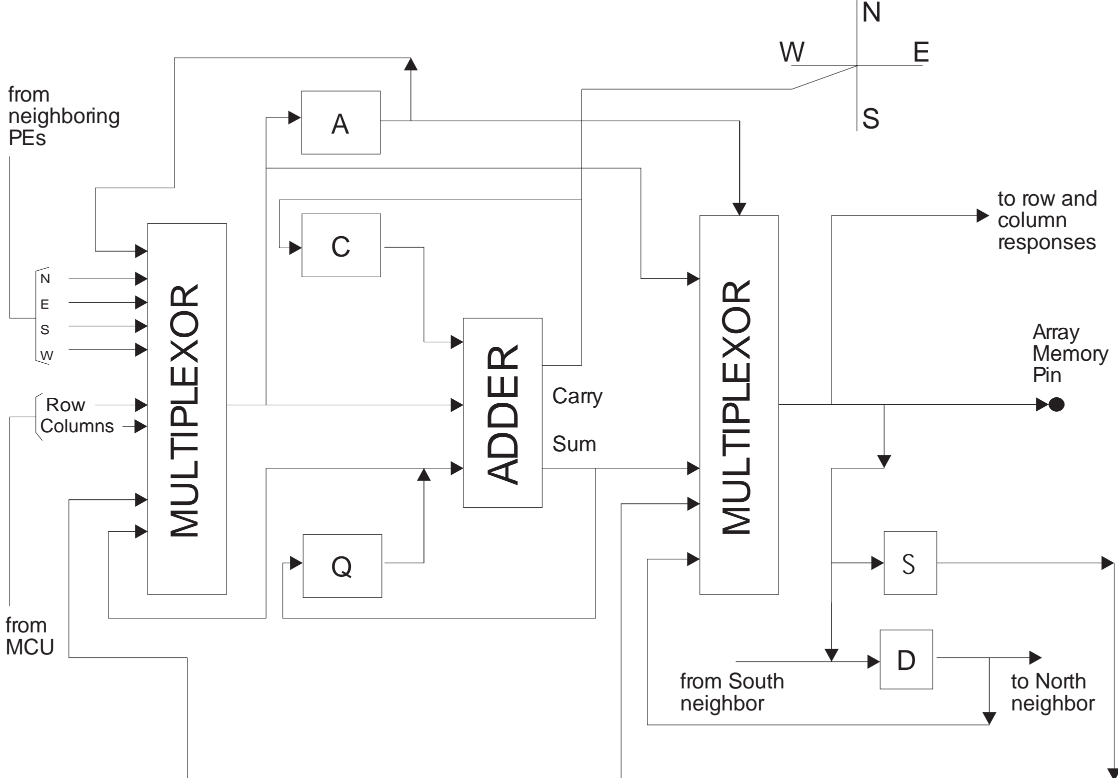


Figure 2.2 *1-bit Processor*

edge of the PE array to the external device, independently of any other processing. Data is input by reversing this procedure.

S Register

The S register is used by instructions that both read from and write to the array memory.

PE Connections

Neighbor Connections

Each PE is connected to four neighbors; north, south, east and west, as shown in Figure 2.4. Using these connections, data can be shifted from the Q register of each 1-bit processor to the Q register of its neighbor in any one of the four directions. PEs at an edge of the array are connected to those at the opposite edge of the array, so that shifts can “wrap-around” if required.

Row and Column Data Paths

PEs are also connected to row and column data paths (Figure 2.4). The data in an MCU register can be broadcast along these paths so that the bit n of the register is received by all the PEs in row n (or column) of the array. In this way a vector of bits can be copied to

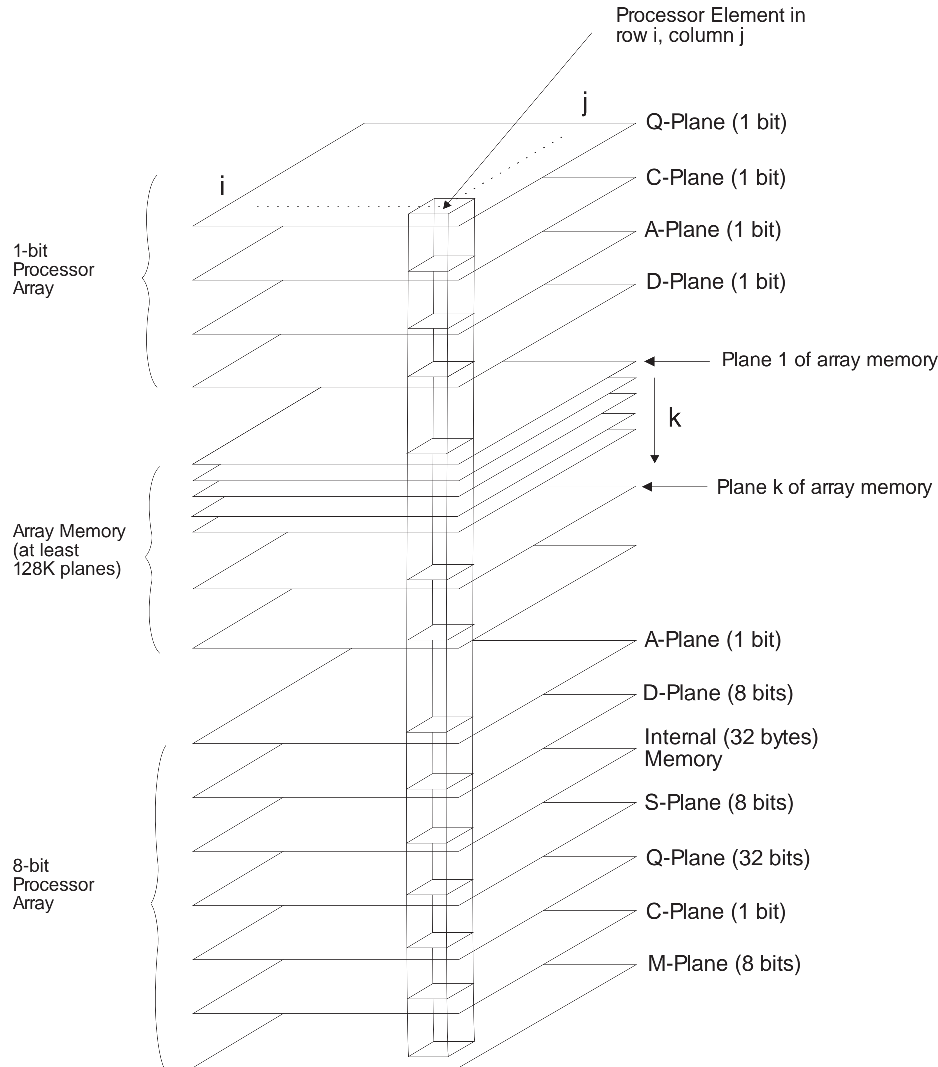


Figure 2.3 *Processor Element Array*

every row (or column) in one instruction. The row and column data paths can also be used to extract a row (or column) and place it in an MCU register or to AND together all the rows (or columns) and place the result in an MCU register.

Assembly Language for 1-Bit Processors

See also "Assembly Language for 8-Bit Processors" on page 2-9.

The 1-bit processors execute the Array Processor Assembly Language (APAL) instructions broadcast to the array by the MCU. APAL also includes scalar, control, and loop control instructions which are executed solely by the MCU. The high-level language

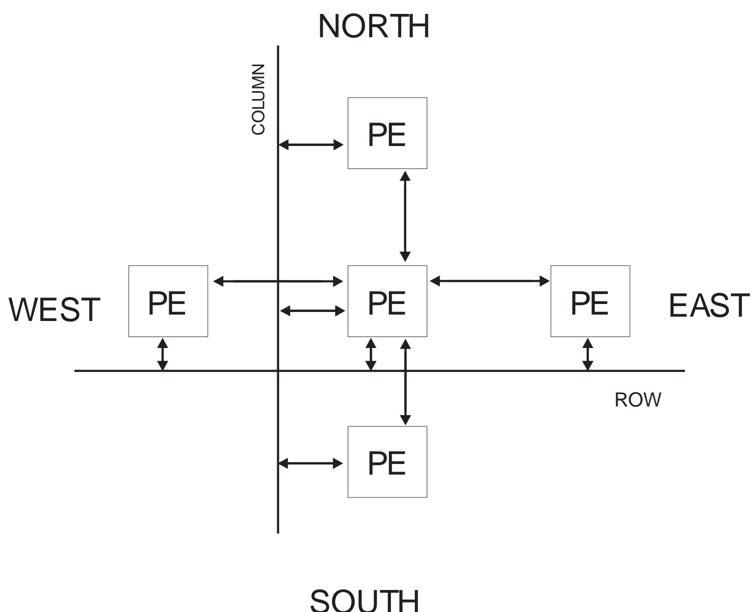


Figure 2.4 *PE Connections*

program (C++ or Fortran-Plus) is compiled into a series of calls to intrinsic functions that are linked into the program. A typical intrinsic might shift a user array by loading each plane of memory in turn into the Q-plane, shifting Q by the required distance, and storing it in the corresponding result plane.

8-Bit Processor

Figure 2.5 on page 2-8 shows the main components of an 8-bit processor. Most of the registers and data paths are 8 bits wide.

Register Names

Many of the register names are the same as those in the 1-bit processors, and, as with the 1-bit processors, can be thought of as forming planes (bottom of Figure 2.3 on page 2-6). However, the 8-bit processors' registers are distinct from the 1-bit processors' registers

Internal Memory

Each 8-bit processor has 32 bytes of internal memory which is used to hold the operands, results and intermediate values of 8-bit processor operations.

D Register

The D register, at the bottom right of Figure 2.5, is used to transfer data to or from the array memory (using a 1-bit data path), and to and from internal memory (using an 8-bit data path).

A Register

The A register shown in Figure 2.5 is a copy of the A register associated with the 1-bit PE, and likewise provides activity control.

Some instructions that write a result from the 8-bit processor to array memory are conditional on the value of the A register, thus permitting data to be written or not, dependent on some local condition.

S Register

The S register is an 8-bit operand register, mainly used when reading data from internal memory.

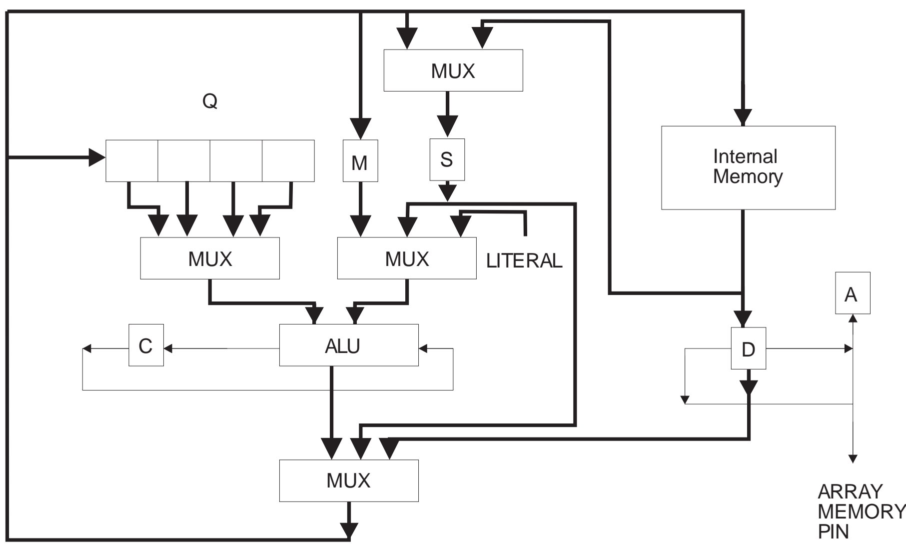


Figure 2.5 *The 8-bit Processor*

Q Register

The Q register acts as an accumulator and is 32 bits wide. Any one of its four bytes can be selected for use as an operand of the 8-bit Arithmetic-Logical-Unit (ALU). The complete Q register can be shifted right by one bit or one byte, and in the latter case the 8-bit value shifted in comes from the ALU via the multiplexer shown below it. Shifting of Q can be done unconditionally or can be dependent on a local condition.

M Register

The 8-bit M register may be shifted left or right. Its least significant bit may control the multiplexer on the ALU output or control the shifting of the Q register. During a multiply the M register holds a byte of the multiplier.

Multiplexer

The multiplexer shown below the ALU in Figure 2.5 allows selection of either the ALU output or the contents of the S register, thus giving a form of conditional control.

Arithmetic Logic Unit

The ALU provides 8-bit addition and subtraction operations, and a variety of bit-by-bit logical operations.The carry out from arithmetic operations can be clocked into the 1-bit C register and used as a

carry in for a following instruction, thus allowing multiple-byte operations to be implemented.

Performance

A Gamma II^{Plus} 4000, running at 30 MHz can perform 4,096 8-bit operations every 33 nanoseconds giving a total rate of 120 GOPS.

Assembly Language for 8-Bit Processors

See also “Assembly Language for 1-Bit Processors” on page 2-6.

The 8-bit processors execute Coprocessor Assembly Language (CPAL) instructions broadcast to the array by the Master Control Unit (MCU). CPAL is used only for intrinsic functions such as floating point arithmetic on arrays.

CPAL Intrinsic Functions Library

A CPP-supplied library of such intrinsics — performing a variety of functions on different types and lengths of data — is loaded automatically at system initialization. The high-level language (Fortran-Plus or C++) compilation system generates calls to these CPAL intrinsics, and each application program is linked to the required intrinsics as it is loaded.

There is no mechanism to call CPAL direct from Fortran-Plus, but CPAL can be called from Fortran-Plus by means of an intermediate APAL routine. Such CPAL code is loaded and linked at the time the APAL routine is loaded, as a part of the complete program.

APAL Control of CPAL

Each CPAL intrinsic comprises a sequence of low-level instructions which, once started, run to completion without looping or branching. All control of the CPAL instruction stream is performed by APAL instructions; these initiate the execution of CPAL intrinsics (with task queueing to a depth of one), and can test for the completion of those intrinsics. Once execution of a CPAL intrinsic has been started, it executes autonomously and in parallel with the execution of subsequent APAL instructions.

Data Transfer

The APAL instruction stream is also responsible for transfer of data between the array memory and the internal memory of the 8-bit processors. At the array memory these data transfers occur one bit-plane at a time. Within the 8-bit processor the data is buffered within the 8-bit D register shown in Figure 2.5, and at every eighth data bit, a complete byte is written to, or read from, the internal memory. If the 8-bit processors are executing CPAL at that time, then a clock cycle is stolen to access the internal memory.

CPAL Pipelining

Usually the compilation system allocates the internal memory in such a way that while a given CPAL intrinsic is being executed, the result of a previous intrinsic is written to array memory, and the operands for the next intrinsic are copied from array memory to internal memory. By this means, an efficient pipeline of operations

is established, making good use of both the processing capability and the bandwidth of array memory.

Array Memory

PE Memory Size

Each PE has a one-bit wide, direct connection to its own section of the array memory. The size of each PE's section of array memory depends on the configuration of Gamma II^{Plus}but is in the range 128 Kbits to 1 Mbit.

Array Memory Shape

The array memory can be regarded as a three-dimensional array of bits consisting of at least 128K *memory planes* (Figure 2.3 on page 2-6). A memory plane consists of the bits at the same address in each PE's memory. Bit (*i, j*) of a memory plane is therefore associated with PE (*i, j*).

The array memory can also be regarded as a sequence of *rows*, the length of a row being determined by the size of the Gamma II^{Plus} PE array. On a Gamma II^{Plus} 1000 machine each memory plane can be regarded as 32 rows of 32 bits each and on the 4000 as 64 rows of 64 bits. In general, each row comprises a number of 32-bit words. On a Gamma II^{Plus} 1000, words and rows are equivalent. On a Gamma II^{Plus} 4000, a row is two words.

Data Representation

The Gamma II^{Plus} is not committed to any particular representation of data and generally regards data as arrays of bits, the interpretation of which (as fixed or floating-point numbers, for example) is defined by the software.

There are many possible mappings, or arrangements of data, on the Gamma II^{Plus}. There are, however, two simple, natural mappings:

- **Horizontal Mode**, in which successive bits of a data item are mapped onto successive bits of a single row of a memory plane.
- **Vertical Mode**, in which successive bits of a data item are mapped onto the same bit position in successive memory planes.

These mappings correspond to the scalar (horizontal) and the vector or matrix (vertical) storage modes of the Gamma II^{Plus} high-level programming languages.

Memory Performance

All processors can simultaneously transfer one bit to or from memory within a single Gamma II^{Plus} clock cycle; this means that a Gamma II^{Plus} 4000, running at 30 MHz, has a transfer rate of 120 Gbits/sec.

Master Control Unit

The Master Control Unit (MCU) is a dedicated pipelined processor whose role is the overall control of the Gamma II^{Plus} array. It includes scalar processing facilities, but is mainly concerned with issuing instruction streams to the array of 1-bit processors and the array of 8-bit processors.

The main components of the MCU are shown in Figure 2.6 on page 2-12, and the functions of each part are outlined in the following sections.

APAL Control of MCU

The MCU is controlled by the APAL instruction stream. Some of these instructions it executes itself as control operations or scalar arithmetic, but most of the instructions are merely decoded and passed to the array for execution by the 1-bit processors.

APAL Code Store

APAL instructions are 32 bits wide and are held in a dedicated code store. This is organized in banks of 1Mbyte, and the MCU may be populated with one, two, or four such banks, giving a maximum configuration of 4Mbyte, or 1M instructions.

Master Control Chip

At the heart of the Master Control Unit (MCU) is the Master Control Chip (MCC), which interfaces to the rest of the system via buses carrying address, data, and decoded array instructions.

The MCU fetches and decodes APAL instructions from the Code Store. A high proportion of APAL instructions executed take one clock cycle, but for those that take longer, a lower level of microcode control is used. The microcode store is also used to decode part of the array instruction, even for single-cycle instructions.

DO Loop

As well as the usual jump instructions, the MCC includes logic dedicated to an APAL DO instruction. This provides efficient looping over a sequence of instructions, such as those used to implement operations on each bit-plane of a data set in turn.

Master Control Chip Registers

Within the MCC, a triple port register file holds 14 general purpose 32-bit wide MCU registers, plus a further register accessible only by privileged (system) code. These registers may hold scalar data or the addresses of array data, and arithmetic, logical, or shift operations may be applied to them. MCU registers may be broadcast to the array, or they may be loaded with data returned from the array.

Array Memory Addresses

The MCC generates addresses for the array memory. Often a complete bit-plane is addressed, but other instructions provide for accessing a row, column or word within a plane. Addresses may be

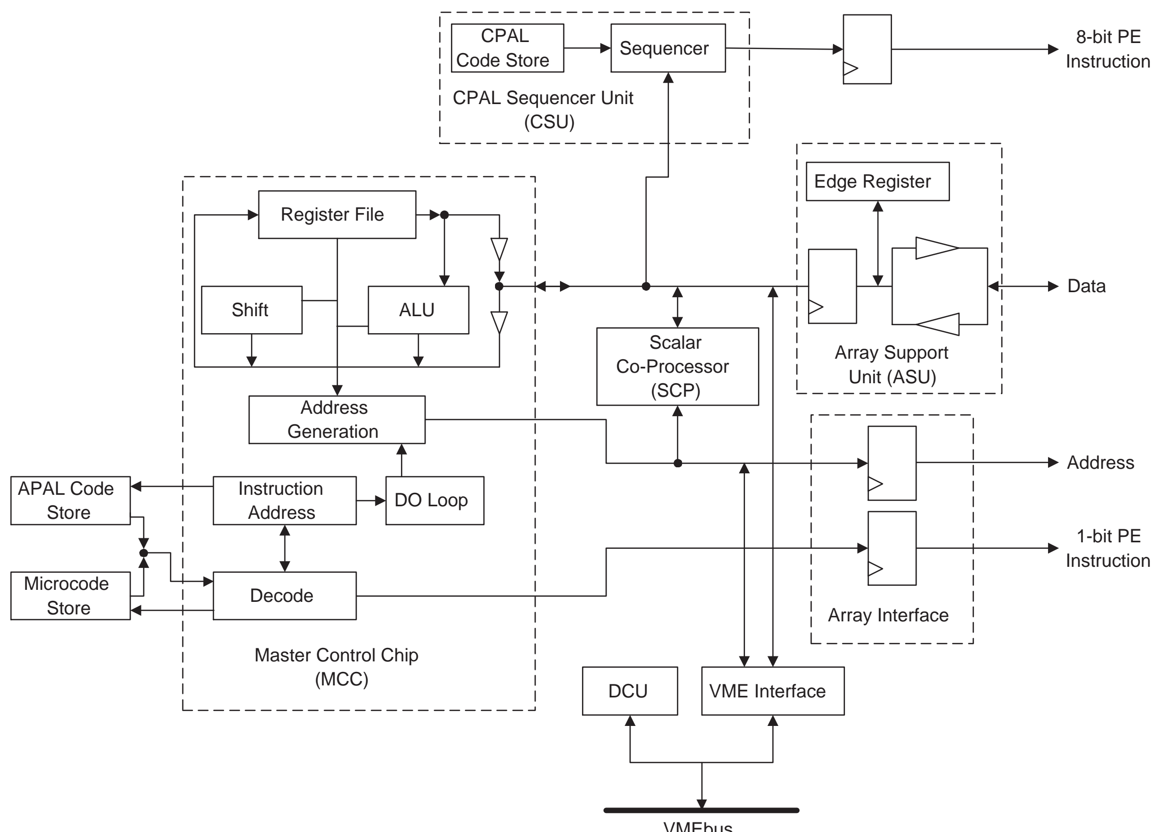


Figure 2.6 *The Master Control Unit (MCU).*

modified by the contents of one of the MCU registers, and may be stepped (incremented or decremented) on each pass of a DO loop, thus providing efficient access to successive planes, rows, or words within array memory.

Scalar Coprocessor

An integer scalar multiplier chip is connected as a coprocessor to the MCC. This extends the range of scalar operations that may be applied to MCU registers.

Array Support Unit

The Array Support Unit (ASU) acts as an interface between the data bus within the MCU, which is 32 bits wide, and the data bus in the array, whose width matches the edge size of the array. For a 64×64 array, the array data bus is 64 bits wide, for example. When sending data to the array, the MCU data may either be replicated or zero extended (if necessary) to fill the array bus. When receiving data from the array, the MCU may (if necessary) select one of the words from the array data bus.

ASU Edge Register

Within the ASU there is a register known as the Edge Register, whose size matches the array edge size. It may thus be broadcast to the array, or its contents loaded direct from the array data bus, regardless of the size of the array. The Edge Register is addressed by the MCU in the same way as the MCU registers. It can be shifted and tested, but cannot otherwise take part in MCU scalar operations.

CPAL Sequencer Unit

Instructions for the 8-bit processors are issued by the CPAL Sequencer Unit. CPAL instructions are held in their own code store which can hold 65536 instructions of 32 bits.

APAL Initiates CPAL Execution

APAL instructions are used to initiate the execution of CPAL code sequences, and to transfer data between array memory and the internal memory of the 8-bit processors. Once a CPAL intrinsic has been started, the CSU continues to fetch CPAL instructions and issue them to the 8-bit processors independent of the APAL instruction stream, until the end of the CPAL intrinsic is reached. Start commands for CPAL intrinsics may be queued to a depth of one, and when one intrinsic ends another may be started automatically from the queue.

APAL instructions can test the state of the CSU to determine whether it is idle, busy, or busy with another intrinsic in the queue.

Array Interface

The array address and the decoded instructions for the 1-bit and 8-bit processors are buffered and distributed by array interface logic. The equivalent function for the array data is implemented by the ASU.

VMEbus Interfaces

The MCU provides both master and slave interfaces to the VMEbus. This allows devices attached to the VMEbus to access data in the array memory, and to load the APAL and CPAL code stores and the microcode store at system initialization, and when loading individual application programs.

Incoming Requests from VMEbus

An incoming request from the VMEbus causes the APAL instruction stream to be suspended while the required memory access is performed under control of the MCC microcode.

Privileged APAL Code

Privileged APAL code can access system resources via the VMEbus, as well as code store and memory-mapped control locations within the Gamma II^{Plus} itself.

Diagnostic Control Unit

The Diagnostic Control Unit (DCU) provides control for the hardware initialization sequence of the MCU and array. It connects to the VMEbus as a slave interface and is thus controlled from that bus.

VMEbus and Connections

The VMEbus interconnects the main components of the system, and provides a link to an optional remote host workstation. It is used for both data transfer and control messages, and may have standard VMEbus cards connected to it. (See Figure 2.1 on page 2-2).

VMEbus Slave-Interface

VMEbus slave-interfaces on the MCU and FIO controllers implement the full protocol as described in the VMEbus specification document (IEEE 1014 rev C.3).

The array memory, the APAL code store, and the CPAL code store together with certain control locations are all mapped into the VMEbus address space. When such an address is detected by the MCU slave interface, the APAL instruction stream is suspended for a few cycles to perform the required access.

Restrictions

Not all VME access modes are supported. In particular, only 32-bit (VME double word) access is supported, and no provision is made for byte or 16-bit word access.

Block transfer is not supported.

Gamma II^{Plus} Backplane

The backplane configurations for the Gamma II^{Plus} 1000 and Gamma II^{Plus} 4000 are shown in Figure 2.7. All boards are 6U VME form factor.

PEs and Array Memory

The Gamma II^{Plus} 1000 array (PEs plus array memory) occupies two boards, and the Gamma II^{Plus} 4000 array, with four times as many PEs, occupies eight boards.

MCU

The MCU occupies a single board in both the 1000 and the 4000.

VME

Further slots are provided for VME boards (up to twelve on the Gamma II^{Plus} 1000 and five on the Gamma II^{Plus} 4000). One of the VME slots is occupied by the microprocessor controller, and a second may be occupied by a remote host interface card; the remaining slots are available for other standard VME cards.

Fast Input/Output

The backplane makes provision for FIO, with two Gamma Input/Output Controller (GIOC) slots plus two external adaptor unit (EAU) slots on both the Gamma II^{Plus} 1000 and the Gamma II^{Plus} 4000.

Microprocessor Controller Board

The Gamma II^{Plus} incorporates a microprocessor controller board. The main function of this is to provide control for external devices such as fast disks. However, it may also be used as a host for the Gamma II^{Plus} (see “Host Interface” on page 2-16).

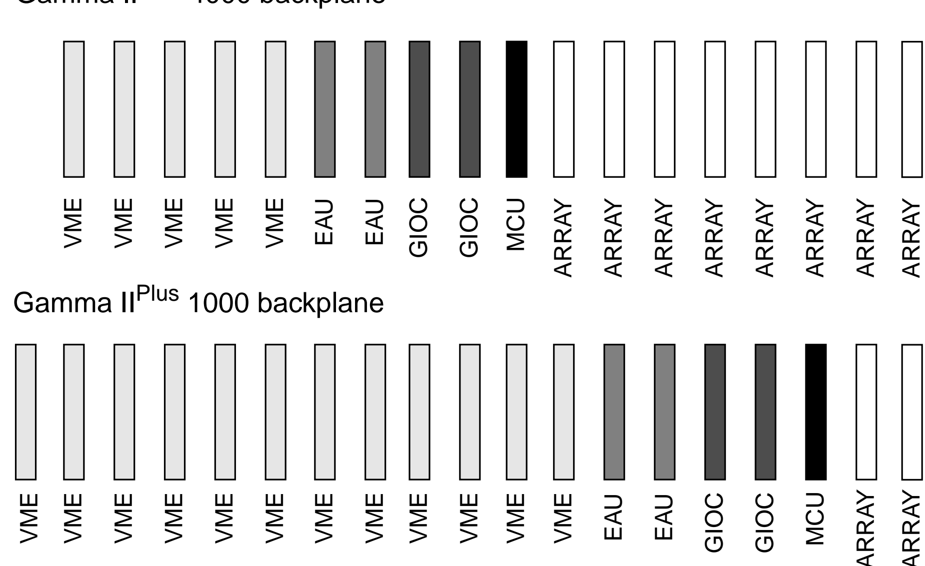


Figure 2.7 *Gamma IIPlus Backplanes*

The board incorporates the VMEbus system controller. It is provided with Ethernet interfaces, and runs VxWorks, a real-time operating system.

Controller Boards

Microprocessor controller boards currently supported are:

- Radstone PPC 603e (supplied as an integrated host or with a Sun host interface)
- Performance Technology PT151E (supplied with VAX/VMS host interface)

Host Interface

The Gamma IIPlus acts under the control of a host computer which controls the loading and execution of programs. In addition, the host can also provide:

- interfaces, such as keyboard and screen
- file storage
- access to software such as editors and compilers

Internal Microprocessor as Host

As noted previously, the internal microprocessor can be used as a host; this allows the Gamma IIPlus to be used in embedded applications or as a stand-alone machine.

External Workstation as Host

The host can also be an external workstation computer running UNIX or VMS, connected to the Gamma II^{Plus} via the optional host interface. In this case, the workstation can provide a complete development environment for Gamma II^{Plus} applications.

The Gamma II^{Plus} is currently available with the following external host configurations:

- Sun host via SBus or PCI bus, and Bit3 VME
- VAX host via Digital Q-bus and IKON VME

Sun as Host

Any standard Sun workstation or server which has an SBus slot available can be used as a host. An SBus interface card is provided which is linked via a cable to a VME card in the Gamma II^{Plus}. The interface is used to provide direct access to the Gamma II^{Plus} by mapping host reads and writes through to the VMEbus as well as DMA (direct memory access) for larger data transfers (more than 64 words). This capability matches the two types of communication required. The first is low latency short message transfers in order to, for example, start a Gamma II^{Plus} application routine. The second is fast bulk data transfers between the host and the Gamma II^{Plus}.

VAX as Host

This interface is achieved by using Digital standard external interfaces, DR11W and DRV11WA, cabled to VME boards with compatible interfaces. Thus the interface is physically similar to the SBus based solution outlined above. The Gamma II^{Plus} includes a VME microprocessor board to provide the intelligence to manage the Gamma II^{Plus} side of the communications link. This is needed because the Digital interfaces are DMA-only, and do not allow the mapping of Digital processor reads and writes through to the Gamma II^{Plus}.

Other Hosts

Other interfaces can be developed to meet customers' requirements.

Fast Input/ Output

The Gamma II^{Plus} architecture lends itself to very fast input and output of data with minimal impact on processing.This unit describes the architecture of the FIO and of the I/O controller, the Gamma Input/Output Controller (GIOC).

D-Plane

The high bandwidth I/O of the Gamma II^{Plus} is achieved through a notional array-sized shift register, known as the *D-plane.* This is made up of the 1-bit D registers of each Processing Element (Figure 2.8). Data is output from array memory by first loading the

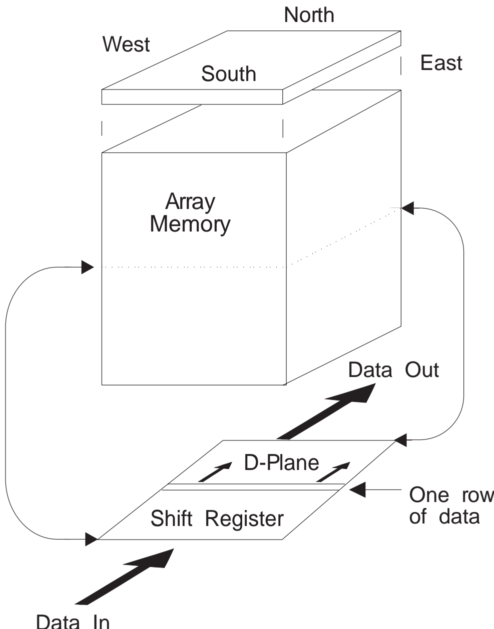


Figure 2.8 *Gamma II^{Plus} Array showing D-Plane*

D-plane with the values from the required memory plane. The data is then shifted across the array and out of the D-plane such that on each cycle an *edge-size* data word (64 bits on a Gamma II^{Plus} 4000, and 32 bits on a Gamma II^{Plus} 1000) is clocked out of one edge.

Concurrent Shifting

Shifting proceeds concurrently with instruction execution in the array. At the same time as data is being shifted out of the array, new data can be shifted into the array from the opposite edge, and a plane of this data written to the array memory to perform input. In this way, data input, data output and instruction execution can all occur at the same time, with only a few percent of the Gamma II^{Plus}

clock cycles being taken from application processing to transfer data between the array memory and the D-plane.

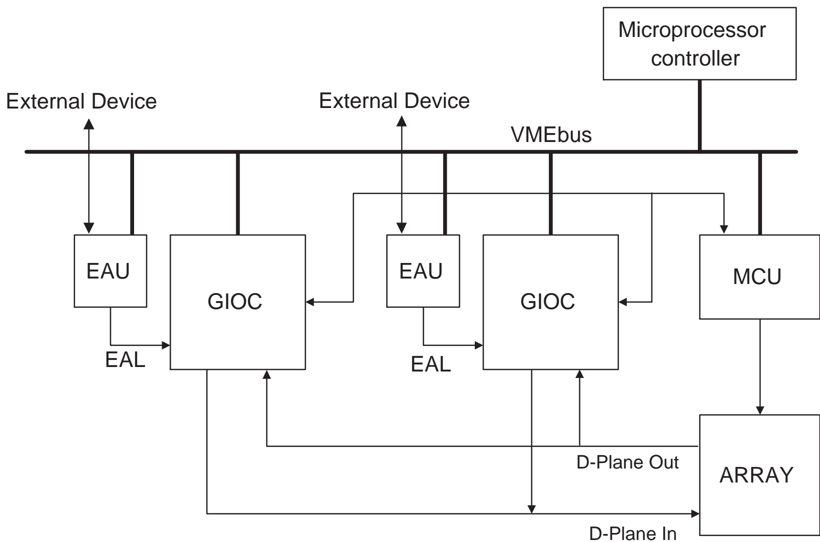


Figure 2.9 *Gamma II^{Plus} Input/Output System*

Performance

On a Gamma II^{Plus} 4000, data can be shifted in and out of the D-Plane at a combined rate of up to 465 Mbytes/sec. The corresponding figure for a Gamma II^{Plus} 1000 is 225 Mbytes/sec.

D-Plane to External Interface Connection

A typical Gamma II^{Plus} system includes one or more device controllers connecting the D-Plane to an external interface (Figure 2.9). The microprocessor controller governs the FIO controllers using instructions sent across the VMEbus. Once an FIO data transfer is initiated, the low-level sequencing is controlled by the FIO controller, rather like a DMA operation in a conventional system; the FIO controller performs D-plane shifting directly, and for loading or storing the D-plane it sends a request to the MCU which performs the required operation using a short microcode sequence.

Number of FIO Controllers

Only one FIO controller can use the D-Plane at a given time. However, there is provision for up to two FIO controllers to be present in a Gamma II^{Plus} system and to arbitrate amongst themselves when they wish to use the D-Plane.

VME Microprocessor Board

The Gamma II^{Plus} includes a VME microprocessor board to provide the intelligence to manage the Gamma II^{Plus} side of the communications link. This is needed because the Digital interfaces

are DMA-only, and do not allow the mapping of Digital processor reads and writes through to the Gamma II^{Plus}.

Gamma/Input Output Controller

The GIOC is a sophisticated high speed input-output device. It connects the Gamma II^{Plus} with interface cards linked to external devices, and can re-order data in transit. The GIOC can be used for input or output, but not both simultaneously.

Links to Customer Devices

The GIOC can easily be connected to customer devices using a simple FIFO-based External Adaptor Link (EAL) via External Adaptor Units (EAUs).

Architecture

The main components of the GIOC are shown in Figure 2.10 on page 2-21. Data can flow in either direction through the GIOC:

- input data is taken from an external device via the EAL interface, and is passed to the array memory via the re-ordering datapath and the D-plane interface
- output data is taken from the array memory via the D-plane interface, and is passed to an external device via the re-ordering datapath and the EAL interface

Re-Ordering Datapath

The GIOC has re-ordering capabilities which provide great versatility and power for data input and output. This versatility is achieved by means of two buffer memories that hold the data as it passes through the GIOC and a sophisticated address generation scheme that allows flexibility in the order of reading or writing the data in the buffer memories.

Corner Turning

Corner turning, or mode conversion, changes the data format between word serial/bit parallel and bit serial/word parallel.

This is done in two stages, one in each buffer memory. In the bit buffer, corner turning is performed on bits within each data byte. In the byte buffer, corner turning is performed on bytes within each word.

Address Sequencing Engines

GIOC address sequencing engines (ASEs) generate sequences of addresses used when reading or writing data in the GIOC buffers and in the array memory. Since the ASEs are programmable they confer great flexibility to the GIOC data re-ordering.

- *ASE0* generates address sequences for the array memory.
- *ASE1* and *ASE2* generate separate address sequences for reading and writing the bit buffer, and control the corner turning within that buffer.

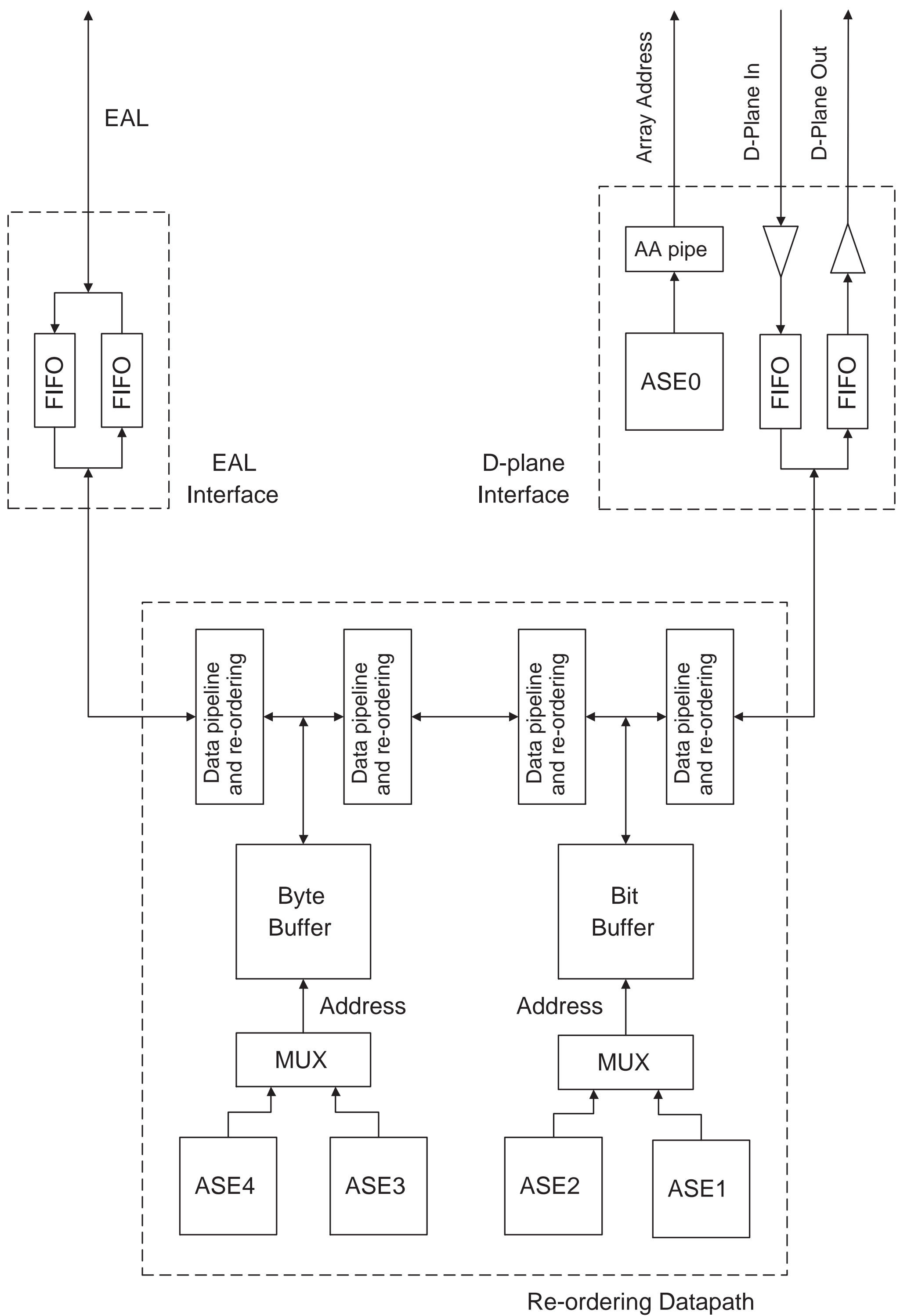


Figure 2.10 *Schematic Diagram of the GIOC*

- *ASE3* and *ASE4* generate separate address sequences for reading and writing the byte buffer, and control the corner turning within that buffer.

D-Plane Interface

The D-plane interface supplies a plane address for the array memory when loading or storing the D-plane, and includes FIFO buffers connected to the D-plane input and output paths.

External Adaptor Link Interface

The EAL is a 32-bit wide bi-directional link controlled by an external adaptor unit (EAU). FIFO buffers permit the EAL to be largely autonomous of the operation of the rest of the GIOC.

Configurations

The GIOC is available in a number of configurations. The data path is 32 bits wide and there is a selection of buffer memory sizes with a maximum of 8 Mbytes.

Performance

The GIOC operates at a sustained rate of up to 60 Mbyte/s.

GIOC Control

The GIOC connects as a slave to the VMEbus. This interface is used to load control programs into the Address Sequencing Engines, and to initiate the high speed data transfers between array memory and the External Adaptor Unit.

Software for the GIOC

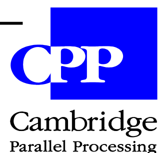
Facilities exist for the automatic generation of GIOC addressing programs for those data reorganizations that can be described using Parallel Data Transforms (see page 3-3). Manual generation of addressing programs is required for more complex cases.

Once GIOC addressing programs are available for the required I/O transfers, the GIOC device driver is used to manage the transfers. Address programs are loaded and transfers started when requested by the application code. Synchronization requests can be made to permit synchronous or asynchronous I/O with single or multiple buffer strategies.

External Adaptor Units

Two external adaptor units are currently available for the GIOC:

- the VME/VSB unit, which supports high performance I/O between the Gamma II^{Plus} and the VMEbus
- the PCI External Adaptor Unit, which supports high performance I/O between the Gamma II^{Plus} and PMC devices on the Adaptor Unit internal PCI bus



3 Gamma II^{Plus} Software

In this Section

This section comprises the following units:

Unit	What is Covered	Page
Overview	An overview of the high-level languages (C++ and Fortran-Plus), of the low-level languages (APAL and CPAL), of program development tools, and of Parallel Data Transforms.	3-2
C++	Details of the C++ language and the additional facilities supplied by CPP.	3-4
Fortran-Plus	Details of the Fortran-Plus language and the additional facilities supplied by CPP.	3-11
Assembly Languages	Notes on the APAL and CPAL assembly languages (used with Fortran-Plus).	3-19
Program Development	Details of the C++ and Fortran-Plus development environments.	3-20
Application Support Libraries	Notes about the General Support and other libraries supplied by CPP to assist program development.	3-22
Parallel Data Transforms	Information about PDTs — arranging data between two data mappings.	3-24

Overview

The Gamma II^{Plus} can be programmed using two high-level languages:

- C++
- Fortran-Plus

These languages include a wealth of parallel data constructs and operations, giving the programmer easy access to the power of the Gamma II^{Plus}.

The Gamma II^{Plus} can also be programmed at a low level using:

- Array Processor Assembly Language (APAL)
- Coprocessor Assembly Language (CPAL)

C++

CPP provides the *C++ AP Class Library* for parallel array processing. This library enables parallel algorithms to be expressed concisely and naturally in a hardware-independent manner. The highly parallel array operations performed by means of the library are transparently implemented using the Gamma II^{Plus} hardware.

Fortran-Plus

Fortran-Plus was developed by adding data-parallel extensions to Fortran 77. The extensions let the user manipulate complete arrays as single objects. Many of the features of Fortran-Plus are shared with Fortran 90.

Assembly Languages

APAL is a low-level language which gives the user complete control over the MCU and 1-bit processors of the Gamma II^{Plus}. APAL and Fortran-Plus (but not C++) routines can be freely mixed, enabling the time-critical parts of a program to be converted into assembly code if required.

The 8-bit processors may be programmed in CPAL, and controlled from APAL code.

Program Development

Fortran-Plus and APAL programs (with associated CPAL routines) can be debugged using a CPP-supplied, powerful, interactive debugging facility known as PSAM (Program State Analysis Mode).

C++ programs are developed using the SPARC Workshop. This is a state-of-the-art development environment produced by Sun Microsystems.

Application Support Libraries

CPP has developed a number of application support libraries to aid the rapid development of programs for the Gamma II^{Plus}. The General Support Libraries includes parallel versions of common mathematical functions and a variety of routines that manipulate arrays of data. The Graphics Library provides basic software to

construct images and display them on a color monitor. There are also libraries for specific application areas:

- Image Processing Library
- Linear Algebra Library
- Transforms Library (mathematical transforms, such as Fast Fourier Transforms)

For an overview of all CPP documentation, including interactive access to documents for online users, see the Documentation Map at the back of this manual.

Parallel Data Transforms

Parallel Data Transforms (PDTs) give the user an elegant means of specifying the mapping of regular data structures on the Gamma II^{Plus} and efficient ways of re-arranging data between mappings. PDTs are a powerful tool in the implementation of many applications and algorithms, such as FFTs and sorting.

Summary

The following table summarizes all the information in this unit:

High-Level Language Options

Facilities	C++	Fortran-Plus
Class Library	AP Class Library	None
Application Support Libraries	• General Support • Graphics • Image Processing • Linear Algebra • Transforms	• General Support • Graphics • Image Processing • Linear Algebra • Transforms
Parallel Data Transforms	Yes	Yes
Assembly Language Programming	Not Available	APAL and CPAL
Development/ Debugging	SPARC Workshop	PSAM

C++

CPP provides high-level array-processing facilities as part of a C++ system. The AP Class Library uses many of the concepts developed in Fortran-Plus and so retains the advantages gained from a clear abstraction of the underlying hardware. However, developing parallel applications in C++ has the following additional benefits:

- programming is in a conventional, modern, widely-used object-oriented language
- object-oriented systems development methodologies can be used
- state-of-the-art facilities for program design, development and debugging are both familiar and readily available

The AP Class Library integrates array processing with C++ by defining new classes for parallel data objects and operations on these parallel objects. The data in the new classes are held and processed on the Gamma II^Plus PEs. The implementation of the operations transparently invokes the Gamma II^Plus to perform compute-intensive tasks.

The following sections outline the main facilities of the AP Class Library and give a number of coding examples using its classes.

AP Class Library

The AP Class Library defines classes that enable the creation of objects that are to be processed using the parallel processing capability of the Gamma II^Plus hardware. For example, vector classes are provided for all the basic C++ types:

- `BoolV` (Vector of booleans.)
- `CharV` (Vector of signed 8-bit characters.)
- `UcharV` (Vector of unsigned 8-bit characters.)
- `ShortV` (Vector of signed 16-bit integers.)
- `UshortV` (Vector of unsigned 16-bit integers.)
- `IntV` (Vector of signed 32-bit integers.)
- `UintV` (Vector of unsigned 32-bit integers.)
- `FloatV` (Vector of 32-bit floating point numbers.)
- `DoubleV` (Vector of 64-bit floating point numbers.)

Similar classes are provided for matrices: `BoolM`, `CharM`, `UcharM` and so on. Classes are also provided for sequences of vectors and for sequences of matrices. The scalar classes `BoolS`, `CharS`, `UcharS` and so on are also provided.

The shape (the number and sizes of the dimensions of the objects) of vectors, matrices and sequences can be described using the classes `ShapeV`, `ShapeM` and `ShapeSeq`.

Object Declarations

Creating and Destroying Objects

Objects of the above classes are created and destroyed using the constructor and destructor mechanisms of C++.

Declaring Vectors and Matrices

Vectors and matrices may be declared using named shapes. For example:

```
ShapeM    s(30,100);     // 30*100 matrix shape
IntM      myMat(s);      // 30*100 integer matrix
```

They may also be declared by anonymous shapes, as in:

```
// 30*100 integer matrix
IntM      myMat(ShapeM(30,100));
// 1000-component bool vector
BoolV     v(ShapeV(1000));
// sequence of 10 1000-component vectors
BoolVseq  vs(ShapeSeq(10),ShapeV(1000));
```

Dimensions can be used directly when declaring vectors and matrices, as in:

```
CharM     myMat(40,50);  // 40*50 character matrix
```

Declaring Local AP Objects

Unlike C++ local arrays, locally declared AP objects may be dynamically sized as in the following declaration where the dimensions are given by the values of `d0` and `d1`:

```
IntM dMat(d0,d1);// d0*d1 integer matrix
```

Default vector and matrix shapes can be made effective by means of `defShape` functions. These may then be used in AP declarations having no explicit shape. This default shape mechanism is particularly useful in situations where a number of objects are declared having the same shape. For example:

```
ShapeM::defShape(d0,d1);// default matrix shape is d0*d1
IntM m1,m2,m3,m4;        // all d0*d1 matrices
```

Declaring Vectors and Matrices Stored on the Heap

Vectors and matrices stored on the heap are declared using the `new` operator:

```
ShapeM::defShape(d*d);   // default matrix shape is d*d
IntM* p = new IntM;      // p points to newly allocated
                         // d*d matrix on heap
```

Operations and Functions

Operations on vectors and matrices fall into two categories: componental and aggregate. The componental operations correspond to scalar operations applied to each component of the vector or matrix. Aggregate operations are non-componental and generally produce a result of a different class from the input.

Componental Operations

Basic unary operations may be applied to vectors and matrices and binary operations may be applied to corresponding components of conforming objects. Binary operators may be applied between one scalar and one vector or matrix operand. Assignment operators and the increment and decrement operators are supported. Functions, rather than operators, are used for componental operations such as square root and logarithm. The following statements do no meaningful calculation but are given to illustrate the use of componental operations:

```
ShapeM::defShape(30,40);
FloatM    x,y,z;
FloatS    s;
x = y + z;
x += y;
z = s * y;
z = y * (x / 4.0);
y *= s;
y -= 3.4;
++x;
x = sqrt(y * z);
```

Aggregate Functions

Global functions are provided for non-componental operations on vectors and matrices. The following example computes the average of the components of a vector of floating point numbers:

```
FloatV    x(2000);
FloatS    average;
average = sum(x) / x.size();// average value of x
```

Member functions are also provided for certain functions and are analogous to assignment operations in that they normally provide a shorthand form in cases where the result of an operation is assigned to the operand. The following example shifts the components of a vector `x` to the left by four places and assigns the result to `x`:

```
FloatV    x(2000);
x.shlp(4);
```

Some functions provide initialization of vectors and matrices as in:

```
ShortV    x(1000);
x.enumerate();
```

which sets successive components of the vector `x` equal to: `0,1,2,3,...,999`. This illustrates the polymorphic nature of the C++ class library.

Boolean vectors and matrices are especially important since they provide control values for conditional parallel processing. Functions are provided to generate various important regular boolean objects, as in the following example which sets up a diagonal matrix mask in the boolean matrix `diag`:

```
BoolM     diag(d,d);// Declare d*d boolean matrix
diag.diagmask();    // Set up diagonal mask in diag
```

Components

Various techniques are available for extracting components and sub-objects from objects, and for conditional processing.

Accessing a Component of a Matrix or Vector

An individual component of a matrix or vector can be accessed using indexing operations in either a left- or right-hand-side context. Subscript expressions are parenthesized following the name of the matrix or vector. Commas separate multiple subscripts. For example:

```
IntV ivec(100);
ivec(0) = ivec(99);// set first component equal to last

CharM     cmat(10,10);
cmat(0,1) = cmat(9,9) - cmat(3,4);
```

Accessing a Vector or Matrix

A similar notation is used to access a vector or matrix from a sequence of vectors or matrices. Two groups of subscripts may be juxtaposed to select individual components of such sequences as in the example:

```
ShortVseq ssq(ShapeSeq(4,4),ShapeV(100));
ssq(0,0) = ssq(3,3)(99);
```

which declares a 4*4 sequence of 100-component vectors and sets all components of the vector `ssq(0,0)` equal to the last component of the vector `ssq(3,3)`.

Sub-objects can also be referenced in both left- and right-hand-side contexts. For example:

```
ShortM    sm(30,30);
sm.row(i) = sm.col(j);
```

sets the components of row `i` of the 30*30 matrix `sm` equal to corresponding components of column `j` of the same matrix.

Conditional Processing and Regions

Conditional parallel operation is achieved by means of the `where` member function. For example:

```
IntM x(100,100);
x.where(x < 0) = -x;// Equivalent to x = abs(x);
x.where(diagmask(x)) = x + 100;
     // modify components on the diagonal of x
```

The `where` member function produces a region whose construction and use may be separated as in the following code which has a similar effect to that above:

```
IntM region negx = x.where(x < 0);
negx = -x;
```

Examples

Partial Sums

This example creates a vector *r* from vector *v* such that component *i* of *r* contains the sum of components *0* to *i* of *v*:

$$r_i = \sum_{j=0}^{i} v_j \qquad i = 0, \ \dots, \ n-1$$

A parallel algorithm to perform this computation sums pairs of numbers, then pairs of pairs and so on. It can be coded as:

```
IntV ScanAdd(const IntV& v)
{
     Int  n = v.size();
     IntV r(v.shape());
     r = v;
     for (Int i = 1; i < n; i += i)
          r += shrp(r,i);
     return r;
}
```

The function `ScanAdd` returns its result in a vector of integer values. A local variable `n`, of type `Int`, is declared to hold the number of components of the input vector, which is returned by the function `size`. A local vector, `r`, is then declared to have the same number of components (shape) as the input vector, `v`, and `r` is initialized to the input values. The partial sums are accumulated in $log_2 n$ steps. The function `shrp` moves the data to the right (higher index values) by `i` places and shifts zeros in from the left-hand-side. These zero values have no effect on the partial sums that have been completed.

This code may be generalized to any type of numeric vector using a C++ template:

```
template<class T> T ScanAdd(const T& v)
{
     Int  n = v.size();
     T    r(v.shape());
     r = v;
     for (Int i = 1; i < n; i += i)
          r += shrp(r,i);
     return r;
}
```

Simple Laplace

This example shows how the `where` construct can be used to control activity. Iteration is used to find an approximate solution of the Laplace equation within a region defined by a boolean matrix mask.

```
void Laplace(FloatM& f,const BoolM& mask,Float epsilon)
{
     FloatM    oldf(f.shape());
     do
     {
          oldf = f;
          f.where(mask) = 0.25 * (shnp(f)+shsp(f)
                                   +shep(f)+shwp(f));
     } while (any(abs(f-oldf) >epsilon));
}
```

The `where` construct has the effect that those values of `f` within a region defined by the *true* (non-zero) values in `mask` are set to the average of their four nearest neighbors, and that the values not in this region are left unchanged. The local matrix `oldf` is used to hold the values of the previous iteration. It has the same dimensions (shape) as the input matrix `f`. The iteration stops when the solution has converged, that is when all the changes in values of `f` are less than the scalar value `epsilon`. The global function `any` returns the scalar value true if one or more of its input values is true; it returns false if all the input values are false.

Matrix Multiplication

The following code multiplies two matrices by summing outer products of the columns of the one matrix with the rows of the other. It illustrates the use of aggregate functions to select sub-regions of matrices and to create new matrices by replicating values.

If A is an k by m matrix and B is an m by n matrix then the result R is an k by n matrix.

```
FloatM Mmpy(const FloatM& A,const FloatM& B)
{
     Int  k = A.size(0);
     Int  m = A.size(1);
     Int  n = B.size(1);
     FloatM    R(k,n);
     R = 0.0;
     for (Int i = 0; i < m; ++i)
          R += matc(A.col(i), n)*matr(B.row(i), k);
     return R;
}
```

The member function `col` extracts the `k` values forming column *i* from the matrix. These are passed to the global function `matc` which returns an `k` by `n` matrix having *n* copies of the input column. The functions `row` and `matr` work similarly on rows.

Bubble Sort

The following code sorts a set of series of numbers using what is termed “outer loop parallelism”. The series are stored in a sequence of vectors. Each vector in the sequence holds one value from every one of the series to be sorted. Thus, a single series is traced by stepping through the sequence with the same vector location.

The AP Class Library copy constructors return a reference to the source object. This is used in the code below as shorthand for indexing the vector sequence.

```
void Sort(IntVseq& vs)
{
     Int  n = vs.sizeSeq();
     IntV aa(vs.shapeV());
     for (Int j = 1; j < n; ++j)
          for (Int i = 0; i < n-j; ++i)
          {
               IntV a = vs(i);
               IntV b = vs(i+1);
               BoolV mask = (a >= b);
               aa = a;
               a.where(mask) = b;
               b.where(mask) = aa;
          }
}
```

The vectors `a` and `b` are references to consecutive vectors in the sequence and progress through the sequence with iterations of the inner loop. The boolean vector `mask` is set to indicate which components need exchanging. The `where` construct only exchanges the out of order values indicated by `mask`.

Fortran-Plus

Fortran-Plus is a dialect of Fortran 77, suitably extended for the Gamma II^{Plus}; many of its features influenced the array extensions of the Fortran 90 language. Fortran-Plus lets the programmer manipulate whole arrays as single objects. The language gives the programmer a clear abstraction of the Gamma II^{Plus}.

The most important feature of Fortran-Plus is the ability to manipulate complete one- and two-dimensional data structures — vectors and matrices. The language provides a comprehensive environment for the manipulation of such objects, including the following operations:

- manipulating complete objects without using any loop constructs (assignment expressions)
- acting conditionally on only part of complete objects (indexing)
- carrying out special operations, such as summing the components of a matrix or vector (aggregate functions)

A program designed to run on the Gamma II^{Plus} using Fortran-Plus has two parts. The first part is the Gamma II^{Plus} program, developed using the Fortran-Plus tools on a workstation and downloaded for execution on the Gamma II^{Plus}. The second part is a host program, created with the usual workstation development tools (for example Fortran or C). The host program invokes the Gamma II^{Plus} program through function or subroutine calls.

Data Modes

The parallel processing power of the Gamma II^{Plus} is accessed from Fortran-Plus by the use of parallel data *modes*.

Matrices and vectors are two of the data modes provided by Fortran-Plus. The third data mode is equivalent to ordinary data in Fortran 77, and is known as scalar mode. All data and expressions in Fortran-Plus fall into one of these three modes.

Vector and Matrix Modes

Data items in a vector or matrix mode object are distributed over the Gamma II^{Plus} PEs to allow them to be processed in parallel. If the size of a vector exceeds the number of PEs or either of the dimensions of a matrix exceeds the edge size of the Gamma II^{Plus} array, the compiler automatically maps the object onto the Gamma II^{Plus} array so that some PEs will contain more than one component. In general, the programmer need not be concerned with how this mapping is done.

Declaring Vectors and Matrices

Vectors and matrices are declared in a similar way to one- and two-dimensional arrays in Fortran 77, except that the first or both dimension subscripts are preceded by an asterisk. A dimension

declared in this way is called a parallel dimension and all operations on such dimension(s) are performed in parallel. Arrays of vectors and matrices can be declared by using additional subscripts in the normal Fortran manner. Typical Fortran-Plus declarations are:

```
INTEGER matrix(*164,*164)
REAL vector(*100)
REAL vector_array(*164,16,16)
INTEGER matrix_array(*50,*50,20)
```

Here, `vector_array` is a 16×16 array of 164-component vectors, and `matrix_array` is an array of twenty 50×50 matrices.

Data Type and Length

Fortran-Plus has the following kinds of data type:

- logical
- character
- integer
- unsigned integer
- real

Performance

As a Gamma II^{Plus} PE is bit or byte oriented, the time taken to perform an operation on given data depends on the number of bits or bytes used to represent that data. The compiler determines if a given operation will be implemented by the 1-bit processors or the 8-bit processors or a combination of the two.

Range of Data Lengths

To provide for flexibility and efficient execution, Fortran-Plus provides a wide range of data-lengths for integer and real data types. For example, floating point data is provided not only in the two lengths of Fortran 77 (“real” and “double precision”), but in any number of bytes ranging from three to eight. Similarly, integer data can occupy any number of bytes from one to eight. In Fortran-Plus (in matrix or vector mode) a component of a logical data item occupies only one bit, making operations on logical data very fast.

The data-length of a data item is declared as part of the type; a typical declaration is:

```
REAL*5 x(*23,*46)
```

which declares a matrix `x`, each of whose components is a floating-point number stored in five bytes. The storage required for vectors and matrices is proportional to data-length.

Expressions

The type and mode of a Fortran-Plus expression depends on the type and mode of its sub-expressions. The rules for the type of the

result of an expression are similar to the rules in Fortran 77; for example, a combination of real and integer gives a real result. Similarly, the rule for the data-length of an expression is that the result has the data-length of the longer of the two sub-expressions.

The mode of an expression composed of sub-expressions of different modes is determined as follows:

Scalar and Vector or Matrix

The scalar is replicated to give a vector or matrix with identical components. Thus:

```
INTEGER my_scalar, my_vector(*50)
INTEGER result(*50)
.
.
result = my_vector*my_scalar
```

Each of the 50 components of `result` is the product of the corresponding component of `my_vector` with `my_scalar`.

Vector and Vector (or Matrix and Matrix)

The objects must have the same “shape” - that is, have the same number(s) of components along their parallel dimension(s).

```
INTEGER my_scalar, v1(*51), v2(*51)
INTEGER result(*51)
INTEGER*2 matrix1(*51,*24),matrix2(*51,*24)
INTEGER*2 av(*51,*24)
.
.
result = v1*v2 + my_scalar
av = (matrix1 + matrix2)/2
```

Vector and Matrix

Simple combination of these two modes is an error. However, there are intrinsic functions available to form a matrix by the replication of a vector by columns or rows. For example, the following code uses the declarations above to create a matrix with each of its columns the same as the vector:

```
matrix1 = MATC(v1,24)
```

Indexing and Assignment

Fortran-Plus contains indexing mechanisms to select sub-parts of an object for processing. These mechanisms are significant contributions to the expressive power of Fortran-Plus programs. Indexing can be applied:

- to an expression (right-hand-side indexing), where it generally returns a value of a different mode from that of the original expression
- on the left-hand-side of an assignment, where it indicates which components of an object will take part in the assignment

Right-Hand-Side Indexing

Right-hand-side indexing selects an object of one mode from an object of another mode. We can select:

- a scalar from a scalar array, vector, vector array, matrix or matrix array
- a vector from a vector array, matrix or matrix array
- a matrix from a matrix array

For example, simple selection of a vector from a matrix is specified by a null subscript (indicating “select all components”) in the appropriate parallel subscript position:

```
INTEGER   my_matrix(*200,*150)
INTEGER   my_vector(*150)
.
.
my_vector = my_matrix(13,)
```

`my_matrix(13,)` selects the 13th row of `my_matrix`, a 150-component vector. Similarly, `my_matrix(,13)` would give a 200-component vector, the 13th column of `my_matrix`.

Null subscripts in both the parallel subscript positions selects a matrix from a matrix array:

```
INTEGER matrix_array(*150,*200,5,7)
INTEGER my_matrix(*150,*200)
.
.
my_matrix = matrix_array(,,3,1)
```

Fortran-Plus supports many more advanced indexing constructs.

Left-Hand-Side Indexing

The variety of indexing constructs already described can also be used on vectors or matrices on the left-hand-side of an assignment statement. They are then interpreted as specifying which components are to be updated.

Of particular importance is an indexing method where a logical matrix, a logical vector, or a logical expression is used to mask assignment to a conforming (that is, same shape) matrix or vector. Every component of the matrix or vector that corresponds to a true value in the mask is updated. The hardware of the Gamma II^Plus, particularly its activity control, makes this very efficient.

```
INTEGER my_matrix(*500,*500)
.
.
my_matrix(my_matrix.LT.0) = my_matrix + 5
```

Here, `my_matrix.LT.0`, which is a logical matrix expression, is used as a mask for the assignment of `my_matrix+5` to `my_matrix`. The effect of this is to add 5 to the negative components of `my_matrix`.

The Fortran-Plus code shown above has the same effect as the following standard Fortran 77 code.

```
      INTEGER MY_MATRIX(500,500)
      .
      .
      DO 20 J=1,500
      DO 10 I=1,500
           IF (MY_MATRIX(I,J).LT.0) THEN
                MY_MATRIX(I,J) = MY_MATRIX(I,J)+5
           ENDIF
10    CONTINUE
20    CONTINUE
```

Subroutines and Functions

Routine Arguments

Matrices and vectors can be passed as arguments to subroutines and functions. There is a syntax to specify that a formal matrix or vector argument is of assumed size (taking dimensions from the actual argument), as in the following example:

```
      SUBROUTINE Dist(dx,dy,d)
C     Sets d equal to square root of the
C     sum of squares of dx and dy

C     IN
           REAL dx(*:,*:), dy(*:,*:)
C     OUT
           REAL d(*:,*:)
C     BEGIN
           d = SQRT((dx*dx)+(dy*dy))
           RETURN
      END
```

Alternatively, the extent of an array dimension can be passed as argument to a routine, and the formal argument then used in the parallel subscripts. This is analogous to the definition of array arguments in Fortran 77.

SIZE Function

The extent of an array dimension can be found by using the built-in intrinsic function SIZE.

Type and Mode of a Function

A user-defined Fortran-Plus function may a return matrix, vector or scalar result. The type, data-length and mode of the result should be given both when the function is defined and when it is declared in a subprogram calling that function. An example of the latter is:

```
      EXTERNAL FUNCTION CubeRoot
      REAL CubeRoot(*:,*:)
```

Here, the size of the matrix returned is dependent on its arguments, so the assumed size syntax is used.

Recursion

Subroutines in Fortran-Plus can make calls to themselves (that is, they can be recursive), unlike subroutines in Fortran 77. Fortran-Plus functions may also be indirectly recursive (invoke themselves via intermediary routines).

Built-in Functions

Fortran-Plus provides a wide range of built-in functions. These include many of the intrinsics of Fortran 77, which in Fortran-Plus are extended to work with matrix, vector and scalar data. Built-in functions also include many operations that are appropriate only to parallel data objects.

Error Handling

Programming Error Checks

Fortran-Plus checks at run time for a number of problems that can arise from programming errors. These checks include testing that:

- the shapes of objects in expressions conform
- DO loop parameters are valid
- array subscript ranges are valid
- the arguments of subroutines and functions match their declarations

This feature can be turned off in order to enable the highest performance of correct code.

If a Programming Error is Detected

If an error is detected by the run-time system, control is passed to the diagnostics system. By default, the diagnostics system invokes the interactive debugger, but this action can be changed so that the program does one of the following:

- aborts
- reports error and continues
- produces a dump of the variables and then continues

Computational Errors

In addition, there are sophisticated facilities for handling computational errors. If a computational error occurs, then, by default, control is passed to the run-time diagnostics system. However, Fortran-Plus lets the programmer maintain close control at run-time over which computational errors cause interrupts. Interrupts can be suppressed for certain sections of code and re-instated for others. The occurrence of computational errors can be recorded, even if interrupts are suppressed, for later testing. These

options can be applied globally to all variables, or limited to nominated parts of vectors or matrices.

Examples

This section contains Fortran-Plus versions of the C++ examples presented earlier (see “Examples” on page 3-8).

Partial Sums

The effect of the `shape` member function is achieved by declaring the input to be of assumed size. The C++ `for` loop is implemented by means of an `IF` statement and a `GOTO` statement.

```
     FUNCTION ScanAdd(v)
     INTEGER ScanAdd(*SIZE(v))
C    IN
          INTEGER v(*:)
C    LOCALS
          INTEGER i, n
C    BEGIN
          n = SIZE(v)
          ScanAdd = v
          i = 1
10        IF (i.LT.n) THEN
                    ScanAdd = ScanAdd + SHRP(ScanAdd,i)
                    i = i + i
                    GOTO 10
          ENDIF
          RETURN
     END
```

Simple Laplace

The effect of the `where` construct used in the C++ code is achieved by means of masked assignment in Fortran-Plus.

```
     SUBROUTINE Laplace(f, mask, epsilon)

C    INOUT
          REAL f(*:,*:)
C    IN
          LOGICAL mask(*:,*:)
          REAL epsilon
C    LOCALS
          REAL oldf(*SIZE(f,1),*SIZE(f,2))
C    BEGIN
10        CONTINUE
                    oldf = f
                    f(mask) = 0.25*(SHNP(f)+SHSP(f)+
     &                                  SHEP(f)+SHWP(f))
          IF (ANY(ABS(f-oldf).GT.epsilon)) GOTO 10
               RETURN
     END
```

Matrix Multiplication

The C++ `row` and `col` member functions are replaced by suitable indexing.

```
      FUNCTION Mmpy(a,b)
      REAL Mmpy(*SIZE(a,1),*SIZE(b,2))

C     IN
           REAL a(*:,*:)
           REAL b(*SIZE(a,2),*:)
C     LOCALS
           INTEGER k, m, n, i
C     BEGIN
           k = SIZE(a,1)
           m = SIZE(a,2)
           n = SIZE(b,2)
           Mmpy = 0.0
           DO 10 i = 1, m
           Mmpy = Mmpy + MATC(a(,i),n)*MATR(b(i,),k)
10         CONTINUE
           RETURN
      END
```

Bubble Sort

Masked assignment is used in place of the `where` construct and the data swapping requires explicit reference to vs.

```
      SUBROUTINE Sort(vs)

C     INOUT
           INTEGER vs(*:,:)
C     LOCALS
           INTEGER aa(*SIZE(vs,1))
           INTEGER i, j, n
           LOGICAL mask(*SIZE(vs,1))
C     BEGIN
           n = SIZE(vs,2)
           j = 1
10         IF (j.LT.n) THEN
                     DO 20 i = 1, n-j
                          mask = vs(,i).GT.vs(,i+1)
                          aa = vs(,i)
                          vs(mask,i) = vs(,i+1)
                          vs(mask,i+1) = aa
20                   CONTINUE
                     j = j + 1
                     GOTO 10
           ENDIF
           RETURN
      END
```

Assembly Languages

For specialized applications, and where highest performance is required, it may be appropriate to use low-level programming for part of the application. However, this is only possible when working in a Fortran-Plus, not a C++, environment.

Array Processor Assembly Language

APAL can be used to program the MCU and the 1-bit processors. The APAL Manual describes both the language itself, and how to interface subroutines and functions written in APAL to other parts of the program written in Fortran-Plus. (See also “Assembly Language for 1-Bit Processors” on page 2-6 and “Documentation Map” at the back of this document).

Coprocessor Assembly Language

CPAL can be used to program the 8-bit processors. CPAL code is written as a set of intrinsics that are initiated from APAL code, and is described in the CPAL manual. (See also “Assembly Language for 8-Bit Processors” on page 2-9, and “Documentation Map” at the back of this document.)

Language Facilities

Each language incorporates macro, substitution, and conditional assembly facilities, which help to reduce the complexity of the detailed low-level programming used in such cases.

Program Development

CPP supplies a range of facilities exist to help the programmer:

- C++ class library and support libraries
- a complete Fortran-Plus environment, including debugger and support libraries

“Summary” on page 3-3 contains a table which illustrates this fully.

Multiple Users Possible

The Gamma II^{Plus} hardware and software environment gives multiple users on a network simultaneous access to the system via multi-programming, and each user can run several jobs at one time.

Provided that there is enough array memory and code store in the Gamma II^{Plus}, up to 20 jobs can be resident at the same time. Each job is given a time slice allocation on a priority basis; both priority and length of slice can be monitored and controlled by the system manager.

Program Timing

Counters maintain statistics of the clock cycles used by each job, and of the clock cycles each job is resident in the Gamma II^{Plus}. Users can access these counters by calling system subroutines, and so find out the elapsed time and running time of their own job(s) independent of the presence of other jobs in the system. The counters can also be accessed via PSAM (see “Fortran-Plus Environment” below).

C++ Environment

C++ programs are developed using the standard C++ development environment. This environment, together with the CPP-supplied member functions for printing the values of parallel objects, provide a full set of state-of-the-art development facilities.

Fortran-Plus Environment

Fortran-Plus programs are developed using CPP’s proven compiler and interactive debugging system.

Interactive Debugging with PSAM

The Gamma II^{Plus} development environment for Fortran-Plus includes PSAM (Program State Analysis Mode), a powerful interactive debugging system for Fortran-Plus and APAL. By default, PSAM is entered whenever the Gamma II^{Plus} program halts due either to a run-time error, or to executing a breakpoint inserted by the programmer.

PSAM can be controlled interactively via a command line interpreter or from a command file, with redirection of output as required. A screen mode option is available under VMS. The names of the commands have been selected to be appropriate to the particular host development environment (UNIX or VMS), but

under UNIX there is also an alias facility for creating alternative command names. There is a help command which gives descriptions of all the commands.

PSAM Features

Features of PSAM include:

- **Examination of Variables**. Selected variables, including all or parts of matrices and vectors, can be examined. Variable names used for selection can contain wildcard characters. Variables can be displayed in several different formats.
- **Breakpoints.** Breakpoints can be set or changed on executable Fortran-Plus and APAL source statements or instructions while program execution is halted, or just before a program is entered. A Fortran-Plus statement is specified by its line number in the current source file or by a procedure name; an APAL instruction is specified by its address offset, or by its code section name. Breakpoints can be disabled and enabled, completely cleared or listed.
- **Program Control**. A program can be continued or abandoned from PSAM. There are also commands which continue for a specified number of source statements allowing the user to step through the program. Called procedures may be stepped through locally or may be treated as single statements.
- **Source Code Access.** Access to the original Fortran-Plus source is available, and the current line or any selection of lines from the current file can be listed. Access to APAL source is also available.
- **Stack Examination.** A stack backtrack (that is the procedure call sequence) can be displayed automatically when a program enters PSAM, or by explicit command. It is also possible to traverse the stack from one procedure to another to examine variables.

Fortran-Plus and APAL Trace Facilities

Fortran-Plus and APAL have language-defined trace facilities which output the values of specified data items to the host computer at run-time. There are five levels of trace, and a compile-time switch specifies which levels are to be compiled.

Application Support Libraries

CPP provides a number of libraries to support application development on the Gamma II^{Plus}. The design and coding of these libraries is the result of many man-years of research and development by CPP. The libraries combine ease of use with high performance, and are available for both C++ and Fortran-Plus. (For an overview of CPP publications covering these libraries, see the Documentation Map at the back of this manual.)

General Support Library

The General Support Library is designed to aid rapid application development on the Gamma II^{Plus} series. It provides a wide range of utility routines, including routines for:

- irregular access to data; indirect indexing through gather and scatter routines
- access to mathematical constants and to constants of the data representation (for example, maximum and minimum representable values)
- packing and unpacking data
- random number generation
- sorting
- values of mathematical functions, (for example hyperbolic sines and cosine, Bessel functions and so on)

These routines generally support data arrays of a range of data-lengths and arbitrary dimension.

Image Processing Library

The Image Processing Library provides implementations of the major building blocks used in image processing applications including convolutions (both specialized and general purpose), image analysis and manipulation, and loading and storing of data.

Coupled with the processing features of the Gamma II^{Plus} programming languages, which support low-level operations (for example thresholding) on image-like objects, the Image Processing Library facilitates the development of high-performance solutions to image processing problems.

Routines in this library exploit the parallelism inherent in pixel-based operations by representing an image as a matrix mode variable. As a result, the image dimensions can be arbitrary and the pixel data can be represented by any of the data types supported by the high-level language.

The functions that the Image Processing Library provides include:

- convolution, including optimized average, Kirsch, line detection, Prewitt, Roberts and Sobel masks
- dilate, erode, fill and skeletonize

- file transfer
- histogramming
- high-level analysis, classification, labelling, segmentation
- median filtering
- peak detection, box-in-box and difference-of-Gaussian
- range adjustment
- spatial transforms, rotate and zoom
- thresholding

Transforms Library

The Transforms Library provides a range of fast mathematical transform routines, including Fast Fourier Transforms (both real and complex), Discrete Cosine Transforms and Hadamard Transforms.

Routines in this library provide forward and inverse transforms with a choice of data orderings on both input and output. One- and two-dimensional versions are provided for all the transform types. Transforms are provided for all C++ or Fortran-Plus numerical data types and data-lengths. Implementations of the transforms are tailored to the parallelism of the input data so the user can choose between performing one transform at a time or, for higher performance, many transforms simultaneously on independent datasets.

Linear Algebra Library

The Linear Algebra Library is a set of routines for common operations on matrices. These include matrix multiplication, inversion and solution (dense and tri-diagonal), as well as the solution of over-determined systems. Most operations are performed on floating point data, at all data-lengths accepted by the high-level language. The matrices are represented as matrix mode variables.

Graphics Library

The Graphics Library allows on-screen display of data held in the array memory of the Gamma II^Plus. Library routines allow nomination of the area to be displayed, control of the display update frequency (discrete or continuous) and control of the interpretation of the pixel values (setting of color look-up-tables). In addition to the low-level control of the display characteristics, the Graphics Library also provides drawing primitives to produce lines, rectangles and characters on the screen and to merge or swap images.

Parallel Data Transforms

When performing parallel computations such as those offered by the Gamma II^{Plus} system, it is necessary to map data sets appropriately so that computations may be applied between them, and it may be appropriate to change the mapping of a data set during the course of the computation.

Re-mappings such as shift and transpose are provided within the C++ Class Library and within the Fortran-Plus language. Other classes of data re-mapping arise within routines such as the Fast Fourier Transform and sorting, where typically two halves of a data set (or two halves of part of a data set) are exchanged during the course of the algorithm. Such cases involve bulk data movement, but the re-organization is performed according to a regular pattern.

Parallel Data Transforms (PDTs) consist of both a compact notation for describing regular data mappings, and a set of CPP-supplied software that can be used to rearrange data between two data mappings described in that notation. PDT facilities are available both within the C++ Class Library and the Fortran Plus language.

Data mappings can either be constant (known at compile time) or variable (constructed or changed during the running of the program). When you specify that data is to be re-ordered from one mapping to another, the PDT software invokes code to perform that re-mapping, making efficient use of the hardware capabilities of the processor array. When both source and destination mappings are known at compile time, the compilation system may analyze several strategies for performing the re-ordering before choosing the best one to use.

Index

Numerics

1-bit processor . . . 2-4
  A register . . . 2-4
  assembly language . . . 2-6
  C register . . . 2-4
  D register . . . 2-4
  memory transfers . . . 2-10
  Q register . . . 2-4, 2-5
  S register . . . 2-5
  schematic diagram . . . 2-5
8-bit processor . . . 2-4, 2-7, 2-9
  A register . . . 2-7
  assembly languages . . . 2-9
  D register . . . 2-7
  internal memory . . . 2-7
  M register . . . 2-8
  Q register . . . 2-8
  S register . . . 2-8
  schematic diagram . . . 2-8

A

accumulator *see* Q register
activity control . . . 2-4, 3-14
address of Cambrige Parallel Processing . . . 1-2
Address Sequencing Engine (ASE) . . . 2-20
aggregate function . . . 3-6
AP Class Library . . . 3-4
  copy constructor . . . 3-10
APAL
  code store . . . 2-11
  see also Array Processor Assembly Language
application support library . . . 3-2
array
  data bus . . . 2-13
  interface . . . 2-13
  memory . . . 2-11
Array Processor Assembly Language (APAL) . . . 2-6, 3-2, 3-19
Array Support Unit (ASU) . . . 2-12
ASE *see* Address Sequencing Engine
assignment . . . 3-13
ASU *see* Array Support Unit

B

backplane 2-15
Bit3 VME host configuration 2-17
broadcast 2-5
bubble sort 3-10, 3-18

C

C++ 3-2, 3-4
- AP Class Library 3-2
- conditional processing 3-8
- constructor mechanism 3-5
- data type 3-4
- destructor mechanism 3-5
- development environment 3-20
- matrix 3-4
- object declaration 3-5
- shape of objects 3-5
- vector 3-4

Cambridge Parallel Processing
- address of 1-2
- information about 1-1

classes
- scalar 3-4

clock cycles 3-20
code store 2-11
conditional processing with C++ 3-8
constructor mechanism for C++ 3-5
Coprocessor Assembly Language (CPAL) 3-2, 3-19
- intrinsics 2-9

copy constructor in AP Class Library 3-10
corner turning 2-20
CPAL
- Sequencer Unit (CSU) 2-13
- *see also* Coprocessor Assembly Language (CPAL)

CSU *see* CPAL Sequencer Unit
customer devices, links to 2-20

D

data mapping 2-10
data mode 3-11
data paths
- Input/Output 2-20
- row and column 2-5

data representation
- horizontal mode 2-10
- software defined 2-10
- vertical mode 2-10

data transfer
- fast input and output 2-18

data types 3-12
- C++ 3-4
- Fortran-Plus 3-12

data-length 3-12

DCU *see* Diagnostic Control Unit

debugger 3-2, 3-20, 3-21

declaring
- local AP objects 3-5
- vectors and matrices in heap 3-5

declaring objects with C++ 3-5

destructor mechanism for C++ 3-5

Diagnostic Control Unit (DCU) 2-14

Discrete Cosine Transforms 3-23

DO loop 2-11

D-Plane 2-18, 2-19
- performance 2-19

dynamic sizing of AP objects 3-5

E

EAL *see* External Adaptor Link

EAU *see* External Adaptor Unit

edge register 2-13

error handling, Fortran-Plus 3-16

Ethernet 2-16

examples
- C++ 3-8
- Fortran-Plus 3-17

expressions, Fortran-Plus 3-12

External Adaptor Link (EAL) 2-20

External Adaptor Unit (EAU) 2-15, 2-20, 2-22
- for PCI 2-22
- for VME/VSB 2-22

external device 2-15

F

fast disk 2-15

Fast Fourier Transforms 3-23

fast input and output 2-18
- external connection 2-19

Fast Input/Output Controller 2-3, 2-19

Fortran 77 3-2

Fortran 90 3-2

Fortran-Plus ... 3-2, 3-11
data lengths ... 3-12
data mode ... 3-11
data structures ... 3-11
data types ... 3-12
error handling ... 3-16
expressions ... 3-12
functions ... 3-15
indexing ... 3-13, 3-14
scalar broadcast ... 3-13
subroutines ... 3-15

functions
C++ aggregate functions ... 3-6
C++ componental functions ... 3-6
Fortran-Plus ... 3-15
Fortran-Plus built-in ... 3-16
member ... 3-6

functions, Fortran-Plus ... 3-15

G

Gamma II^{Plus}
application areas ... 1-1
backplane ... 2-15
main components ... 2-2
product line ... 1-1
system diagram ... 2-2

Gamma Input/Output Controller ... 2-18, 2-20
Address Sequencing Engines ... 2-20
architecture ... 2-20
configurations ... 2-22
Data Path ... 2-20
D-plane interface ... 2-22
performance ... 2-22
schematic diagram ... 2-21
software for ... 2-22

General Support Library ... 3-2, 3-22

GIOC *see* Gamma Input/Output Controller

global function ... 3-10

GRALIB *see* Graphics Library

Graphics Library ... 3-23

GSLIB *see* General Support Library

H

Hadamard Transforms ... 3-23

horizontal mode ... 2-10

host
- Bit3 VME ... 2-17
- interface ... 2-16
- PCI bus ... 2-17
- Q-bus ... 2-17
- reasons for connection to ... 2-16
- SBUS ... 2-17
- Sun ... 2-17
- using external workstation ... 2-17
- using internal microprocessor ... 2-16
- using Sun workstation ... 2-17
- using VAX workstation ... 2-17
- VAX ... 2-17

I

IKON VME ... 2-17

Image Processing Library ... 3-3, 3-22

indexing ... 3-13
- C++ ... 3-7
- Fortran-Plus ... 3-14

intrinsics
- CPAL ... 2-9

IPLIB *see* Image Processing Library

L

LALIB *see* Linear Algebra Library

languages
- C++ ... 3-4
- C++ and Fortran-Plus ... 3-2
- object-oriented ... 3-4

Laplace ... 3-17

library ... 3-2, 3-22
- AP Class ... 3-4
- General Support ... 3-2, 3-22
- Graphics ... 3-23
- Image Processing ... 3-3, 3-22
- Linear Algebra ... 3-3, 3-23
- Transforms ... 3-3, 3-23

Linear Algebra Library ... 3-3, 3-23

local AP objects, declaring ... 3-5

low-level programming ... 3-19

M

Master Control Chip (MCC) ... 2-11

Master Control Unit (MCU) ... 2-2, 2-11
- register ... 2-11
- schematic diagram ... 2-12

matrix 3-4, 3-11
as argument 3-15
initialization 3-6
multiplication 3-18
operation on 3-6
shape 3-5
MCU *see* Master Control Unit
member function 3-6, 3-10
memory 2-11
microcode store 2-11, 2-13
microprocessor controller board 2-15
multi-programming 3-20

O

object declaration with C++ 3-5
object-oriented language 3-4
objects 3-5

P

Parallel Data Transform (PDT) 3-3, 3-24
PCI bus host configuration 2-17
PCI External Adaptor Unit 2-22
PDT *see* Parallel Data Transform
PE *see* Processor Element
performance 2-9
general 1-1
of D-Plane 2-19
of Gamma Input/Output Controller 2-22
Performance Technology PT151E 2-16
pipelining 2-9
plane 2-4
processor
1-bit 2-4
8-bit 2-7, 2-9
Processor Element (PE) 2-2, 2-4, 3-4
connections between 2-5
schematic diagram of 2-6
product line for Gamma II^{Plus} 1-1
program development 3-20
Program State Analysis Mode (PSAM) 3-2, 3-20
features 3-21
program timing 3-20
PSAM *see* Program State Analysis Mode

Q

Q-bus host configuration 2-17

R

Radstone PPC 603e . . . 2-16

register
- file . . . 2-11
- names . . . 2-7
- planes . . . 2-4
- shifts . . . 2-5

registers
- A register . . . 2-4, 2-7
- C register . . . 2-4
- D register . . . 2-4, 2-7
- M register . . . 2-8
- Q register . . . 2-4, 2-5, 2-8
- S register . . . 2-5, 2-8

rows, sequence of . . . 2-10

S

SBus host configuration . . . 2-17

scalar . . . 3-11, 3-13, 3-14
- broadcast, Fortran-Plus . . . 3-13
- class . . . 3-4

Scalar Coprocessor (SCP) . . . 2-12

SCP *see* Scalar Coprocessor

shape . . . 3-9, 3-13
- of vectors and matrices . . . 3-5

shape of objects . . . 3-5

shift register . . . 2-18

SIZE function . . . 3-15

store planes . . . 2-10

subroutines, Fortran-Plus . . . 3-15

Sun host . . . 2-17

T

timing of programs . . . 3-20

trace facility . . . 3-21

Transforms Library . . . 3-3, 3-23

TRLIB *see* Transforms Library

V

VAX host . . . 2-17

vector . . . 3-11
- as argument . . . 3-15
- classes in C++ . . . 3-4
- initialization . . . 3-6
- operation on . . . 3-6
- shape . . . 3-5

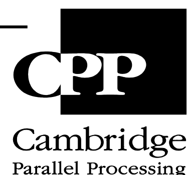
vectors in C++ . . . 3-4

vertical mode 2-10
vertical storage, matrix and vector 3-11
VME 2-17
    microprocessor board 2-19
    VME/VSB External Adaptor Unit 2-22
    VMEbus 2-3, 2-15
    VMEbus interface 2-13
    VxWorks 2-16

W

where construct 3-9
workstation, using as host 2-17

Documentation Map



Section	Document		
Languages	Introduction to Fortran-Plus	Fortran	
	Fortran-Plus	Fortran	
	APAL		
	CPAL		
Libraries	AP Class		C++
	General Support	Fortran	C++
	Image Processing	Fortran	C++
	Graphics	Fortran	C++
	Linear Algebra	Fortran	C++
	Transforms	Fortran	C++
	Parallel Data Transforms	Fortran	C++
	Fast Disk	Fortran	C++
System	Program Development	UNIX	VMS
	System Management		VMS
	System Calls	Fortran	
	Run-Time Messages		VMS
General	Gamma II Technical Overview		
Hardware	Engineering Test Software		
	Writing Device Drivers	UNIX	VMS
	Gamma II Installation		

Your Comments Invited

We at Cambridge Parallel Processing welcome our readers' views on the content of our publications. Inquiries and comments should be directed to either of the addresses below:

Publications Manager	Publications Manager
European Operations	Worldwide Headquarters
Cambridge Parallel Processing Ltd.	Cambridge Parallel Processing
Centennial Court	16841 Armstrong Avenue
Easthampstead Road	Irvine
Bracknell, Berks. RG12 1YQ, UK	CA 92606, USA
Tel: (01344) 861024	Tel: (949) 724-8300
Fax: (01344) 305544	Fax: (949) 724-8399
e-mail: info@cppuk.co.uk	e-mail: info@cppus.com