ICL DAP (2)

Customers as at 31 March 1990

Totals:

74 systems

28 to universities
3 to government laboratories
43 to industry

Breakdown:

Customer	*Application*
UK	
20 to UK Universities/Research Centres	
Bristol University	Molecular physics
Edinburgh University(2)	Molecular biology, physics, neural net
Queen Mary College	Image processing, finite element
Queen Mary College	General Service
Salford University	Engineer applications
Southampton University	Computer design simulation
Kent University	Parallel Processing Centre
Queen's University, Belfast	Parallel computing research
Strathclyde University	Signal processing
SWURCC	Parallel computing
Exeter University	Fluid flow
Reading University	Parallel computing
Cambridge University	Parallel computing
Queen Mary College	DAP training
Imperial Cancer Research Fund	Genetic sequence matching
Aberdeen University	Medical imaging
Manchester University	Medical imaging
London Hospital	Medical imaging
Medical Research Council	Genetic sequence matching
2 to Government Laboratories	
RAL	Circuit simulation, image processing
RSRE	Radar processing
5 to Industrial Companies	
ICL	Defence applications
Plessey	Radar processing
BP	Image processing, neural networks
Hydraulics Research Ltd	Fluid flow
A news agency	Key word searching

USA

5 to USA Universities

Washington University	Image processing
Texas University	Signal processing
RPI	Image processing
Rutgers	Image processing
Old Dominion University	Fluid dynamics

1 to Government Laboratory

Argonne National Laboratory	Parallel computing

34 to Industrial Companies

E-Systems	Radar processing, neural networks
A defence contractor	Radar processing
Apple Computer Inc	Human interface research
Texas Instruments	Image processing
Severn Systems	System Integration
US Army Corps of Engineers	Image processing
NIST	Image processing
DoD	Signal processing
Lockheed	Image processing
STX Systems	Data compression
Ocean Research	Image processing
NOSC	Simulation
FCI	Genetic sequence matching

EUROPE

3 to European Universities

Erlangen University (Gr)	Parallel computing
Brussels University	CAD applications
Dublin University	ECAD

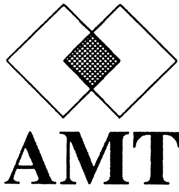
3 to Industry

APTEC (Gr)	Distributor
LERIS (Fr)	Distributor
FTZ (Gr)	Signal procesing

REST OF WORLD

1 to Industry

Furukawa Electric Co Japan	Image processing



DAP SERIES TECHNICAL OVERVIEW

Introducing the DAP/CP8 range

INTRODUCTION

The DAP/CP8 range provides an arithmetic processing capability to work alongside the existing 1-bit processor elements (PEs) on computationally intensive tasks such as floating-point arithmetic and integer multiplication. Logically, each existing PE has a new co-processor associated with it, the two processors being able to work together by accessing the same set of array memory locations. The co-processors (or CP8) have 8-bit wide data paths, thus offering an order of magnitude performance increase over the original 1-bit PEs for suitable operations. Figure 1 shows the DAP/CP8 system with its array of 1-bit processors and array of 8-bit co-processors. The array edge size of the DAP/CP8 model 510C is 32, and for the model 610C is 64.

The new computation facilities can be regarded as an enhancement to the existing capability, and indeed it will be possible to upgrade existing systems, essentially by changing the array boards. The fact that the existing PEs are retained alongside the co-processors ensures compatibility with existing object code, and to take advantage of the co-processors it is necessary only to re-compile the user's Fortran-Plus programs.

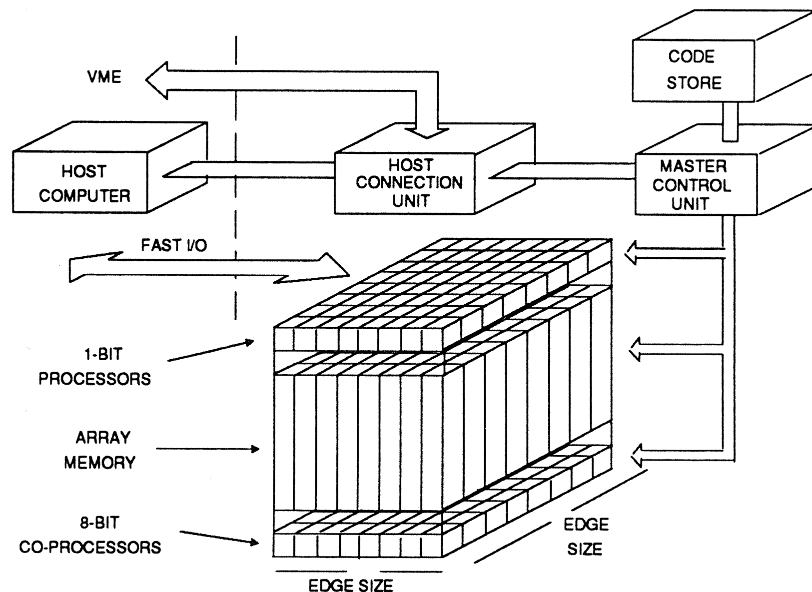


Figure 1 Main components of the DAP/CP8 system. The array edge size is 32 for the model 510C and 64 for the model 610C.

Simple operations such as Boolean work or integer addition are already extremely fast and make full use of the array memory bandwidth. Thus there is no benefit in putting individual operations of this type into the co-processor. Likewise, all data communication other than the memory paths (ie neighbor operations, and row and column data paths) continue to be dealt with by the bit-organized processors. Of course this allocation of function is done automatically by the high-level language compilers, and therefore most users will be unaware of the existence of the co-processors, apart from noticing the higher performance.

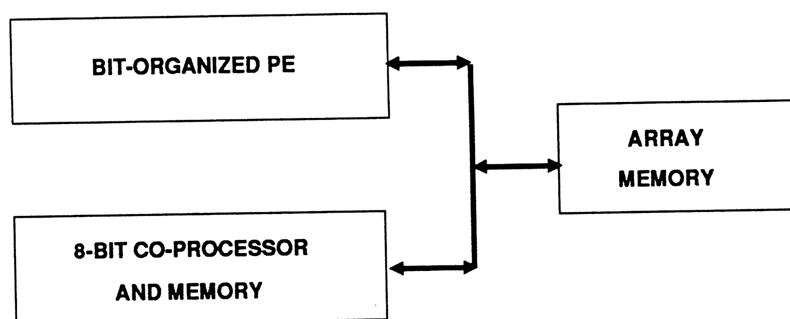


Figure 2 Relationship of the co-processor to the 1-bit PE and array memory.

Figure 2 shows one of the existing PEs and its associated section of array memory, with the 1-bit wide data path connecting them. Each bit-organized PE has a co-processor associated with it and a 1-bit connection to the memory path. Thus, either of the two processors can access or operate on data in the array memory, although there is no direct transfer of data between the two processors. Internal to each co-processor there is a memory of 32 bytes accessed 8 bits wide used for holding operands and workspace for one or two arithmetic operations at normal precisions. Thus the low-level sequence of operations to apply an intrinsic to one DAP-sized array is:

- copy the operand data from the array memory to the co-processor;
- perform the operation in the co-processor memory;
- copy the results back to the array memory.

This is analogous to the operation of the existing software intrinsics, where the operations are performed from array memory to array memory, but in that case a specific region of array memory is designated as a workspace for the intrinsic.

CO-PROCESSOR FUNCTIONS

Although it has been designed with the multiply operation and floating-point work in mind, the co-processor, like the existing 1-bit PEs, offers great flexibility of function. Again there is flexibility of precision with a tradeoff between word length and performance, though in this case the natural step in word length is eight bits.

Figure 3 shows the main components of one co-processor, most of the registers and data paths being 8 bits wide. Although most of the register names are the same as those in the 1-bit PEs, and they have somewhat similar function, they are of course quite distinct from the registers in those PEs.

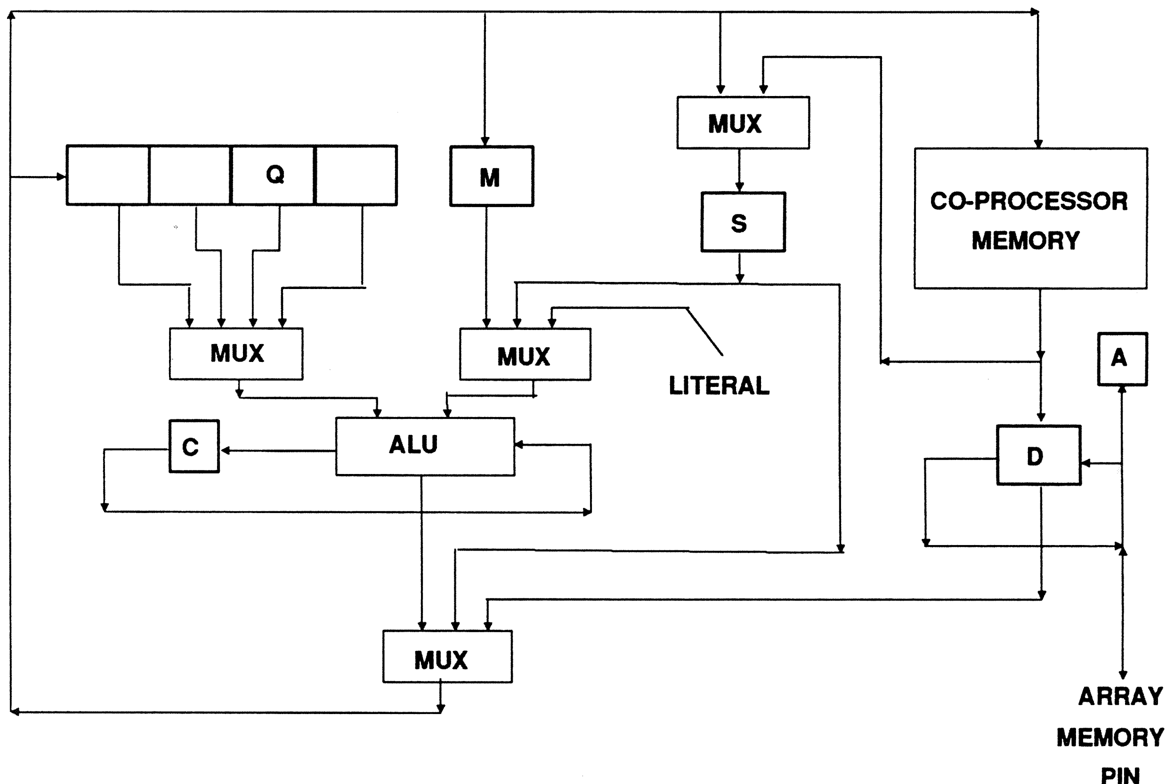


Figure 3 Main components of the 8-bit co-processor (CP8).

At the bottom right of the figure, the D register is used for interfacing to the array memory pin. For input of operands to the co-processor, successive array memory bits are shifted in until D holds a complete byte; that byte is then written to a location in the co-processor memory. For output of results, a byte is loaded into D from the co-processor memory, and then shifted one bit at a time with the data being output to the array memory pin. The single-bit 'A' register may be loaded from the array memory pin

and may be used for activity control in the array memory; the latter is implemented as a *read-merge-write* sequence on each array memory bit in turn.

Shifting of the D register for either input or output may take place concurrently with, and independent of, processing functions in the remainder of the co-processor; it is necessary to steal a cycle from the processing functions only for writing bytes from D to co-processor memory, or for loading D from that memory.

For processing functions, the co-processor memory is again written or read as one byte wide. In the latter case the result is always loaded into the S register, though S is also usable as a general-purpose register.

The Q register acts as an accumulator and is 32 bits wide. Any one of its four bytes may be selected for use as the left operand of the arithmetic-logic-unit (ALU). The complete Q register may 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 may be done unconditionally or may be dependent on a local condition.

The multiplexer shown below the ALU allows local selection of either the ALU output or the contents of the S register, thus giving a form of conditional control. As an implementation detail, this multiplexer is a convenient point to input the D register to the data path in order to write it to co-processor memory.

The M register acts as a general-purpose register and, as well as being able to load the complete register, it may be shifted one bit left or right. The least significant bit of M also has a special function in that it may control the multiplexer on the ALU output, or the shifting of Q.

A multiplexer on the right input of the ALU allows selection between the contents of the S register, the M register or an 8-bit literal value common to all the co-processors.

The ALU provides the usual addition and subtraction operations, and a variety of bit-by-bit Boolean functions. For the arithmetic operations, the carry out may be clocked into the single-bit C register, and used as a carry in for a subsequent instruction, thus allowing multiple byte arithmetic to be implemented.

Much of the performance improvement of the co-processor arises because of the wider data paths, but it is also significant that the Q register with its multiple bytes acts as a specialized form of memory, allowing both ALU operands to be supplied in a single cycle. The ability to shift Q conditionally is especially useful in operations such as floating-point alignment and normalization.

MICROCODE

Code for intrinsics to be run on the co-processors is written in a microcode language specific to the co-processor and, like the APAL language for the 1-bit PEs, offers great flexibility of function. At any given time, all the co-processors are executing the same micro-instruction sequence, so the co-processors may be thought of as a second SIMD array operating alongside the array of 1-bit PEs and communicating with it at high bandwidth via the array memory. Micro-instructions are held in a separate code store and issued by a sequencer which acts under the overall control of the master control unit (MCU). The microcode consists of many short sequences, each corresponding to an intrinsic function, or some other operation such as normalization checks. It is entered at the beginning of a specified microcode sequence which then runs to completion without branching; thus the micro-sequencer is extremely simple.

As a software convention, operands for any such intrinsic are initially in co-processor memory (rather than registers), and results, including exception flags where applicable, are returned to co-processor memory.

Microcode for common intrinsics is loaded at system load time and is shared between all user programs. There is also provision for microcode libraries or microcode specific to a particular application to be loaded along with individual applications programs.

Conceptually, the microcode memory and its associated controller are each a single entity common to the entire array. However, in order to allow co-processors to be added to existing systems with minimal effect on the rest of the system, the microcode is replicated on each array board. The individual micro-sequencers stay exactly in step with one another, although in principle they need not do so.

CONTROL BY THE MCU

Overall control of the co-processors is provided by the MCU via new instructions compiled into the application program. These provide for the starting of the microcode controllers at a particular micro-instruction address, and for testing whether the co-processors are still busy. Other new instructions are provided to copy data between the co-processor D register and the array memory.

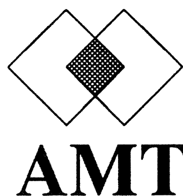
Once the MCU has started co-processor operation it is free to simultaneously perform other operations, including array memory accesses and issuing instructions to the bit-organized processors, until such time as it needs to access the results of the co-processor operation. The most likely case is that, while the co-processors are busy, the MCU will transfer into the co-processor memory the operands for the next intrinsic. Then, when the first intrinsic is finished, the second will be started; concurrently the results of the first intrinsic are copied to array memory and then the operands for the third intrinsic copied into the co-processor. By this means an efficient 'pipeline' is established making good use of both the processing capability and the memory bandwidth.

A common situation where pipelining is easy to organize from the software point of view is when a given intrinsic is applied to arbitrary-sized arrays that are larger than the DAP array dimensions, since in that case the same operation is applied to a succession of DAP-sized arrays. The potential for overlap may be exploited in other situations as the compiler technology is developed further.

BENEFITS

The DAP/CP8 range offers a peak performance improvement of more than an order of magnitude over previous DAP systems for computationally intensive tasks. The Fortran-Plus compiler is well able to exploit this capability and offers outstanding ease of programming.

April 1990



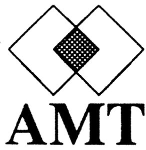
AMT

Active Memory Technology Inc
16802 Aston Street, Suite 103
Irvine
California 92714
U.S.A.

Tel (USA): (714) 261-8901
Fax (USA): (714) 261-8802

Active Memory Technology Ltd
65 Suttons Park Avenue
Reading RG6 1AZ
Berkshire
United Kingdom

Tel (UK): 0734 661111
Fax (UK): 0734 351395



The DAP/CP8 range : model 510C Massively Parallel Computing System

INTRODUCING THE DAP 510C

The DAP/CP8 range is a fine-grain, massively-parallel computing system designed and manufactured by Active Memory Technology. The DAP (Distributed Array of Processors) model 510C has an array of 1024 processor elements which work in parallel to achieve high-speed computing.

The DAP can be attached to computer systems, such as Sun[1] and VAX[2], to gain speed improvements of up to several hundred times for many applications. The design of the DAP incorporates a high-speed input/output facility which can simultaneously transfer data at 40 Mbytes per second in each direction, with only a small effect on the processing speed. AMT provides a range of powerful software development tools and there are extensive software libraries to aid the rapid production of solutions for applications.

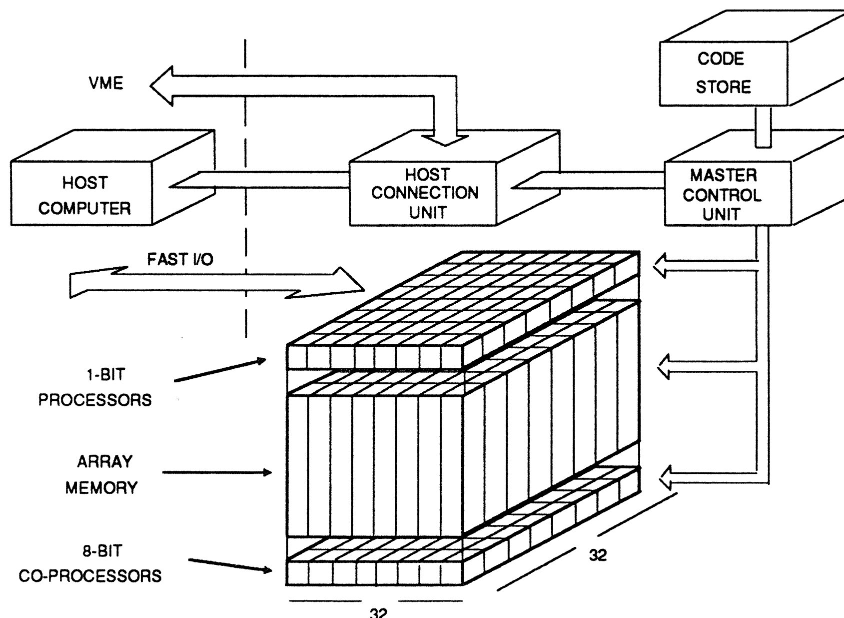
DAP HARDWARE

The DAP 510C is a SIMD (Single Instruction, Multiple Datastream) machine in which 1024 processor elements simultaneously execute the same instruction on data within their local memory. A technical description of the DAP is given in the AMT publication *DAP Series Technical Overview.*

The processor elements (PE) are arranged in a square array (32 x 32), each comprising a general-purpose bit-organized processor and an 8-bit wide co-processor (or CP8) for performing complex functions such as floating point arithmetic. Each PE has a local memory which ranges from 32 Kbits per PE (ie 4 Mbytes in total) up to a limit of 1 Mbits per PE set by the present architecture.

Each bit-organized processor is connected to its four nearest neighbors in the square array, and to row and column data paths used for rapid broadcast or combination of data. This processor also performs Boolean (logical) and arithmetic processing. However, computationally intensive tasks are generally done in the co-processor, which has 8-bit wide registers and data paths. Each co-processor has 32 bytes of storage for operands and intermediate results.

Program control is performed by a master control unit (MCU), which takes instructions from a code store, interprets them and controls the PEs, the memory and data transfers. The MCU also



Main components of the DAP 510C system.

1 *Sun is a trademark of Sun Microsystems Inc.*
2 *VAX is a trademark of Digital Equipment Corporation*

issues tasks to the co-processors, which execute in parallel a separate microcode instruction stream. A host connection unit (HCU) controls the interaction between the DAP and a host computer through a SCSI or VME bus interface. Also, the VME bus offers connection to other devices at up to 4 Mbytes per second.

High speed input and output is a major feature of DAP systems. The DAP 510C system requires only 3% of the processor cycles to sustain a transfer rate of 40 Mbytes per second. AMT supplies a number of fast I/O interfaces and peripheral sub-systems, including a high resolution video interface for viewing the data during processing, an I/O computer which can re-order the data during input and output, and a fast disk system. These products can be used to interface to other DAPs or to a variety of user-specified I/O devices.

The DAP 510C system is contained within a single cabinet that fits under a desk. The system requires standard 110 or 220 V power, is ambient air-cooled and needs no special environmental conditions.

DAP users will recognise the bit-organized processors in the DAP 510C as being identical to those in the current DAP 510 system. Existing DAP 510 systems can be easily upgraded, essentially by installing new array boards containing the additional co-processor chips.

DAP SOFTWARE

All DAP systems are supported by a wide range of software:

* A high-level language called Fortran-Plus, which is an extended version of Fortran that allows the handling of matrices and vectors of any size, taking advantage of their intrinsic data parallelism.
* An assembly language called APAL, providing low-level control of the MCU and PE array if required.
* An interactive debugging system, which allows users to examine the state of the memory in terms of high-level program language variable names.
* A DAP run-time system supports host–DAP communications and access to fast I/O and VME devices.
* A general support library of mathematical functions, graphic support software and libraries for image and signal processing.
* A DAP simulator to allow those who do not have access to DAP hardware to develop and test DAP programs.

Program development takes place on Sun host systems under the UNIX[3] operating system, and on VAX hosts under VMS[4].

PERFORMANCE AND RELIABILITY

Performance

Examples of the performance of the DAP 510C massively parallel system are:

* transfers between memory and processors at 1.3 Gbytes per second;
* Boolean (logical) operations at 10 billion per second;
* character handling at 1 billion per second;
* 8-bit integer multiply at 600 million per second
* 32-bit floating-point addition at 140 million per second.

Reliability and Integrity

The DAP 510C is designed to be resilient and offers high reliability and easy maintenance. All data paths and memory in the system are parity checked. The complete array of PEs is duplicated, and all operations are checked by master/slave comparison, thus giving complete integrity – ensuring that all answers provided are 'correct.' The master control unit also validates the operations of its scalar computational units by master/slave comparison.

3 *UNIX is a trademark of AT & T Bell Laboratories*
4 *VMS is a trademark of Digital Equipment Corporation*

Specifications of the DAP/CP8 model 510C

OPERATIONS

Data types .... Boolean, integer, character, floating point

Precision .... 8 to 64 bits integer, in 8-bit steps
24 to 64 bits real, in 8-bit steps

Dimensions .... scalar
vector (1 parallel dimension)
matrix (2 parallel dimensions)

INSTRUCTIONS

Address types ...... row/column, word, bit-plane, register bit

Addressing facilities .... direct or modified,
auto increment/decrement

Addressing range – data .... 128 Mbytes

Addressing range – code .... 4 Mbytes

MEMORY

Data memory .... 4, 8, 16 or 32 Mbytes

Program memory .... 0.5, 1, 2 or 4 Mbytes

Type .... static RAM

Error handling .... parity checks

Storage protection .... datum and limit

MASTER CONTROL UNIT (MCU)

Functions .... scalar processor, array controller

Registers .... 14 general purpose (32 bit),
1 array edge size (32 bit), and 1 privilege link (32 bit)

Instruction sequencing .... program counter register,
and hardware-controlled low-level loop

Instruction control .... 4-stage pipeline

Array control .... single instruction,
multiple datastream (SIMD)

Error handling .... parity checks on data paths,
master/slave checking on ALU components

PROCESSOR ARRAY

Number of PEs .... 1024, arranged 32 x 32,
each comprising one bit-organized processor
and one co-processor

Bit-organized processor .... 1-bit wide data paths with
accumulator, carry and activity registers

Co-processor ...... 8-bit wide data paths and arithmetic unit,
32-bit wide accumulator, 32 bytes of storage

Connectivity .... row and column data paths,
4-way nearest neighbors

Error handling .... memory parity check,
parity check on MCU data paths,
master-slave checking on processor elements

INTERRUPTS

MCU levels .... hardware fail, input/output subsystem,
user program error, and HCU

HCU communications .... MCU interrupts HCU

HOST CONNECTION UNIT (HCU)

Processor .... MC68020 system

Memory .... 256 Kbyte PROM, 1 Mbyte dynamic RAM

Error handling .... memory parity check

External interfaces:
SCSI port: to Sun systems, etc
VME bus: to VAX computers via DR11-W or DRB32
interfaces, or to user-defined interfaces
Aptec IOC via Openbus
RS232 (two ports)

SPEED

MCU and processor array .... 10 MHz clock

Host Connection Unit .... 10 MHz clock

VIDEO INTERFACE (DAP-VO)

Operation .... a high resolution graphic frame buffer

Bandwidth .... up to 38 Mbytes per second

Buffer .... 2 x 1 Mbyte (8-bit data option)
or 2 x 3 Mbytes (24-bit data option)

Video format .... 1024 x 1024, 1:1 pixel aspect ratio,
1 to 8 bits per pixel or 1 to 24 bits per pixel hardware
options; 60 Hz frame rate, non-interlaced,
RGB, separate sync or sync-on-green

Color display monitor .... Hitachi CM2086A,
Sony GDM1950 or equivalent

FAST I/O COMPUTER (DAP-IOC)

Operation .... performs a comprehensive re-formatting
of the input or output data in half duplex mode;
two units can be connected for full duplex operation

Bandwidth ... up to 40 Mbytes per second in each direction

Buffer .... 2 x 1 Mbyte buffer memories

FAST DISK SYSTEM

Operation .... offers a high-speed transfer
for bulk data applications

Bandwidth .... up to 16 Mbytes per second

Storage capacity .... up to 45 Gbytes

PHYSICAL

Dimensions .... height 25.0 inches (63.5 cm),
width 13.4 inches (34 cm),
width at base 17.0 inches (43 cm)
depth 31.0 inches (79 cm)

Weight .... 110 pounds (50 kg)

Power .... 110/220 VAC, 50/60 Hz, 750 Watts

Cooling .... internal fan

SOFTWARE

Fortran-Plus compilation system

General support library

Digital signal processing library

Image processing library

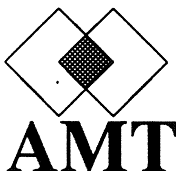
Device driver configuration package

APAL assembler

DAP simulator

OEM PRODUCTS

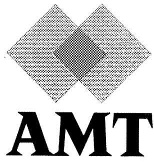
DAP systems are available as board sets for systems integrators.



Active Memory Technology Inc
16802 Aston Street, Suite 103
Irvine
California 92714
U.S.A.
Tel (USA) : (714) 261-8901
Fax (USA) : (714) 261-8802

Active Memory Technology Ltd
65 Suttons Park Avenue
Reading RG6 1AZ
Berkshire
United Kingdom
Tel (UK) : 0734 661111
Fax (UK) : 0734 351395

AMT is a multinational manufacturer of massively parallel computer systems with Company headquarters in Irvine, California. AMT maintains Research and Development activities at its Irvine, California location and at its European headquarters in Reading, U.K.



Ferromagnetic Demonstration: Ising Model Simulation

Introduction

The simulation of the ferromagnetic properties of materials is greatly enhanced by having powerful graphical interaction and high-speed computation. The Distributed Array of Processors (DAP) is ideally suited to such applications which for the present demonstration performs at about:

2 x CRAY X-MP
10 x CYBER 205
500 x FPS-264

The DAP's price/performance is also outstanding.

Description

Ferromagnetic materials, such as iron, display magnetic properties below their critical temperature Tc. They spontaneously develop regions called 'domains' in which the elementary magnets (or spins) become aligned. The demonstration simulates the time evolution of a ferromagnet with the widely used Ising model on a 1024 x 1024 lattice. The display shows each lattice point with red for positive and white for negative polarities.

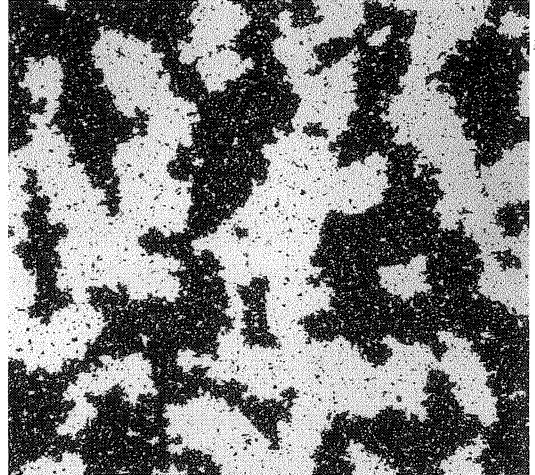
As the temperature of the material is reduced below Tc, interesting patterns appear which evolve by random walks of the boundaries between domains of opposite magnetisation, with small domains tending to die out.

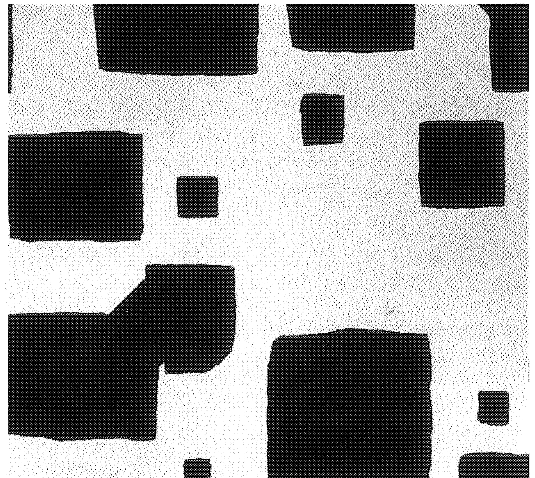
An external magnetic field will favour one of the two directions of magnetisation. If the field is applied at a time when the favoured direction (or phase) is in a minority, then bigger clusters of the favoured phase will tend to grow until the whole system is predominantly that phase; smaller clusters of the favoured phase will tend to die out. The critical cluster size and other properties of the domain dynamics depend on the values of the temperature and field, and how they vary with time.

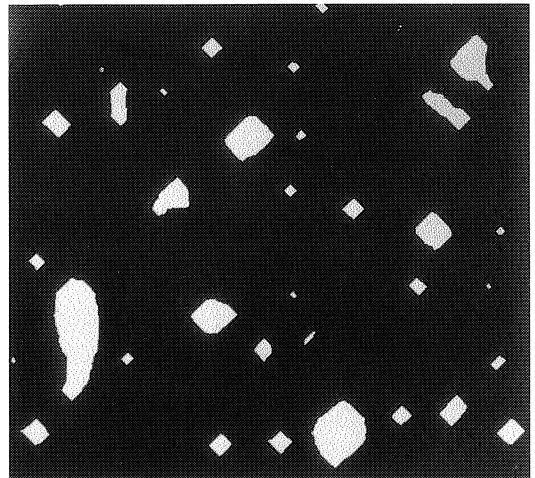
Fully Interactive

The DAP video output enables the state of the lattice to be directly viewed as the computation proceeds, with very little processing overhead. The main demonstration allows the temperature and field to be varied interactively using a 'mouse' attached to the host computer. The complete lattice is updated 20 times per second. The program can be slowed down in order to follow rapid changes.

The variation of the domain shapes is illustrated in the pictures. At temperatures around Tc and at zero field, both fractal geometries and domains within domains are observed. Below Tc and with a magnetic field,







A ferromagnet just below its critical temperature. The top picture shows the appearance of magnetic domains when there is no external field. The two lower images show the appearance of square and diamond shapes when magnetic fields are applied.

expanding domains are observed as *square* shapes and contracting domains as *diamond* shapes, phenomena not directly observed before.

Computational physicists perform massive calculations using the Ising model; the DAP's combination of a fully interactive capability and the highest computational speeds adds a new dimension for both teaching and research.

Algorithm and Codes

The lattice is represented by an array of logical variables. It is updated by each spin deciding whether to flip its magnetisation state according to a count of the states of its four neighbours and a probability table. The table is determined by energy considerations from the temperature and field values, and the selected probability is compared against a good quality 24-bit random number. The code uses the Monte Carlo method and runs at 20 million spin flips per second in the fully interactive mode.

Faster codes are available for zero field and for fixed temperatures. A subsidiary demonstration shows a much faster code for zero field that uses a cellular automata approach.

Performance

The measured performance for the DAP 500 using various codes is compared below with recently published data on some other machines for *fixed temperature* and *zero field*.

Code	Machine	Million spin flips/sec	
Monte Carlo	DAP 500	105	
	CYBER 205	38	(ref. 1)
Cellular automata	DAP 500	1250	
	CRAY X-MP	670	(ref. 2)
	CYBER 205	117	(ref. 3)
	FPS-264	2.2	(ref. 4)

1. Wansleben S, *Comp. Phys. Commun.* 43 (1987) 315
2. Herrmann H J, *J. Stat. Phys.* 45 (1986) 145
3. Creutz M et al, *Comp. Phys. Commun.* 42 (1986) 191
4. Barkai D et al, *John von Neumann Center, Princeton*, Report JVNC 86-1 (1986)

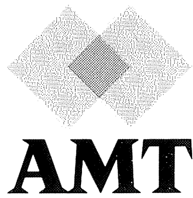
Benefits of using the DAP

DAP is the most cost-effective machine for this application, having the highest performance as well as being fully interactive. It is ideally suited to handling logical variables over large lattices.

This demonstration is based upon work done by the Physics Department of Edinburgh University and AMT consultants.

Active Memory Technology Ltd.,
65 Suttons Park Avenue, Reading RG6 1AZ,
United Kingdom.
Telex: 846931. Tel: 0734 661111. Fax: 0734 351395

Active Memory Technology Inc.,
16802 Aston St., Suite 103,
Irvine, California 92714, U.S.A.
Tel: (714) 261 8901. Fax: (714) 2618802



AMT

DAP Video Interface – DAP-VO

HIGH-RESOLUTION VIDEO INTERFACE

The DAP Video Interface – DAP-VO – is a high resolution graphic frame buffer which provides high speed transfer of data from the DAP massively-parallel computing system to a color display. This allows the user to view the data in the memory of the computer while the calculations are proceeding. Because of the basic architecture of the DAP, this visualization of the data has only a small effect on the processing power. The DAP-VO interface transfers data from the DAP computer memory into an internal frame buffer at a peak rate of 40 Mbytes per second. Software is available to provide an efficient means for image generation and display. Video circuitry attached to the frame buffer drives a high-resolution color monitor at 64 KHz line frequency, to give a 1024 × 1024, 60 Hz non-interlaced image. The instantaneous response of the system offers a powerful facility for the rapid control of your computations. The DAP-VO is particularly useful for applications such as signal and image processing, where the display is an integral part of the calculation.

HIGH-SPEED DATA TRANSFER

The AMT DAP (Distributed Array of Processors) system comes in two models:

	Number of processors	*Memory bandwidth*
DAP 510	1024	1.3 Gbytes per second
DAP 610	4096	5.1 Gbytes per second

The processor array has been specially integrated with a high-speed input/output facility (known as the D plane). A full description of the DAP system is contained in the AMT publication *DAP Series Technical Overview*.

To output data, the DAP-VO unit loads the contents of a memory plane into the D plane in one DAP clock cycle (100 nanoseconds). This suspends the normal DAP instruction stream for one cycle only. The data in the D plane is then clocked out of the array at 10 MHz to give a sustained data transfer rate of 38 Mbytes per second, including all the overheads. The shifting out of the data is independent of the normal processing and therefore this transfer rate uses only 0.8% of the DAP 610 cycles, or 3% of the DAP 510 cycles.

HARDWARE

The DAP-VO interface is a high-resolution video frame buffer, which provides real-time data visualization for both the DAP 510 and DAP 610 systems. The interface is available in two hardware configurations:

- DAP-VO-8, which contains two 8-bit image buffers with 8-bit to 24-bit look-up tables for displaying up to 256 colors selected from a total of 16 million. The size of each image buffer is 1 Mbyte.
- DAP-VO-24, which contains two 24-bit image buffers (8 bits each of red, green and blue) for directly displaying up to 16 million colors. Each image buffer is 3 Mbytes.

The double frame buffer configuration allows one frame to be displayed while a second is being updated with new data from the array. While the data transfer rate is the same for either board, the frame is three times faster for a VO-8 using 8-bit pixels than it is for a VO-24 using 24-bit pixels. The interface has ***red-green-blue*** and ***sync*** outputs which are used to feed a high-resolution monitor (see back page for specifications).

SOFTWARE

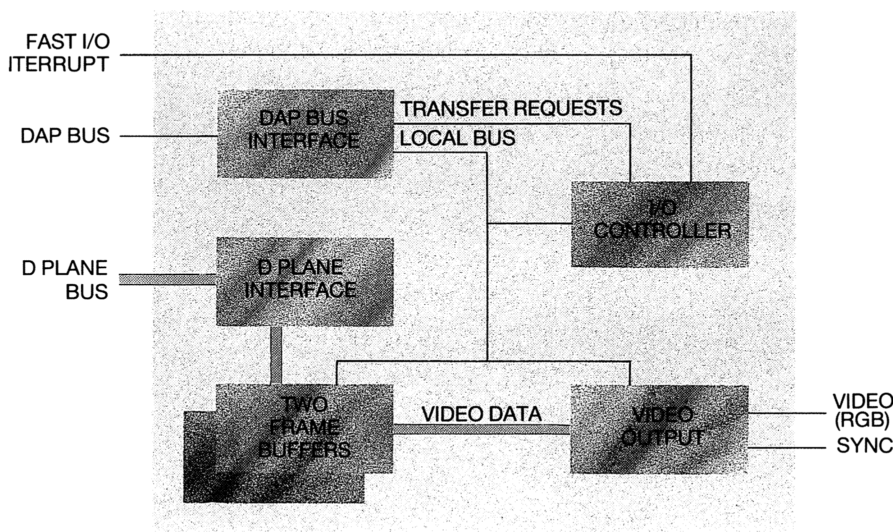
A set of software associated with the video interface provides a straightforward and efficient means of using the data visualization facility of the DAP. Basic software is provided to construct images and cause them to be displayed on a high-resolution color monitor. The displayed image is 1024 × 1024 pixels, each pixel being stored with a precision of up to 8 bits or 24 bits depending on the hardware option installed.

The software includes routines to draw dots, lines and characters in a data area in the user's program. This data area may be copied to the hardware frame buffer on a one-off or a regular basis. The software is optimized for the DAP architecture.

Specifications

DAP VIDEO INTERFACE

The video interfaces DAP-VO-8 and DAP-VO-24 are compatible with the DAP 510 and DAP 610 computer systems. These interfaces achieve a sustained data transfer rate of 38 Mbytes per second.



Functional block diagram of the high-resolution video interface.

IMAGE CHARACTERISTICS

	DAP-VO-8	DAP-VO-24
Dimensions	1024 × 1024	1024 × 1024
Pixel format	from 1 to 8 bits	from 1 to 24 bits
Color generation method	translated using look-up tables	direct RGB specification
Color range	up to 256 (out of 16 million)	up to 16 million
Frame buffer size	2 × 1 Mbyte	2 × 3 Mbytes

VIDEO CHARACTERISTICS

Display .... 60 Hz frame rate, non- interlaced

RS-343A compatible

Signal output .... RGB analog, 0.7 volts peak-to-peak, BNC type connector, 75 ohm

sync: TTL, negative polarity either sync-on-green or separate sync

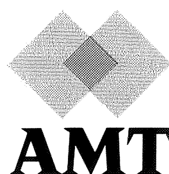
HORIZONTAL TIMING

Pixel clock	107.88 MHz (9.27 ns)
Line frequency	64.06 KHz (15.61 μs)
Active time	9.49 μs
Front porch	1.33 μs
Sync pulse	1.63 μs
Back porch	3.15 μs

VERTICAL TIMING

Frame rate	60.15 Hz (16.625 ms)
Active time	16.0 ms 1024 lines
Front porch	46.8 μs 3 lines
Sync pulse	46.8 μs 3 lines
Back porch	546.3 μs 35 lines

Typical color monitors which conform to these specifications include the Hitachi CM2086A and Sony GDM1950 models.



Active Memory Technology Ltd
65 Suttons Park Avenue
Reading RG6 1AZ
Berkshire
United Kingdom
Tel (UK): 0734 661111
Fax (UK): 0734 351395

Active Memory Technology Inc
16802 Aston Street, Suite 103
Irvine
California 92714
United States of America
Tel (USA): (714) 261-8901
Fax (USA): (714) 261-8802

AMT is a multinational manufacturer of massively parallel computer systems. Manufacturing and hardware R & D are conducted in the Irvine, California location. Software R & D activities are carried out at the company's facilities in Reading, UK.

AMT DAP—a processor array in a workstation environment

D J Hunt

The principle of single-instruction-stream multiple-data-stream computing (SIMD) is embodied in the AMT DAP 500 and DAP 600 processor arrays, which have respectively 1 024 or 4 096 one-bit processors embedded in a memory module. The distributed array of processors (DAP) system has a memory bandwidth of up to 5.1 Gbyte/s. A standard Sun or VAX computer system acts as host to DAP, thus placing it in a workstation environment. A direct output from the DAP memory to a video monitor leads to the concept of 'data visualization'. Most applications programming is done in an extended FORTRAN language having matrices and vectors as basic elements as well as the usual scalars. The DAP architecture has been applied by more than 1 000 users, and there is thus a wealth of expertise in applications and parallel algorithms that can be exploited.

Keywords: SIMD computing, DAP system, data utilization, FORTRAN, parallel algorithms

A number of themes provide the basis for the DAP 500 and DAP 600 systems:

- Parallelism through use of multiple function units is the key to ever increasing computational power.
- For a very wide class of applications, a single instruction–multiple data (SIMD) architecture is efficient in that only a single control unit is needed, and a regular grid interconnect gives a match to many problem data structures.
- Local memory associated with each processor gives high aggregate bandwidth, thus avoiding memory bottlenecks associated with scalar processors. In effect, the processing power is distributed throughout the memory.

Active Memory Technology Ltd, 65 Suttons Park Avenue, Reading, Berks. RG6 1AZ, UK

- VLSI technology is ideally suited to replication of functional units, thus giving a physically compact implementation.
- There is an increasing emphasis on the use of workstations, either alone or as an adjunct to some central computer system. Hence, DAP is configured as an attached processor, working in conjunction with such a workstation, or as a server on a network of workstations.
- 'Data visualization' via a high-resolution colour display is important in monitoring the progress of a computation, and in interpreting its results. Hence, a small fraction of the high-memory bandwidth of the DAP is made available for video output or other high-speed input–output operations.

ARCHITECTURE

The DAP 500[1] and DAP 600 are a new implementation, by Active Memory Technology, of the DAP concept previously researched by ICL[2–4]. DAP installations include the University of Edinburgh and Queen Mary College, London. The latter operate a DAP service, bringing the total number of users to over 1 000 worldwide, and representing a wide variety of applications.

In providing processing power through a set of individual processors, a fundamental design issue is the complexity of the individual processors: for a given quantity of hardware, should there be a few powerful processors or a larger number of simple ones? DAP takes the extreme of the latter approach where the processors (known as processor elements, or PEs) are just one bit wide and there are very many of them: 1 024 in DAP 500, or 4 096 in DAP 600. Having very simple processors gives maximum flexibility in the higher level functions that can be programmed.

The memory data path is also one bit wide for each PE, and at 10 MHz clock rate this gives an overall

 0267–6192/89/020107–08© 1989 Butterworth & Co (Publishers) Ltd

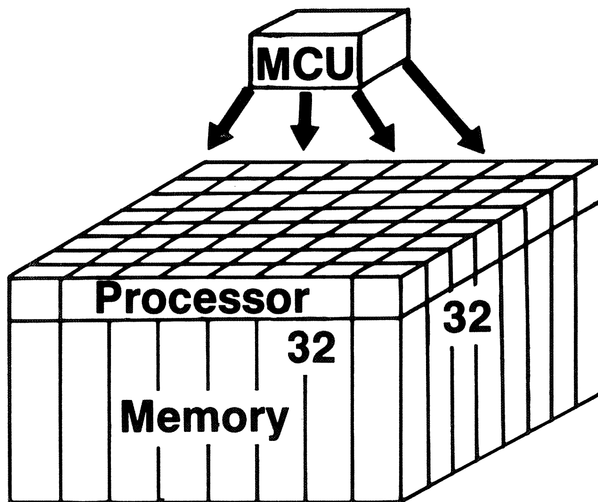


Figure 1. Principle of the DAP

memory bandwidth of 1.3 Gbyte/s for DAP 500 or 5.1 Gbyte/s for DAP 600. The logic of 64 processor elements (PEs) fits in a 176 pin semicustom chip, and the memory is provided using standard static memory components.

Figure 1 shows the basic principles of DAP operation. This shows a DAP 500, which has 1 024 PEs arranged in a square array of 32 × 32, each with a one-bit wide memory below it, thus giving a cuboid of memory. The memory size is currently either 32 kbit or 64 kbit for each PE: a total of 4 Mbyte or 8 Mbyte for the whole array. DAP 600 has the same architecture, but has 4 096 PEs arranged 64 × 64, and at least 16 Mbyte of memory.

Memory addresses are common to all PEs, so in a given instruction each PE accesses a bit of memory at the same memory address. Equivalently, a complete 'plane' of memory is being accessed at the same time. Generally, each data item is held in the memory of a particular processor, with successive bits of each data item occupying successively addressed locations. Thus a matrix of 16 bit values, for example, occupies 16 consecutive bit-planes of the memory. Some instructions, provide access to one row of memory (corresponding to one row of PEs), or one word of memory (32 consecutive bits in a row), selected from a memory plane.

The instruction stream, common to all the PEs, is broadcast from a 'master control unit' (MCU). To be more precise, the MCU performs address generation and decoding on behalf of all the processors, and a set of decoded control signals is broadcast. This means that little decoding is needed in individual processors.

Processor element functions

Figure 2 shows the main components of one PE and its connection to memory, each of the data paths and registers shown being just one bit wide. The main components of the PE are three one bit wide registers, named A, C and Q respectively, and an adder that performs arithmetic and logic functions. The detailed usage of the registers depends on how the DAP is programmed, but the Q register generally serves as an accumulator, and the C register holds a carry.

In a typical PE operation, the input multiplexor (at the left side of the figure) selects an operand bit from memory which is then input to the adder along with values from the C and Q registers. These three inputs are added, producing a sum, which is optionally clocked into Q, and a carry, which is optionally clocked into C. An add operation on 16 bit integers, for example, is performed by 16 successive one bit adds starting at the least significant end, with the C register holding the carry between each bit position and the next.

The multiplexor near the right side of the figure selects data to be written back to the memory. In some cases, the A register acts as a control input to this multiplexor. This means that for a given instruction, in those PEs where the A register is *True* the sum output of the adder is written to memory, but in those where A is *False* the old memory contents are selected and rewritten. This activity control is equivalent to 'switching off' selected PEs in that the old memory contents are preserved, and is very important in implementing conditional operations both within functions and at the applications level.

An activity pattern may be loaded into the A register via the input multiplexor, and there are options to either write that value directly into A or to AND it with the existing A register contents. This latter option is very convenient for rapidly combining several conditions.

Array interconnect

Although much processing is done with each PE working on data in its own local memory, it is important to be able to move data around the array in various ways, and two such mechanisms are provided.

The first interconnection mechanism is that each PE can communicate with its four nearest neighbours in the array. Inputs from these North, East, South and West neighbours are shown at the left of Figure 2. Since the PEs implement a common instruction stream, then if one PE is accessing its Northern neighbour, say, then so is every PE. The overall effect is that a plane of bits in the

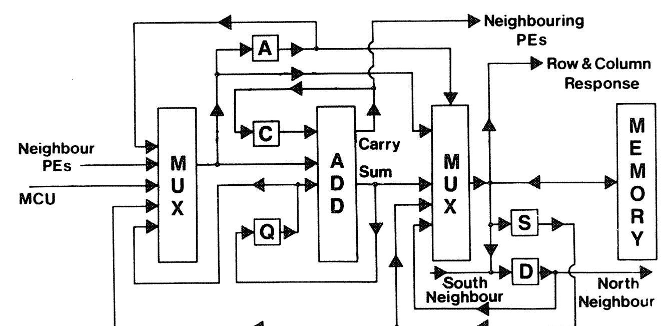


Figure 2. A processor element

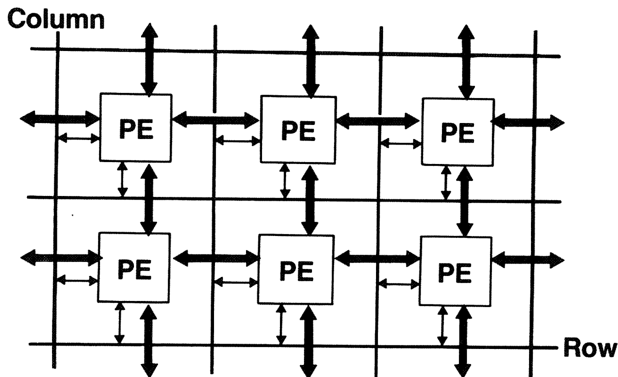


Figure 3. Connectivity

Q registers of the set of PEs may, in one clock cycle, be shifted one place in any of the four possible directions. A matrix of words may be shifted by shifting each bit plane of the matrix in turn, and longer distance shifts are implemented by successive shifts of one place. Such operations are important in applications such as the solution of field equations by relaxation.

The second array interconnection is two sets of data paths that pass along the rows of PEs and along the columns of PEs respectively. These paths are shown in Figure 3 for a small section of the array, along with the nearest neighbour connections mentioned above. One use of these data paths is in broadcasting a row of bits, such that each row of PEs receives the same data pattern, and similarly for broadcasting a column of bits. These broadcast operations each take just one basic clock cycle. Another use is in extracting data from the array, where the basic operations are to AND together all the rows or to AND together all the columns; variants of these operations permit data to be extracted from a single row or column.

Row and column data paths are important in areas such as matrix algebra. For example, in matrix inversion they are used to extract the pivot column of a matrix and replicate it in every column. Another case is a rapid global test implemented by ANDing all the rows in the array and then testing the result for all 1s.

Master control unit

As already mentioned, the PE array is controlled by the master control unit (MCU). This is a 32 bit processor responsible for fetching and interpreting instructions, and its main components are shown in Figure 4. Instructions are held in a codestore (separate from the array data store) which holds 512 kbyte or 1 Mbyte, each instruction being 4 byte.

The MCU implements scalar functions including control transfers such as subroutine entry and exit, but most of the instructions are passed on to the array for execution. Hence, an important part of the instruction decode is to provide control signals for the PEs.

Array operations usually involve loops of a few instructions, and the MCU provides explicit hardware support for these via a 'DO' instruction in the assembly language. Once such a loop is initiated, there are no overheads for loop control since this is dealt with by dedicated hardware. Another important aspect of these loops is the ability for instructions within the loop to automatically reference successive store addresses on successive passes of the loop, again without any execution time overhead.

The lower half of the figure is a conventional data path. The register file has dual port access, and the usual arithmetic, logical and shift operations may be performed upon the register contents. There are 14 general purpose registers, and one register dedicated for supervisor use.

Some of the array instructions involve broadcasting data to the array, and some receive data from the array; these paths connect to the MCU data paths as shown at the bottom right of the figure. The details depend on the array size:

- On DAP 600 the MCU data paths are 32 bit wide but the array paths are 64 bit wide. Logic known as the array support unit interconnects these data paths and provides a further 'edge register' of length 64 bits, which thus matches the array edge size. Complete edge-sized values (for example an array row) may be transferred between the array and the edge register, or words (32 bit items occupying half of a row or column) may be transferred between the array and an MCU register.
- On DAP 500 there is a direct match between the 32 bit data paths in the MCU and the 32 bit width of the row and column data paths in the array. Hence, no array support unit is needed, but the equivalent edge register functionality (32 bit wide) is provided using a spare location in the MCU register file.

Fast input–output

In many applications, fast processing implies high data rates to or from external devices, and the DAP architecture provides this capability with only a small impact on processing performance. In the processor element diagram (Figure 2), the register shown as D may

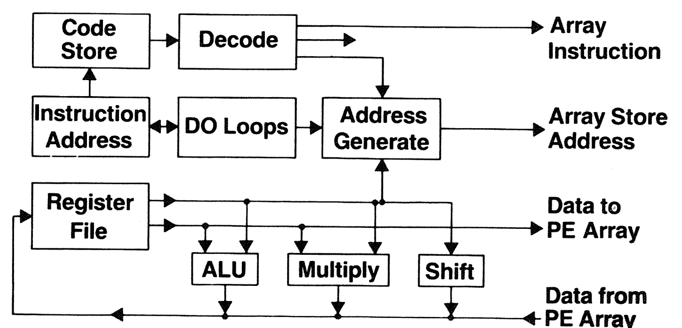


Figure 4. Functions of the master control unit

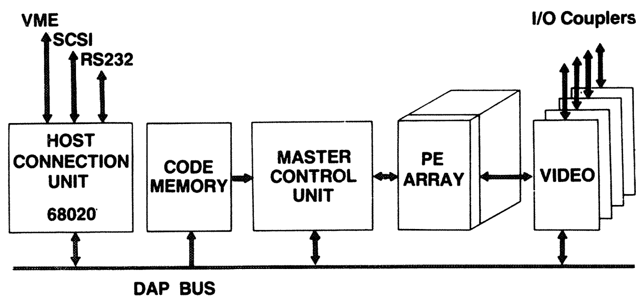


Figure 5. DAP system organization

be loaded from the memory or written back to the memory, but does not otherwise take part in PE operations. Having loaded the plane of D registers from a plane of memory, the D plane may be shifted towards the North edge of the array, so that successive rows of the D plane are output at that edge. The D plane may similarly be used for input to the DAP by presenting successive data words as inputs at the Southern edge of the array. This shifting is done independently of the normal array instruction stream and is usually done at a faster clock rate.

The D plane is controlled by one of a number of input–output couplers, rather than by the instruction stream, so this I/O facility may be thought of as a fast DMA operation. For shifting the D plane, the coupler clocks the D plane independently of the array instruction stream. However, to load or store the D plane (which is, of course, necessary after each complete plane has been output or input), the coupler makes a request to the MCU along with the required memory address. This loading or storing suspends the normal instruction stream for one clock cycle.

Where a DAP system has more than one coupler installed, the couplers arbitrate amongst themselves for use of the D plane, so the available bandwidth is shared between them. Couplers include buffering, which both matches the D plane clock speed to the external interface and avoids any crisis times associated with D plane arbitration.

For the DAP 500, the D plane connection at the edge of the array is 32 bit wide, and the overall bandwidth is 50 Mbyte/s, while only using 4% of the available memory bandwidth. For the DAP 600, the D plane connection is potentially 64 bit wide, but in practice most couplers will have only 32 bit wide paths giving again 50 Mbyte/s, but using 1% of the memory bandwidth.

An important coupler already in use provides a video display facility. The patterns to be displayed are computed by the DAP in the array memory, then transferred to one of two frame buffers in the coupler for continuous refresh of a standard monitor via the usual colour lookup tables and digital-to-analogue converters. The screen resolution is 1 024 × 1 024 pixels, each pixel being eight bits, or with an option for 24 bit.

Other couplers provide an external digital interface for input or output, either via a simple FIFO buffer, or via a more sophisticated double buffer arrangement with facilities for general purpose reordering of data as it passes through the coupler. External interfaces include a serial link (which may provide DAP-to-DAP connection, or links to other bus systems via a remote adaptor card), or high-density digital TV.

System organization

Figure 5 shows how the various DAP components connect together. With the DAP 600, the array support unit, mentioned above, is added between the MCU and the array. During DAP processing there is a continuous flow of instructions from the code memory to the MCU and thence to the array. If I/O is in progress, then there is a concurrent flow of data between the D plane in the array and an I/O coupler.

The DAP bus along the bottom of the figure provides overall communication, including interfacing to a host computer system which provides program development facilities and access to conventional filestore and peripherals. A typical host is a Sun system running Unix* (connected via a SCSI interface), or a VAX system running VMS** (connected via a VME bus), the detail of the interfaces being managed by the HCU (host connection unit). DAP programs are downloaded into the code memory and array memory via the DAP bus, which also serves as a command interface to control entry to specified DAP subroutines. The RS232 interface shown in the figure is available for diagnostic purposes.

The HCU is built around a Motorola 68020 microprocessor system running the VRTX† real time kernel. It performs a supervisory function including the allocation of DAP memory to user programs. In the event of a hardware failure it also provides a diagnostic capability. The lowest level of control of DAP user programs resides in the MCU itself which runs a supervisor program in privileged mode to deal with scheduling of user programs and interrupt handling.

Control of the I/O couplers is done by privileged MCU code in response to a supervisor entry from a user program, the necessary sequence of commands being sent to the coupler via the DAP bus. When a transfer is in progress, the coupler also uses the DAP bus to request the MCU to perform D plane loading or storing; this is actioned automatically by the MCU hardware.

The VME bus may also be used to connect to medium speed peripherals, taking advantage of the wide range of standard boards available in that format. A transfer on such a device is again initiated by a supervisor call from a user program, but the MCU passes on the request to the HCU for detailed management of the transfer.

---

* Unix is a trademark of Bell Telephone Laboratories
** VAX and VMS are trademarks of Digital Equipment Corporation
† VRTX is a trademark of Hunter & Ready Inc.

SOFTWARE

FORTRAN Plus

The main programming language for DAP is known as FORTRAN Plus*[5]—an extended version of FORTRAN 77 having matrices and vectors as basic elements as well as scalars. This language has been the main influence on the parallel processing facilities in the proposed ISO/ANSI standard, currently referred to as FORTRAN 8X.

To illustrate some of the features of FORTRAN Plus, as well as parallel algorithm techniques, an example subroutine is given below. This performs a solution of Poisson's equation using an elementary relaxation techniqe; naturally the DAP has in practice been used with many more sophisticated algorithms.

```
1        SUBROUTINE SOLVE( FIELD , EPSILON)
2        REAL          FIELD( , )
3        REAL                          EPSILON
4 C
5        REAL OLD( , ) , TEMP( , )
6        LOGICAL INSIDE( , )
7 C
8        INSIDE = ROWS( 2, 31 ) .AND. COLS( 2, 31 )
9        FIELD( INSIDE) = 0.0
10 C
11 100 CONTINUE
12       OLD = FIELD
13       TEMP = SHSP( FIELD ) + SHEP( FIELD )
14       FIELD( INSIDE ) = 0.25*
         ( TEMP + SHNP( SHWP( TEMP ) ) )
15       IF ( MAXV( ABS( FIELD - OLD ) ) .GT. EPSILON )
         GOTO 100
16 C
17       RETURN
18       END
```

Subroutine SOLVE is callable from another FORTRAN Plus subroutine and takes two parameters: a real matrix FIELD which contains initially the boundary conditions and into which the solution is to be placed; and a real scalar EPSILON which is a measure of the final accuracy required. In FORTRAN Plus, an array declaration with the first subscript null signifies a vector whose length matches the DAP dimension, or in general a set of such vectors. In this case (lines 2, 5 and 6) the first two subscripts are null, meaning a matrix that matches the DAP in size.

Line 8 sets up a logical matrix to define the interior region; points where the matrix is *False* are fixed boundary points. In this example, intrinsic functions ROWS and COLS are used; ROWS(2,31) is a Boolean expression which is *True* in rows 2 to 31 inclusive and *False* elsewhere (rows 1 and 32 for DAP 500). Thus, a 30×30 region of interior points is defined.

If this code was compiled to run on the DAP 600, the interior region would occupy only the top left quadrant of the processor array. However, changing the row and column numbers specified in line 8 would allow a problem with an interior region of up to 62×62 points to be computed on the DAP 600. In general, any required boundary shape could be defined in line 8, and the rest of the code would be independent of that shape.

* Previously known as DAP FORTRAN

In line 9 the interior points are initialized. This is an assignment of a real scalar value to a matrix, the scalar being implicitly replicated to form a matrix-sized expression. Use of a Boolean matrix as a mask on the left of the assignment, in this case INSIDE, causes the assignment to be performed only where the mask is *True*. Thus the boundary values are preserved.

Line 12 saves the current field value in order to detect changes later; a single statement copies an entire matrix, element to corresponding element, all the copies being done in parallel.

Lines 13 and 14 perform the relaxation step. At each point the sum of the four neighbouring points in the North, East, South and West directions is needed. A shifted copy of a matrix can be produced by functions like SHSP (SHift to the South with Plane geometry). This summation appears to need three adds per point, but closer examination shows that intermediate results can be shared between nearby points, hence the use of the temporary matrix TEMP. In line 14, note again the multiplication of the scalar value 0.25 by every element of the matrix expression, and the masked assignment.

Line 15 evaluates the convergence criterion. Function ABS is applied element by element to its matrix argument. Then function MAXV returns a scalar result which is the maximum value of the elements in its matrix argument. This scalar value is then tested in the normal way inside the IF condition.

The above example illustrates how masked assignment is used for local conditional operations, whereas IF statements are used for global transfer of control, and hence tend to be used less frequently. Indeed, the manipulation of Boolean matrices and their use in masking, merging, and selection are important features of FORTRAN Plus.

Replication of a scalar has been seen above, and it is also possible to replicate vectors. This involves an explicit call to a function, to specify how the vector is broadcast: MATR gives a matrix of identical rows, whereas MATC gives a matrix of identical columns. Various indexing constructs are also available: for example, the extraction of a specified row or column of a matrix giving a vector result.

FORTRAN Plus offers a range of precisions: integers from 8 bit to 64 bit wordlength, and reals from 24 bit to 64 bit wordlength, in steps of eight bits in each case. There is a continuous tradeoff between wordlength and performance, and of course memory usage.

Assembly language

FORTRAN Plus provides a good match between the capabilities of the DAP hardware and the requirements of applications; the wordlength options and Boolean manipulation functions give great flexibility. However, in some applications areas other wordlengths or special number representations may be required for optimum performance.

These can be programmed in the assembly language

APAL. This necessarily involves bit level operations and detailed coding but two aspects mitigate the amount of work involved: the ability to mix FORTRAN Plus and APAL at the subroutine level, so it is often appropriate only to have a few critical routines in APAL; and the availability of powerful macro facilities in APAL, permitting in some cases an intermediate level language to be customized by the user.

Run time system

DAP programs written in FORTRAN Plus or APAL (or both) are compiled on the host and downloaded into the DAP for execution. A standard host program (written in FORTRAN or C) controls the execution of the DAP program by calling routines that perform the following functions:

- Load a specified DAP object program into the DAP.
- Transfer a block of data in either direction between an area in the host program and a named COMMON block in the DAP program.
- Initiate DAP processing at a named subroutine within the DAP program. The host program is suspended while the DAP is running; when the DAP executes a RETURN from that top level subroutine, the DAP program halts, and the host program is restarted.
- Release the DAP resource; this is also of course done automatically when the host program terminates.

In the event of a program exception in the DAP, such as floating point overflow, the default action causes output of a pattern showing where in the array the errors occurred, together with a route map of the sequence of subroutine or function calls that led to the error. Interactive debugging, using names of variables as in the source code, is available, as well as tracing facilities.

A DAP simulation function provides all the above facilities, but running entirely on a serial processor. Naturally, any DAP-specific peripherals (such as the video display) are not modelled, and the performance is much lower than a real DAP, but this does provide a route for experimentation with short sections of DAP code.

Library routines

The FORTRAN Plus compilation system incorporates extensive libraries of low level routines that implement arithmetic and data manipulation operations as the appropriate sequences of bit-level operations. The presence of such routines is however transparent to the user.

At a higher level, the extensive experience of using the DAP architecture has led to the provision of a wide variety of library routines to provide applications-related functions. Some example routines are:

- Matrix manipulation (e.g. inner product multiplication, inversion, eigenvalues)
- Signal processing (e.g. FFT and other transforms)
- Image processing (e.g. windowing, histograms, convolutions)
- Data reorganization (e.g. sorting, permutation)
- Functions (e.g. random numbers, transcendental functions)
- Graphics.

Data mappings

The FORTRAN Plus programming example given earlier assumed a problem size that matched the PE array, or was smaller than the array. In practice, many problems are larger than the PE array, and in this case the processing is done in sections that match the size of the array.

The problem array is declared in FORTRAN Plus as a set of DAP-sized matrices, and the code works serially through the matrices of this set. The number of DAP-sized sections depends of course both on the problem size and on the size of DAP being used; the latter is accessible to the FORTRAN Plus program as a run-time value.

Two possible data mappings for 'oversize' problems are:

- Each array-sized section contains a neighbourhood of the problem array; this is referred to as a 'sheet' mapping.
- Each PE holds a neighbourhood of the problem array; this is referred to as a 'crinkled' (or 'pyramidal') mapping.

Which mapping is best depends on the pattern of communication between elements of the problem array. Often the best mapping is obvious, but in general the choice of mapping is an important aspect of algorithm design for processor arrays[6].

The advanced user may wish to employ more sophisticated mappings, or to change mappings during the course of a computation. To assist with this, a scheme known as parallel data transforms (PDTs)[7] is available. This is both a compact notation for describing regular data mappings, and a means for automatically generating code to transform between different mappings. Some of the library routines, in particular FFTs and sorting, use these PDT techniques.

APPLICATION AREAS

DAP was created to tackle large computing problems such as the solution of field equations and matrix manipulation. It is indeed very successful at such tasks, but there is a growing awareness of its suitability for a wider class of problem, and alongside this an increasing repertoire of parallel techniques that can help with the conversion of seemingly serial processes to parallel form.

Successful DAP applications obviously use large regular data sets, but it is not necessarily the case that all the PEs are usefully active all the time. Indeed the

Table 1. Performance capability of DAP 610 for processing matrix operands

	Logical	up to	40 000 Mops
8 bit	Character	up to	4 000 Mops
8 bit	Integer		
	Add		1 600 Mops
	Multiply		250 Mops
	Multiply by scalar constant	600 to	1 200 Mops
32 bit	Real		
	Add		48 Mflops
	Multiply		32 Mflops
	Square		64 Mflops
	Divide		24 Mflops
	Square-root		44 Mflops
	Maximum value (scalar result)		200 Mflops

DAP, with its facilities for rapid manipulation of Boolean matrices and setting of activity patterns, often deals more effectively, in relative terms, with conditional operations than a serial processor.

The following list gives an indication of the scope of DAP applications areas; it is by no means exhaustive:

- Field equations: wind tunnel, wave equation, Navier–Stokes equation[8], porous dam[9]
- Engineering: finite element analysis[10,11]
- Linear algebra[12]
- Physics: Ising model[13], molecular dynamics[14], lattice gauge theory, galactic simulation[15]
- Medical: NMR image reconstruction, DNA sequence matching[16]
- Radar: signal processing[4], shadow calculation
- Image manipulation[17]: filtering, feature extraction, rotation
- Graphics: molecular graphics[18], ray tracing
- Character handling: data searching[19], sorting
- Design automation: chip layout[20], fault simulation.

PERFORMANCE

Table 1 gives an indication of the performance capability of the DAP 610 (a DAP 600 running at 10 MHz) for processing matrix operands. The figures are given in units of Mflops (millions of floating point operations per second), or Mops (millions of operations per second).

Performance figures depend on the wordlength used, but also on the bit-level complexity of the function being performed, as illustrated by the figures above. For example, evaluating the element-by-element square of a matrix is faster than a general multiply because multiplication involves summation of a tableau of bits, and the library function exploits the symmetry in that tableau. Finding the maximum element of a matrix is fast because the algorithm uses an associative technique to progressively eliminate possibilities, starting by testing the most significant bit position of the matrix; unlike the obvious serial version of this function, it never explicitly compares any elements of the operand matrix.

As already mentioned, Boolean operations are especially fast. The most favourable case is an operation between a value in a PE register plane and a value in a memory plane, and putting the result into a PE register. This takes just one clock cycle to produce one Boolean result in every PE, giving the enormous processing rate quoted in the table.

The DAP is generally much better able to achieve and sustain the quoted performance figures than a conventional processor, since operations such as address arithmetic and even data movement usually take much less time than computing the arithmetic results, whereas on a conventional processor the times for these 'overheads' are often comparable with, or even greater than, the arithmetic.

CONCLUSIONS

The DAP has already shown its enormous potential, and its wide exposure to users has established a strong base of applications software and parallel algorithm techniques. The software is mature, permitting easy program development, and is ready to move forward with new developments in parallel processing languages. The recent emergence of DAP in a workstation environment makes it especially convenient to use, and the inbuilt display capability gives new insights into the nature of the computations as they are being performed.

REFERENCES

1 **Parkinson, D, Hunt, D J and MacQueen, K S** 'The AMT DAP 500' *Proc. 33rd IEEE Comp. Soc. Int. Conf.* (February 1988)

2 **Flanders, P M, Hunt, D J, Reddaway, S F and Parkinson, D** 'Efficient high speed computing with the distributed array processor' in **Kuck, D J, Lawrie, D H and Sameh, A H (Eds)** *High speed computer and algorithm organisation* Academic Press, USA (1977)

3 **Hunt, D J and Reddaway, S F** 'Distributed processing power in memory' in **Scarrott, G G (Ed)** *The fifth generation* Pergamon-Infotech, UK (1983)

4 **Reddaway, S F, Roberts, J B G, Simpson, P and Merrifield, B C** 'Distributed array processors for military applications' *MILCOMP 85. Military computers, graphics and software* Microwave Exhibitions and Publishers Ltd, Tunbridge Wells, UK (1985) pp 74–82

5 **Gostick, R W** 'Software and algorithms for the distributed array processors' *ICL Tech. J.* (May 1979) pp 116–135

6 **Flanders, P M** 'Non-numerical methods on parallel computers' *Comp. Phys. Commun.* Vol 26 (1982) pp 363–371

7 **Flanders, P M** 'A unified approach to a class of data movements on an array processor' *IEEE Trans. Comput.* Vol C-31 No 9 (September 1982)

8 **Grosch, C E** 'Adapting a Navier–Stokes code to the ICL DAP' *SIAM J. Sci. Stat. Comput.* Vol 8 No 1 (1987)

9 **Cryer, C W, Flanders, P M, Hunt, D J, Reddaway, S F and Stansbury, J** 'The solution of linear complementarity problems on an array processor' *J. Comp. Physics* Vol 47 No 2 (August 1982)

10 **Ducksbury, P G** *Parallel array processing* Ellis Horwood Ltd, UK (1986)

11 **Lai, C H and Liddell, H M** 'A review of parallel finite element methods on the DAP' *Appl. Math. Modelling* (11 Oct 1987) pp 330–340

12 **Reddaway, S F, Bowgen, G and van den Berghe, S** 'High performance linear algebra on the AMT DAP 510' *Proc. SIAM Conf. Parallel Processing for Scientific Computing* (December 1987)

13 **Reddaway, S F, Scott, D M and Smith, K A** 'A very high speed Monte Carlo simulation on DAP' *Comp. Phys. Commun.* Vol 37 (1985) pp 351–356

14 **Bowler, K C and Pawley, G S** 'Molecular dynamics and Monte Carlo simulations in solid state and elementary particle physics' *Proc. IEEE* Vol 72 No 1 (1984) pp 42–55

15 **Johns, T C and Nelson, A H** 'Particle simulation of 3D galactic hydrodynamics on the ICL DAP' *Comp. Phys. Commun.* Vol 37 (1985) pp 329–336

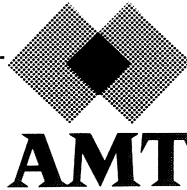
16 **Lyall, A, Hill, C, Collins, J F and Coulson, A F W** 'Implementation of inexact string matching algorithms on the ICL DAP' in **Feilmeir, M, Joubert, G R and Schendel, U (Eds)** *Proc. Conf. Parallel Computing 83* North-Holland, Amsterdam (1984)

17 **Oldfield, D E and Reddaway, S F** 'An image understanding performance study on the ICL distributed array processor' *IEEE Comp. Soc. Workshop Computer Architecture for Pattern Analysis and Image Database Management* (November 1985)

18 **Hubbard, R E and Fincham, D** 'Shaded molecular surface graphics on a highly parallel computer' *J. Mol. Graphics* Vol 3 No 1 (March 1985)

19 **Reddaway, S F and Page, R M R** 'High speed data searching with a processor array' in **Winter, S and Schumny, H (Eds)** *Proc. Conf. Euromicro 88, Zurich* Elsevier Science Publishers, UK (1988)

20 **Hunt, D J** 'Tracking of LSI chips and printed circuit boards using the ICL distributed array processor' in **Feilmeir, M, Joubert, G R and Schendel, U (Eds)** *Proc. Conf. Parallel Computing 83* North-Holland, Amsterdam (1984)



AMT

Active Memory Technology Ltd
65 Suttons Park Avenue
Reading
Berks
RG6 1AZ
United Kingdom
Tele. 0734 661111

Active Memory Technology Inc.
16802 Aston Street
Suite 103
Irvine
CA 92714
USA
Tele. (714) 261 8901

THE ICL DISTRIBUTED ARRAY PROCESSOR

D Parkinson

Large Systems National Research Region
International Computers Limited London

© D Parkinson 1977

D PARKINSON is a graduate of Queen's University Belfast having taken a 1st Class Honours Degree in Applied Mathematics and a PhD in Theoretical Quantum Physics. He has worked on the use of large computers to solve large-scale numerical problems for both the United Kingdom Atomic Energy Authority at Aldermaston, England, and the European Space Research Organization (now ESA) in Rome. In 1973 he joined ICL as a consultant in scientific programming and has recently been mostly concerned with the development of the ICL Distributed Array Processor.

THE ICL DISTRIBUTED ARRAY PROCESSOR

INTRODUCTION

Once upon a time goods were manufactured by a single craftsman who chose and purchased the materials, made his product and then sold the results of his own labours in his own shop.

The requirement for high volume, low cost production has led to the development of manufacturing industries using many people of lower skill levels who all cooperate to produce the finished product. In many ways computing and computers are still in the era of the single highly skilled craftsman. Most modern computers consist of a single central processing unit which accepts data and programs from peripherals, performs a number of manipulations upon it and sends the results to other peripherals. The major trend in computer improvement over the years has kept to this basic concept with only minor changes, allowing some power to peripheral processors to off-load some of the simpler tasks (making the apprentices serve in the shop). Computer power has increased mainly by increasing the skill level of the central processor (floating-point hardware, pipelining, etc).

The increasing demands for more and more computing with high cost-effectiveness is, however, leading to some unconventional computer designs which are no longer just more skilful versions of the von Neumann machine. New computer designs are appearing which replace the single processor with squads of processors. Although each individual processor in the squad may not be as powerful as the most powerful single processors the combination is much more powerful and cost-effective.

The best-known example of these new types of computer is the ILLIAC IV computer whose original design called for 256 individual medium-sized computers to be joined together to work under control of a single manager computer. Manufacturing, financial and organizational difficulties led to only one quarter of the original plans being implemented. This paper describes a new computer being developed by International Computers Limited which from one point of view represents a development of the ideas behind the ILLIAC IV, i e obtaining computing power by the replication in large numbers of a basic simple computer.

COMPUTING IN STORE

A major reason why it is difficult to speed up conventional computers is the functional separation of computing capacity and store. If one looks at a modern computer it is usually quite easy to find different parts of the machine which contain the store and

the arithmetic units: in fact, they are often placed in entirely separate cabinets. This separation automatically implies that some mechanism must exist for moving data backwards and forwards between the store and the computing units. This mechanism, the data highway, forms a potential bottleneck and many tricks are employed to attempt to limit its effects - cache stores, look-ahead, multiple word fetching etc. An alternative strategy, is to abandon the idea of physically separating the data store and the computing and to incorporate the processing capacity in with the store and build what can be called 'active stores' in contrast to the older forms of passive storage.

The best-known form of active store is the associative store or content-addressable store which is, as its name implies, addressed by content rather than by an absolute location. The best known example of a content-addressable computer is the STARAN, a product of Goodyear Aerospace. This class of computer is described by C Foster in this report (see page 169ff). The ICL Distributed Array Processor can be viewed as an associative processor and so forms a bridge between the ILLIAC IV type of array processor built for high-speed, floating-point number-crunching and the content-addressable processor envisaged for such problems as information retrieval.

OVERALL SYSTEM

The ICL Distributed Array Processor consists of a regular two-dimensional array of processors distributed amongst the store of a 2900 series host computer.

The sharing of memory between the 2900 series host and the DAP implies that the full facilities, hardware and software, of the 2900 series of computers can be used to provide input/output facilities, total system control and back-up processing (e g, compiling).

Although for many purposes the DAP may be viewed as a back-end processor, the data store is common to both systems, and hence no time is lost in transferring data across a data highway between the systems.

Figure 1 shows, schematically, how the DAP may be viewed as part of the host system. The DAP is connected as a specific store module and has three major components:

- An array of identical processing elements
- A master control unit (MCU) which issues identical instructions to each and every processing element
- A simple control interface for starting and stopping DAP processing.

THE ARRAY OF PROCESSING ELEMENTS

The array of processing elements typically oonsists of 1024, 4096 or 16384 elements arranged in 32 x 32, 64 x 64 or 128 x 128 square arrays. A processing element will usually be associated with a 4K bit section of the host computer store. Typical memory sizes are therefore ½ Mbyte, 2 Mbyte or 8 Mbyte. The amount of memory is not necessarily fixed at 4K bits per processing element: larger or smaller amounts may be

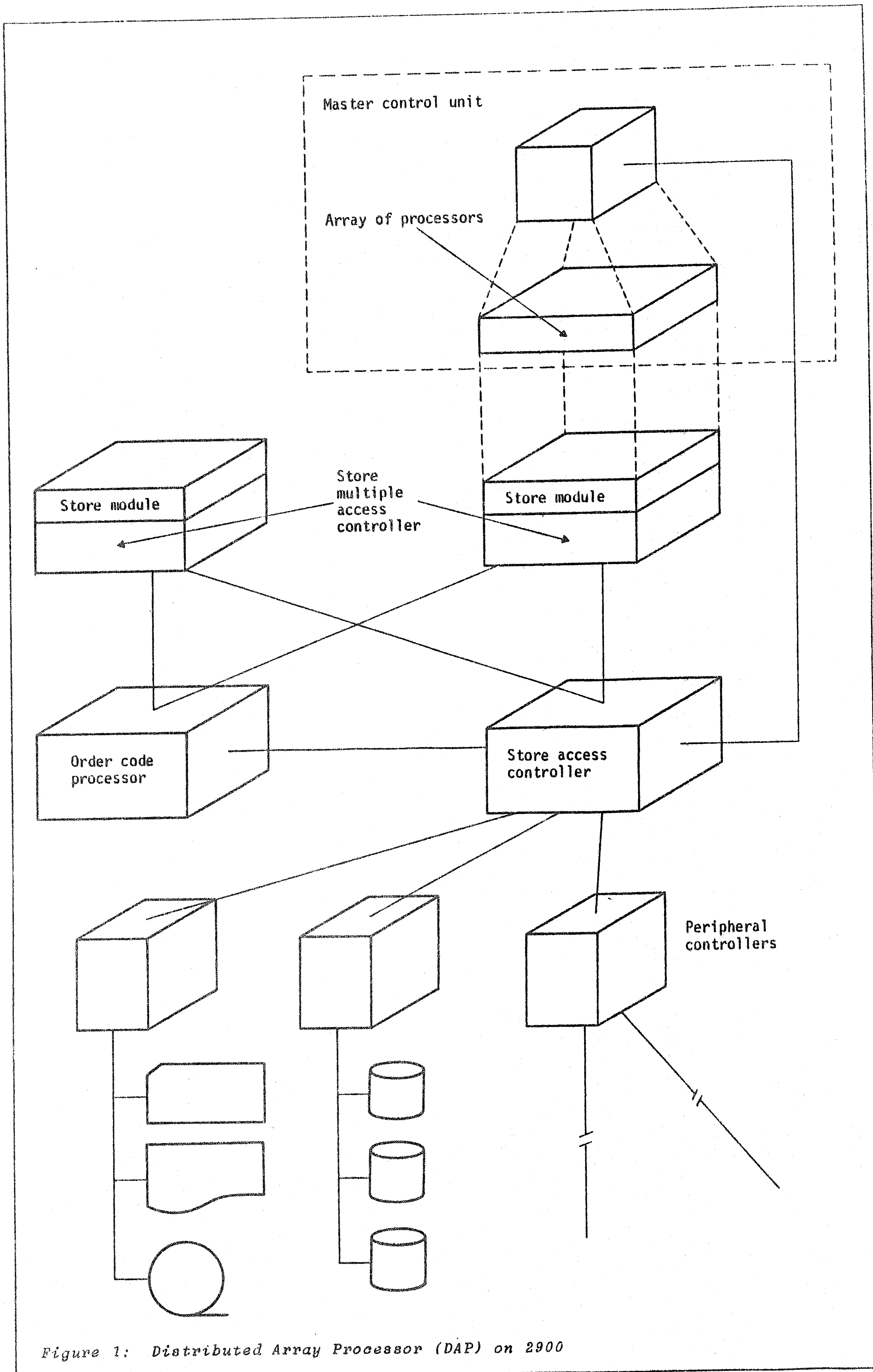


Figure 1: Distributed Array Processor (DAP) on 2900

more appropriate for some installations.

The processing elements themselves are highly flexible, elementary bit serial processors. A bit serial processor may be likened to a conventional processor whose word length, data paths and registers are all reduced to a single bit. Figure 2 shows schematically a processing element.

The processing elements all obey the same instruction broadcast from the master control unit. A typical instruction is of the type,

QS 57

which loads the Q register of every processor with the contents of bit 57 of its own

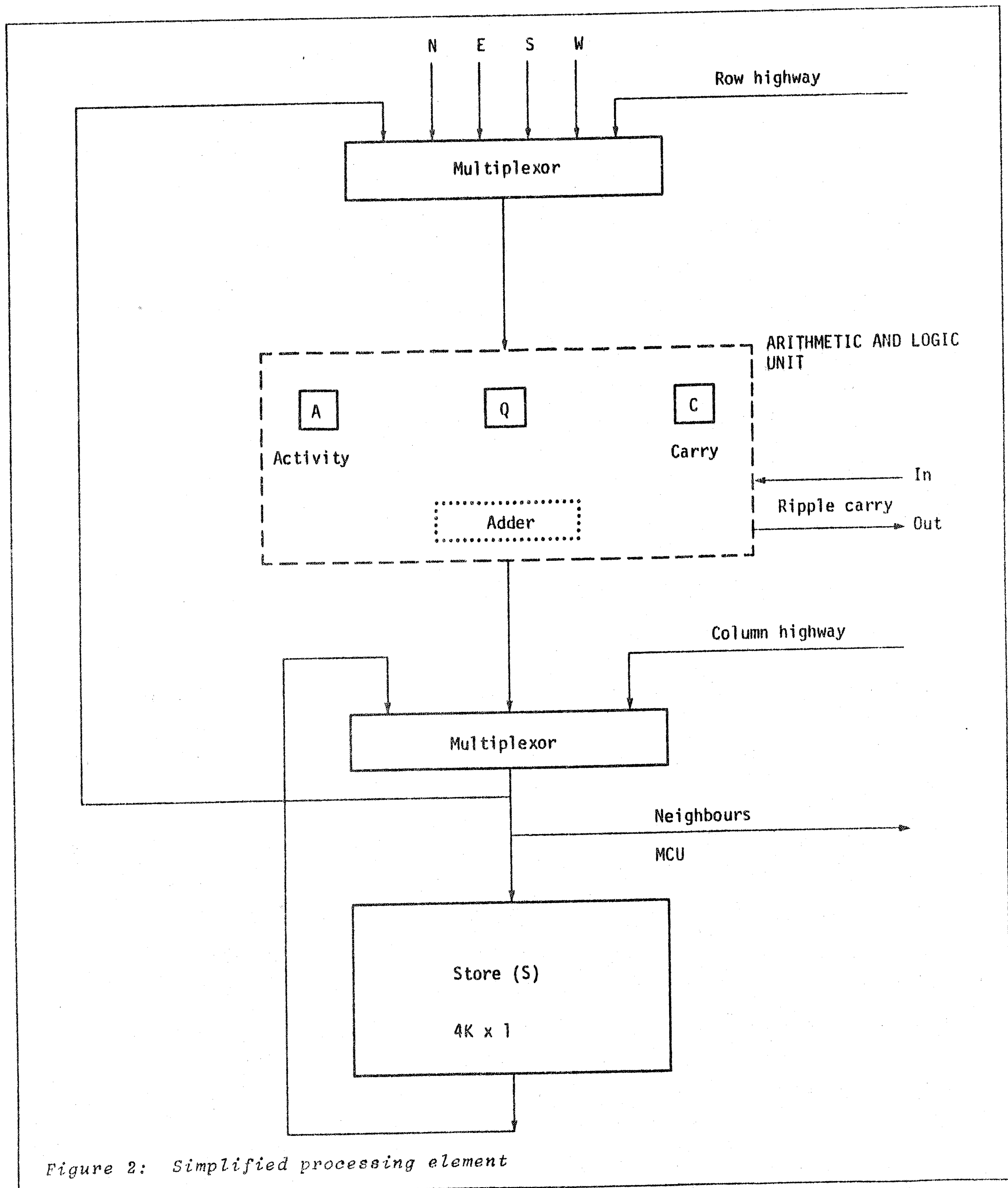


Figure 2: Simplified processing element

store. Loading may also take place from the store of any of the four nearest neighbours.

The inverse operation is the store operation,

```
SQ 25
```

which copies the content of the Q register into location 25. Writing is only permitted into the local storage.

A level of local autonomy is possible by use of the activity register which can be used to inhibit selectively writing to store. The instruction,

```
SQA 25
```

will only be effective in those processors where the activity register contains the value-true. The activity register can be set as a result of previous computations or imposed by the programmer. For example,

```
AQ
```

copies the contents of the Q register into the activity register whilst

```
AS MASK
```

would load the activity register from some bit plane with symbolic address MASK.

Register-to-register transfers are possible between neighbouring processing elements and such transfers form one of the means of routeing data between distant processing elements (see the section on Geometry for the effect at array edges).

All arithmetic operations are built up from elementary bit operations. The majority of computations consist of tight loops manipulating the successive bits of a number and for this purpose the master control unit supports a hardware assisted DO instruction.

A simple example of the use is to examine how the DAP performs the equivalent of the FORTRAN instruction X=Y, where we assume that Y is stored in bits 10-41 and X is to be stored in bits 51-82. The low level DAP code is:

```
   DO 32 TIMES
   QS 10 (+)
L: SQ 51 (+)
```

The label on the last instruction indicates the end of the DO loop. The (+) indicates that the address must be incremented by 1 each time through the loop.

The DAP would require 68 (32 x 2 + 4 overheads) cycles to execute the above loop and so on a model with cycle time 200 n seconds the time for the operation would be some 13.6 μ secs. A 64 x 64 DAP would be performing 4096 operations in parallel and so in 13.6 μ secs a 64 x 64 DAP would perform the equivalent of the following FORTRAN code:

```
   DO   1  I = 1,64
   DO   1  J = 1,64
1  X(I,J)  = Y(I,J)
```

The use of the activity bit may be demonstrated by considering the DAP code for the FORTRAN selective assignment,

```
   DO  1  I = 1,64
   DO  1  J = 1,64
1  IF (Y(I,J) . LT.O) X(I,J) = Y(I,J)
```

Assuming the same storage as the previous example we note that the first bit of Y gives its sign if either Y is an integer or a 2900 series floating-point variable. The DAP code is therefore:

```
   AS   10  ! SETS ACTIVITY TRUE FOR NEGATIVE
   DO   32 TIMES
   'QS  10 (+)
L: SQA 51 (+)
```

The time for this operation would be 14 μ secs.

Only one copy of the DAP instructions is held in the computer. The instructions are decoded by the Master Control Unit (described in the next section) and broadcast to each and every processing element. The instructions are stored in the DAP storage module interleaved amongst the processing elements in such a way that a program with M instructions will require $B = 32M/N^2$ bit planes on a N X N DAP. The effective space left for data would then be typically 4K - B bits per processing element.

THE MASTER CONTROL UNIT (MCU)

The major task of the master control unit is to broadcast to the array of processing elements the instructions to be obeyed. For this purpose the MCU contains many of the instruction generating facilities of a normal CPU (modification registers, address pointers etc). Figure 3 shows a schematic diagram of the master control unit.

An important feature of the master control is the instruction buffer which acts as an instruction slave for program loops, controlled by DO instructions.

The 8 MCU registers MO-M7 have the same number of bits as the array dimension (i e 64 bits for a 64 x 64 DAP). They are used for a number of different purposes.

The first purpose of the MCU registers is to act as instruction modifiers in a fashion similar to index registers on conventional computers. For example, the instruction sequence:

```
   LDSG  M1  X  ! LOAD M1 WITH ADDRESS OF X
   LDSG  M2  Y  ! LOAD M2 WITH ADDRESS OF Y
   DO    16 TIMES
   QS    15 (M1-)
L: SQ    15 (M2-)
```

copies a 16-bit field X starting at location X into the 16-bit field starting at Y. This use of the registers coupled with LINK, EXIT instructions gives a mechanism for subroutine execution.

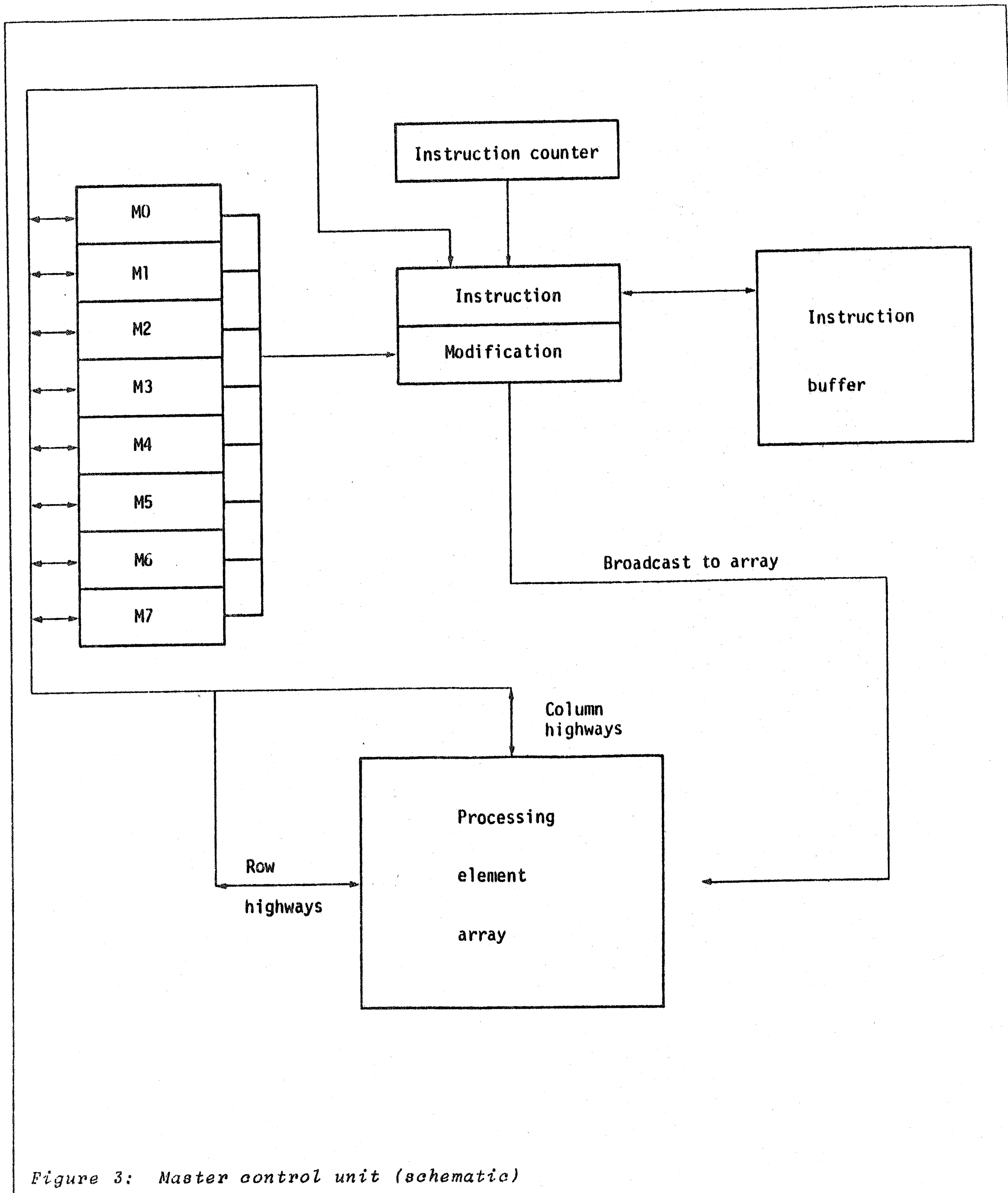


Figure 3: Master control unit (schematic)

An important group of facilities make use of the ability to read and write to or from the array of processing elements using the row and column highways.

Typical instructions for filling the MCU register from the array include

RSM M5 X

which will fill M5 with the AND of corresponding bits all the ROWS of the plane of data bits with symbolic address X. The equivalent operation on columns is RSO M5 X.

Many applications require the selection of data from a single row or column, and this is performed by

RSIM M6 0.5 (M5)

which fills M6 with the 6th row of the bit plane pointed at by M5. (The equivalent operation on columns is done with an RSIO instruction.) Instructions QRM sets each row of Q registers equal to the MCU register, and QRO does the same for each column.

A representative use of these functions is found in the problem of finding the element with largest modulus of an array pointed at by MCU register M1.

```
    AT  !  SET TRUE ALL ACTIVITIES
    QT
    DO  31 TIMES
    ALS  1 (M1+)  ! AND OF ACTIVITY
                      ! BIT AND A BIT OF X
    RANM M4       ! 'AND' OF INVERSE ACTIVITY BITS
                  ! BY ROW INTO M4
    SKPN M4       ! SKIP IF AT LEAST ONE A BIT IS ZERO
    AQ            ! RESTORE A IF NOT
L:  QA            ! SAVE A
```

On exit from this loop the activity bits indicate those processing elements containing the elements with largest modulus. Further manipulations then allow the value of the largest element to be extracted if needed.

ARITHMETIC OF THE DAP

As design of the DAP includes no specialized hardware for performing arithmetic operations, arithmetic (floating or fixed point) is implemented by means of standard library subroutines. A great advantage of this software approach is the ability to match the arithmetic precision to the user's problem requirement.

On standard computers arithmetic is normally only available in two precisions - single and double length. Those applications which require more than single length precision must be done at double precision with a consequential doubling of the storage required. The DAP allows a more flexible choice of precision, say a 48-bit format saving storage, so allowing more data values to be held in a fixed size memory and reducing backing store transfers.

Arithmetic operations are implemented using instructions of the type,

```
    CQP 105
```

which adds the bits in the Q register, the C (carry register) and position 105 together leaving the result in the Q and C registers.

Arithmetic on multiple-bit fields must be built up from the basic operations, as a trivial example we can consider part of an integer add routine for summing two 24-bit integer fields pointed at by M1 and M2 and putting the result into a 24-bit field pointed at by M3.

```
    CF  !  CLEAR CARRY REGISTER
    DO  24 TIMES
    QS  23 (M1-)  !  PICK UP A BIT OF FIRST OPERAND
    CQP 23 (M2-)  !  ADD IN THE SAME BIT OF 2ND OPERAND
L:  SQ  23 (M3-)  !  STORE PART OF RESULT
```

A proper routine would incorporate a few extra instructions to be added to detect, and signal overflow conditions.

Since arithmetic operations are built up by program out of the basic one-bit operations, it follows that the DAP has no special number representations and may be programmed to work at any desired precision etc. In particular, floating-point operations are not limited to, say, base 16 exponents but may use other exponents (e g base 2) as appropriate to the application. Similarly the number of bits in the mantissa and the round-off/truncation philosophy may be varied to suit the problem.

The immediate effect of the software nature of arithmetic is the fact that it is impossible to quote hard and fast operation times for what are normally considered to be the elementary floating-point arithmetic operations. Even if one picked a standard data format, the matter is still complicated as, for example, the fact that algorithms for squaring the contents of the processing elements or multiplying every processing element by the same value will be faster in execution (by factors of two or more) than the algorithm for multiplying two general values.

Figure 4 indicates the approximate times for a number of typical operations expressed as the equivalent FORTRAN operations. These times are upper limits as better algorithms or implementations may be discovered.

Statement	
X(I,J) = Y(I,J)	13 μsecs
X(I,J) = Y(I,J) + Z(I,J)	135 μsecs
X(I,J) = C * Y (I,J)	100 μsecs
X(I,J) = Y(I,J) ** 2	140 μsecs
X(I,J) = Y(I,J) * Z(I,J)	280 μsecs
X(I,J) = Y(I,J)/Z(I,J)	350 μsecs
X(I,J) = SQRT(I,J))	200 μsecs
IF(X(I,J).GT.XMAX) XMAX = X(I,J)	50 μsecs

Explanation

The above are the total times for the equivalent of the FORTRAN

```
    DO 1 I = 1, N
    DO 1 J = 1, N
    STATEMENT
1   CONTINUE
```

When executed on a N X N DAP

Figure 4: Execution times for some typical Array Mode operations (2900 32-bit floating-point format assumed)

MAIN STORE MODE VERSUS ARRAY MODE

In the previous discussion it has been implicitly assumed that all bits of a variable are stored in successive bits of the same processing element: this mode of storage is designated *array mode* working. There is no hardware limitation that this must be so and it is often advantageous to store successive bits of a variable at the same location in a row of processing elements. This mode of storage is known as *main store mode* as it is the natural way for the host computer; semiconductor stores are built with the bits of each word interleaved through a number of semiconductor store chips in order that a whole word may be extracted in one access cycle of the store chip.

The DAP may be caused to process data in main store mode or in array mode. The difference being that, for a N X N DAP working on P bit precision data, main store mode has up to N^2/P degrees of parallel operations but each operation effectively works on all bits of the variables in parallel. Array mode working has N^2 degrees of parallelism but the bits of the word must be processed serially. The choice of mode depends on the degree of parallelism appropriate at each stage of a computation.

The DAP hardware supports a number of functions to facilitate main store mode working, the most important of which is the ripple carry indicated in Figure 2 which allows carry operations to be rippled through at least four processing elements per DAP cycle.

The operation times for main store mode arithmetic are in general faster than those for array mode by a factor of three to six, the factor being strongly dependent on the operation.

In order that array mode computation can be carried out, a transposition of input data from main store mode to array mode is required. This operation is very rapid on the DAP. The time is dependent on the amount of data to be transposed but is normally a trivial overhead when performing a reasonable amount of computation.

Figure 5 shows the relationship between the storage modes for a schematic 4 x 4 DAP, where the host computer has a 4-bit word length.

GEOMETRY

Many problems require certain edge conditions to be applied to the array to model the exact physical conditions. This is accommodated on the DAP by controlling the connection of the processing elements at the edge of the array, using a geometry register in the MCU. By suitable connections the geometries shown in Figure 6 may be modelled.

THE OVERALL PERFORMANCE

Array processors or associative processors, such as the DAP, are difficult computers to assess and great caution has to be applied in defining the measures as most classical measures are inappropriate. There are computational techniques, e g, list programming, which are specifically based on non-parallel processing and for which array processors will show up very badly. There are other computational techniques, e g, finite different methods for solving partial differential equations, which are perfectly adapted to array processors. There are other computational problems, e g, finding the

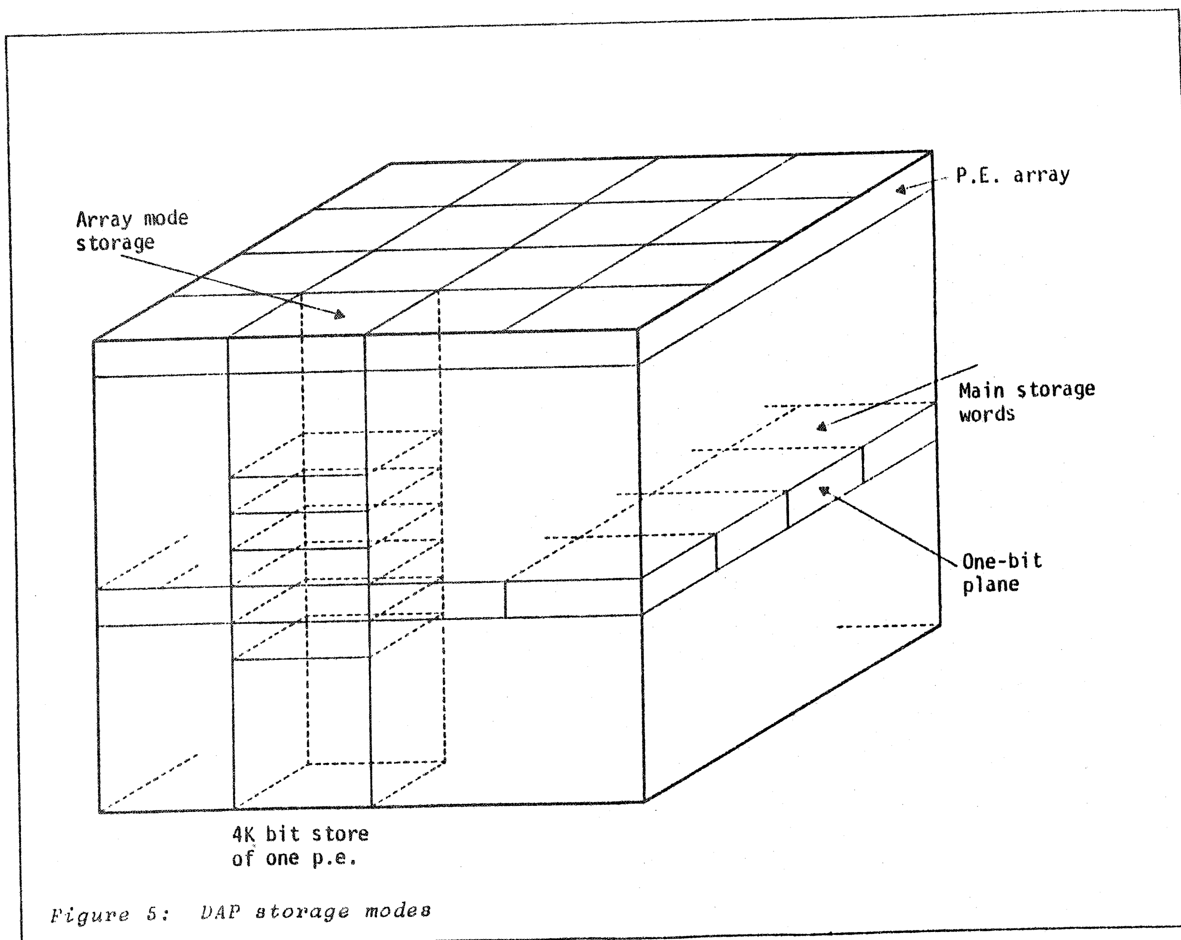


Figure 5: DAP storage modes

largest element of a set, which are ideal for an associative processor. Any total application consists of some mixture of elements adaptable in various proportions to each type of processing technique and the total performance is highly dependent on the appropriate mixture.

The reader is strongly cautioned against making too many deductions about performance from parameters which represent only one of the above factors, such as the multiplication time or the search time. Too literal interpretation of these figures may lead to errors: in the early days of computers it was popular to race computers against a man with an abacus - for adding ten numbers the man always won! There was, of course, a lesson to be learnt from the race (don't forget I/O), but the uninformed conclusion was that computers really were no good. When applied to the type of computation for which it is designed, the DAP has a potential performance many times that of the fastest computers.

For example, for matrix manipulation computations the 32 x 32 Pilot DAP model runs between 1 and 3 times the speed of a CDC 7600. Studies suggest that a 64 x 64 DAP will perform between 4 and 10 times better than a 7600 on suitable floating-point limited technical calculations and possibly up to 100 times on problems which exploit the associative nature of the store.

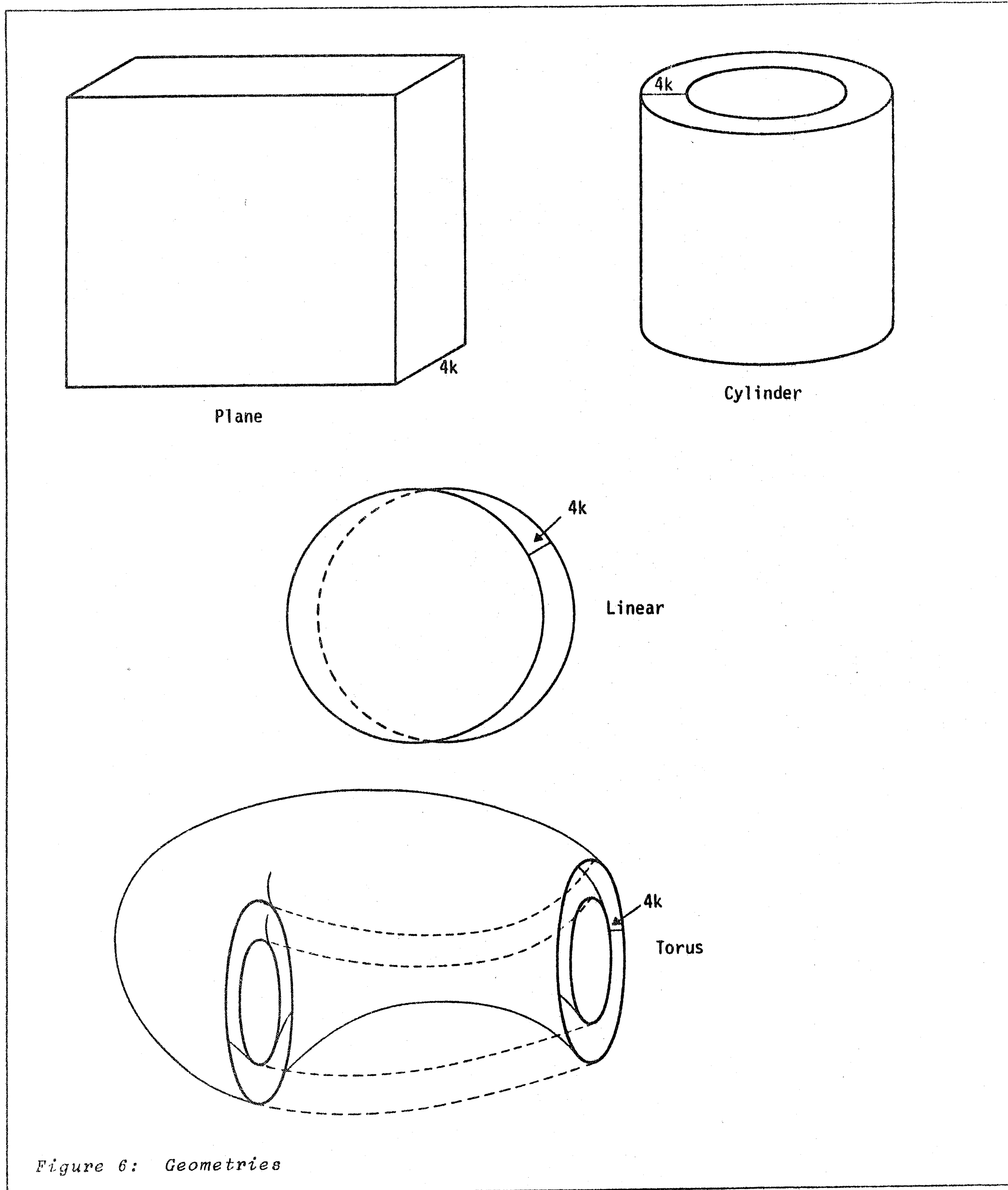


Figure 6: Geometries

1st Annual Symposium on Computer Architecture, Florida, Dec 1973

DAP—A DISTRIBUTED ARRAY PROCESSOR

Dr. S. F. Reddaway
Language and Processor Department
Research and Advanced Development Centre
International Computers Limited

ABSTRACT

An array of very simple processing elements is described each with a local semiconductor store. The array may also be used as main storage.

Bit-organisation gives great flexibility, including the minimisation of word length. Use of MSI and LSI is helped by the simplicity of the serial design. Using 15-bit fixed point, the theoretical performance of a 72 x 128 array is about 10^8 multiplications or 10^9 additions per second. Comparisons are made with other architectures.

Meteorology is considered as an application. It is attractive to have the whole problem in the array storage.

1. INTRODUCTION

This paper describes a design study of an array of elements that can be used either as a "Single-Instruction, Multiple-Data stream" (SIMD) processor or as a store. Architectural features of interest are: (a) the use of serial arithmetic to simplify processor logic and optimise store utilisation; (b) an attempt to avoid I/O bottlenecks by mapping complete problems into the array, without relying on overlay techniques; (c) provision for using all or part of the array as a store when not performing its specialised processing functions; (d) the close integration of storage and logic.

The main attractions of array-type SIMD structures are: (a) high absolute performance on certain problems of importance; (b) high performance/cost, partly resulting from using common control logic.

Several examples of this type of architecture have been proposed (1-8) and applications have been suggested in, for example, meteorology, plasma physics and linear programming. Most structures have a single control unit that broadcasts instructions to a regular array of processing elements (PEs) each with individual storage and an arithmetic unit (AU).

Flynn (2) points out four factors that degrade the performance from the theoretical figure given by "Number of PEs times PE performance": (a) Each PE has direct access only to a limited region of store, and excess time may be taken accessing other regions; (b) Mapping the problem onto the array may leave some PEs unused; (c) Owing to overheads in preparing instructions for the array, there may be times when the whole array is idle; (d) While dealing with singularities or boundary conditions the majority of PEs are idle.

These factors are acknowledged to reduce the applicability of such an array. In the present design attempts have been made to mitigate their effect, but the over-riding consideration has been to simplify the PE design; this has been done to the extent that the theoretical performance is very high, in spite of the AU cost being small compared with that of the storage. In effect, therefore, the store is being adapted to an array processing function. This may be contrasted with attempts to adapt the processor to array operations (e.g. CDC STAR).

A dispersed system, i.e. one with many PEs each with local memory, has potential cost and speed advantages deriving from: (a) reduced "cable" delays; (b) reduced address transforming and checking; (c) faster actual access; (d) simplified data routing and priority logic.

A number of potential PE designs of varying parallelism have been considered for building arrays of the same theoretical performance, with the following general results.

The gate count varies with the degree of internal PE parallelism. A purely serial PE has considerable advantages particularly for low precision work.

Serial PEs have fewer connections at all packaging levels.

The extreme simplicity of serial PEs permits the very effective use of batch fabrication and testing techniques and keeps hardware development rapid and cheap. The small number of circuit and board types helps development, production, spares holding and maintenance.

Serial designs have exceptional functional flexibility; very few decisions are built into the hardware. However, fully indexed addressing is expensive.

The design is somewhat similar to SOLOMON 1 (8); the main differences stem from the exploitation of modern technology.

2. THE ARRAY

2.1 CONFIGURATION

FIGURE 1. M.C.U. DIAGRAM

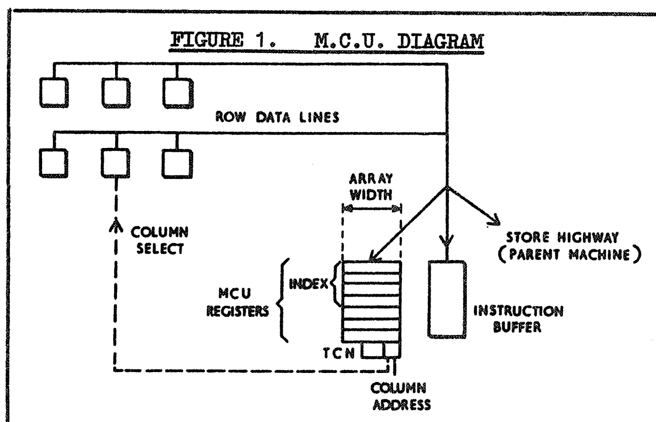


Figure 1 is an overall configuration diagram. The rectangular array has an essentially two dimensional nearest neighbour connectivity, and has one dimension matched to the store highway of a conventional computer (the "parent" machine). This connection provides the route for loading both data and array instructions into the array storage for array processing; it also permits the parent machine to use the array storage as its own main storage. Input/output is done by the parent machine.

The Main Control Unit (MCU) has: (a) a conventional instruction fetching arrangement; (b) an instruction buffer whose purpose will be described later; and (c) a set of registers, many of which can be matched to the array by row or column for a variety of purposes, one of which is indexing. For sizable arrays the MCU is a very small fraction of the total hardware.

After loading, the bits of a word are spread along a column of PEs, and this method of holding data is termed Main Store mode. Another method, termed Array mode, stores all the bits of a word in a single PE. This is more attractive for processing large arrays, but requires initial and final transformation of the data from and to Main Store mode; this is done inside the array.

2.2 THE PE

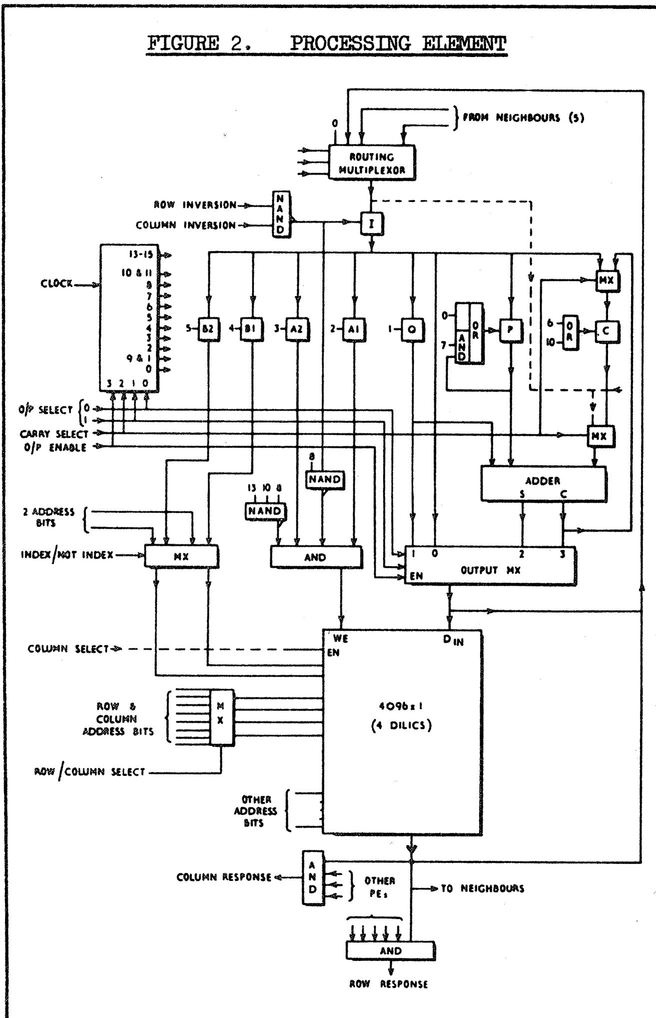


FIGURE 2. PROCESSING ELEMENT

Figure 2 is a PE diagram. The registers are all one-bit; P and Q are for operands, C is the carry register, A1 and A2 are activity bits that can prevent writing to store, and B1 and B2 can supply 2 address bits. The routing multiplexor can select a bit from the PE's own store, or from a neighbour's store, for writing to a register; selecting zero and controlling its inversion permits data input from outside the array (for example, an MCU register). The sum, carry, data input or contents of Q can be output from the logic, usually to the store. The store contents can be output externally (to, for example, an MCU register) via the gates at the bottom of Figure 2; the bits output can be either from a selected column of PEs, or the logical AND of rows (or columns) of PEs. One use for the latter is for a test over all PEs.

The fifth "neighbour" connection is to the PE half a row away in the same row; this permits both faster mass movement of data around the array, and a "2½D" PE geometry. Bit patterns in one or two MCU registers can be applied to the "inversion" inputs to produce a veto selective by rows and/or columns on writing to PE stores. Figure 2 shows 4 address bits capable of being selected by row or column; what indexing facilities should be provided is still an area of debate.

Some differences from the PE in (7) are: (a) more row/column symmetry; (b) a latch feature (shown on the P register) for associative comparisons; (c) data can be shifted directly between PEs without using the store; (d) input data can be loaded directly into store; (e) there is a ripple carry path between PEs for Main Store mode arithmetic; (f) the bipolar store is now 4K instead of 2K.

It is intended to package 2 PEs minus their stores and routing multiplexors in one 24 pin integrated circuit.

2.3 EDGE CONNECTIONS

For instructions that involve neighbours, it is the array geometry that determines what happens at the array edges. Rows or columns may be: (a) cyclic, with their ends connected together; (b) linear, with a continuation onto a neighbouring line; (c) as (b) but with the extreme ends connected; or (d) plane, with external data applied at the relevant edge. In addition, a row may be considered in two halves (2½D geometry). There are thus 32 geometries, and they are set by program.

2.4 CONSTRUCTION

A board would contain a 6 x 4 PE section with 4K bits/PE; there would be 137 external connections and 173 ICs, 96 of them for storage. The array can be viewed as doing processing in the store, and costs only about 25% more than ordinary storage made out of the same technology. A platter would contain a 36 x 16 PE section; the number 36, and multiples of it, match standard store highways. "Folding" of the array makes connections between the extreme edges short.

The economy obtained by the dense packing of the integrated circuits is the result of the favourable marriage of space-limited (or power-dissipation limited) storage and pin-limited logic.

2.5 TIMING

Because most micro-instructions do not involve a response from the array, the equalisation, rather than minimisation, of delays is important. Even with a comparatively slow logic technology, the micro-instruction rate should be about 5-6 MHz; the storage

element delays are the biggest factor, and this illustrates how the array can exploit bipolar store speeds, unlike a large conventional machine.

2.6 FUNCTIONS

In (7) the basis of the micro-programming notation is given and it is shown how Array mode fixed and floating point instructions are built-up. Bit organisation means that only necessary work need be done; for example, multiplication only needs to calculate a single length result.

Code for execution must be compiled down to the one-bit micro-instructions, except that for working regularly along the bits of words a short loop can be constructed. This loop is held in the instruction buffer, so that no further instruction fetching from the array storage is needed during execution of the loop. This feature reduces the instruction fetching overhead from 100% to about 20%. Subroutine construction will be possible.

2.7 PERFORMANCE

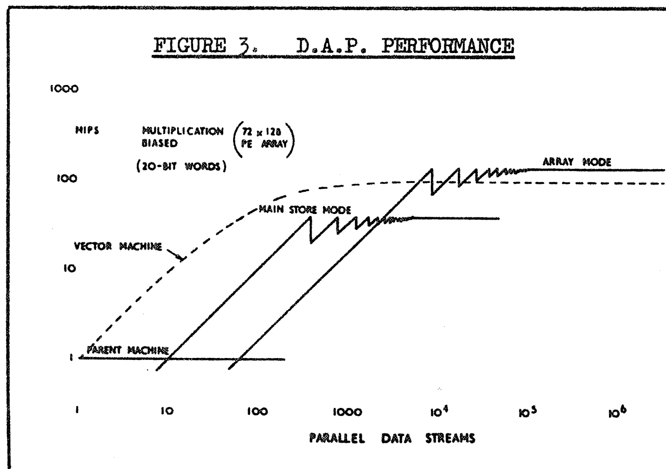
For array mode, fractional fixed point multiplication takes about

$$\frac{n\ (3n + 13)}{2}$$

micro-instructions where n is the word length; fixed point addition takes little more than 3n micro-instructions. Floating point takes a little longer for multiplication, and considerably longer for addition (see (7)). 20-bit multiplication takes about 730 micro-instructions plus about 160 cycles for micro-instruction fetching, and at $5\frac{1}{2}$ MHz would take about 160 μsec; 20-bit addition takes about 12 μsec. Multiplication of an array by a common number can be about four times faster.

Main store mode arithmetic is faster than Array mode for smaller arrays. In terms of absolute speed, addition is about 11 times faster and multiplication, using a carry save technique ending with a ripple carry, is about six times faster for 20 bit precision (the latter factor increases with the precision).

FIGURE 3. D.A.P. PERFORMANCE



The user has three modes of working at his disposal: the parent machine for scalar working, Main Store mode for small arrays and Array mode for large arrays. Figure 3 shows roughly what is possible in the three modes; the useful processing rate in Million Instructions (or, more accurately, results) Per Second (MIPS) is plotted against the number of parallel data streams for the type of computing indicated and a 9200 PE array. Only the top ends of the sloping lines depend on array size. The dashed line shows the similar graph for a powerful vector machine (there are many other differences between the two types of machine).

The overall performance depends on the application and programmer skill.

2.8 A COMPARISON

ILLIAC IV is a well known machine, so a brief comparison is attempted with Array mode, assuming the problem parallelism is sufficient to occupy either machine. Many differences are not easily quantifiable, but as a starting point the main assumptions for a numerical comparison are given in Figure 4. The first four lines give the instruction mix; B is the number of bits precision for the serial design, which has no separate store acesses because all functions are store-to-store. P is the clock period (180 nsec). 20% is subtracted from the ILLIAC IV totals to allow for instruction overlap.

FIGURE 4. DESIGN COMPARISON
ILLIAC IV ASSUMPTIONS

INSTRUCTION MIX AND TIMINGS:

	SERIAL DESIGN	ILLIAC IV SINGLE PRECISION	ILLIAC IV DOUBLE PRECISION	ILLIAC IV TRIPLE PRECISION	
1 ADD/SUBTRACT	$(2+3B)P$	0·125	0·25	0·5 ?	μsec
1 MULTIPLY	$(4B+1{\cdot}5B^2)P$	0·25	0·5	2·0?	μsec
2 STORE ACCESSESS	0	0·325	0·65	1·0?	μsec
1 MODE SETTING (Etc.)	4P	0·05	0·05	0·05	μsec
TOTAL	$(6+7B+1{\cdot}5B^2)P$	0·75	1·45	3·55?	μsec
TOTAL -20%		0·6	1·16	2·8?	μsec
MANTISSA		25	49	73	BITS
EXPONENT		7	15	(23)	BITS
"USEFUL" EXPONENT		4	6	8	BITS

LOGIC/PE.
ILLIAC IV ~12000 FAST ECL GATES
SERIAL DESIGN ~60 TTL GATES
1 FAST ECL = 2TTL GATES
RATIO = 200 x 2 = 400

Figure 5 compares the hardware required to build an array of given performance for words of a particular precision. Logic and storage have equal weight; Figure 4 gives the gates/PE ratio and the storage comparison involves an estimate of the unnecessary bits in the ILLIAC IV word. The graph would favour ILLIAC IV only for working exclusively with 46-49 bit precision. At low precisions serial PEs have a very big advantage.

Such numerical comparisons are of only limited value. For example, the vertical scale of Figure 5 would be multiplied by about 4 if integrated circuit count were used as a hardware measure. Other factors such as hardware simplicity and repetition, pin counts and functional flexibility are equally important.

2.9 EXAMPLE OF STORAGE ECONOMY

For problems with large amounts of data, storage economy is important, particularly if it permits storing the complete problem in the array. The user can apply various tricks. As an example, consider three dimensional field problems. In order to prevent physical "truncation" errors, programs are designed so

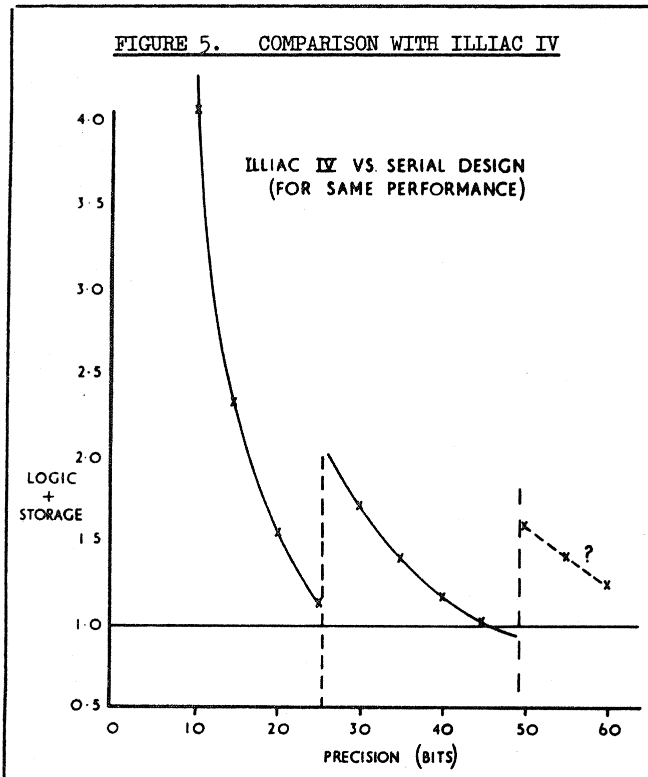


FIGURE 5. COMPARISON WITH ILLIAC IV

that differences between neighbouring variables require fewer significant bits than the variables themselves. If variables have to be held simultaneously for two time steps, then, for example, they can be grouped into sets of 16 nearest neighbours in space and time (2 x 2 x 2 x 2), and held as follows: (a) a short floating point number close to the maximum of the group (maybe a 4-bit mantissa and 3-bit exponent); and (b) 16 differences in block floating point (maybe 12-bit mantissas and a common 2-bit block exponent). This results in 12.6 bits/variable and is roughly equivalent to floating point with a 15-bit mantissa and 3-bit exponent, i.e. a gain of nearly 50%; other machines require floating point variables to occupy up to 64 bits, i.e. up to 5 times more.

3. METEOROLOGY AS AN APPLICATION

This is considered more fully in (7). Meteorology includes both simulation experiments and forecasting, and as simulation programs are central to both, attention will be confined to them. (Forecasting also uses analysis and initialisation programs to assimilate the "real" data). For simulation programs, the frequency of add/subtract and multiply instructions is roughly equal, and divide is much less frequent. For DAP, multiplication takes much longer than addition, so the number of multiplications and their timing give a first approximation to the speed of a program.

The table gives a rough guide to parameters in use today and those that should be aimed at.

Using the 18 bit (fixed point) precision suggested in Section 3.3, each PE can perform a multiplication in about 140 μsec. Section 3.2 discusses the efficiency of PE usage; 50% might be a reasonable figure. Thus about 8000 PEs are adequate to perform the 2.5×10^7 multiplications per second indicated above.

TABLE

	Present		Next stage
	Forecast Programs	Global Research Programs	
Number of Vertical Columns of Grid Points	3000	10 000	x 4
Number of vertical levels	10	5	x 2
Total number of variables	2×10^5	2.1×10^5	x8 (1.6×10^6)
Time step	2 min.	5 min.	÷ 2
Number of time steps	1000	10 000	x 3
Multiplications per column per time step	1000	500	x 2.5
Multiplications/sec.	1.2×10^6	1.2×10^6	x20 (2.5×10^7)
Speed-up over real time	50-100	50-100	50-100

3.1 STORAGE

It may be tempting to use a backing store for big problems; however, the smaller the array storage the larger is the channel capacity required. In (7) an example was studied of a problem using explicit integration which had 1.5×10^6 variables of average length 20 bits, and was processed on an 8200 PE array with an I/O channel of 10^7 bits/sec. Three formulations of the problem had the following trade-offs: (a) 1850 bits/PE and speed degraded by a factor of 2.5, (b) 2800 bits/PE and speed degraded by 1.3, and (c) 4600 bits/PE, the complete problem in the array and no degradation. A similar problem using implicit methods would have its speed degraded by an order of magnitude if a backing store was used.

This sort of problem needs about $5\text{-}10 \times 10^7$ bits of storage. The falling cost of semi-conductor storage makes this amount of array storage feasible, and the simplicity and reliability of a unified semi-conductor system makes it attractive. Partly for these reasons, the array has more resources devoted to storage than to logic.

3.2 PARALLELISM

Efficiency, defined as the fraction of time a PE is active, depends on programmer skill as well as the problem. Numerical procedures used at present have usually been devised with serial machines in mind, and sometimes a slightly different procedure may be much more efficient.

Explicit methods for the "basic" meteorological equations are efficient. Boundaries do not have much effect because it is usually a case of omitting things. "Secondary" effects may cause efficiency to drop. The computation is different if the air is saturated.

Convection may require the checking of neighbouring vertical layers for stability, followed by a relaxation process. Study indicates that these effects need not have a major effect on the overall efficiency.

Once various conditions have been established "branching" by means of activity bits is very rapid, and can be done frequently in order to improve parallelism. (A conditional branch in a conventional program loop, or selection in a vector machine, are slow by comparison).

Implicit methods involve either ADI (alternating direction implicit) or relaxation methods; the former are not particularly efficient but the latter are.

There seem to be 4 types of grid in use: (a) rectangular for fairly local forecasts; (b) octagonal in overall shape (rectangular neighbour connection) for the northern hemisphere; (c) cylindrical on a global latitude-longitude basis; (d) as (c) except that the number of points on a line of latitude is reduced as the poles are approached. (a) and (c) can fit a rectangular PE array. (b) and (d) would waste some of the PEs. (c) has reduced efficiency because a smoothing process is applied more times near the poles; this can be viewed as a trade-off for the wasted PEs of (d).

3.3 PRECISION AND NUMBER REPRESENTATION

Precision costs time and storage space, so that big problems should use only the minimum consistent with accumulated round-off error being small compared with other errors. Different variables can use different number representations and precisions. Knowledge of requirements is only patchy, but should improve; the pay-off, compared with fairly cautious starting schemes, might be a factor of about 1.5 in storage and 2 in speed.

Meteorology is largely concerned with absolute rather than relative accuracy, and the maximum possible values of variables are well understood; this points to either fractional fixed point or a simple floating point. Block-floating of arrays (9) can also be implemented efficiently.

An example of possible economy in space and speed occurs in explicit integration schemes; the increments to variables require considerably less precision than the full variables.

Careful choice of rounding method in order to avoid bias can also lead to economy (7).

A reasonable estimate of the average precision required for fractional fixed point variables might be 18 bits and rather less for the mantissa of floating point variables.

4. OTHER APPLICATIONS

An algorithm to solve the two dimensional Poisson's equation was studied. It used a Fast Fourier Transform technique, but the extensive data shuffling that this involved occupied only 20-25% of the time. There was also reduced parallelism in places, and a typical PE was idle about 50% of the time. On a 72 x 64 PE array, a 256 x 256 mesh was estimated to take 50 msec for 20-bit numbers; this compares very favourably with conventional machines. An interesting aspect is that the main array is held in Array mode and certain row and column features are dealt with in Main Store mode; Main Store mode vectors are combined with the array elements in single arithmetic operations.

For the array to be useful, problems must fulfil three conditions: (a) Processing, as opposed to I/O, must be important; (b) Much of the problem must be programmed with parallel and identical operations (these may, however, be selective); (c) Excessive time should not be spent shuffling data round the array. (In some cases this means the data should be fairly regular).

These requirements are not very severe, and the biggest barrier to widespread use is likely to be in devising an acceptable programming language. (In spite of many problems being naturally parallel, many users are indoctrinated by sequential thinking).

Some applications for array processors are discussed in (5). Further applications are suggested by the fact that the array can be used as an "associative processor"; examples might be air traffic control, graphics processing and symbol processing. Associative information retrieval can look attractive over quite a wide range of parameters; with the associative latch, each PE can scan 1 bit every micro-instruction, and so 10 000 PEs can scan 5×10^{10} bits/second.

The user has the freedom to optimise and experiment from the bit level upwards; this may help him understand his real computing requirements. The array is not arithmetic biased, and the functional flexibility permits functions to be tailored for all sorts of purposes. The hardware simplicity permits parameters such as the number of bits/PE and the type of storage to be varied easily; for example, a slower, cheaper MOS version would extend the range of applications considerably. The array modularity (almost like storage modularity) means that sizes from 500 to 30 000 PEs are reasonable.

ACKNOWLEDGEMENTS

The author would like to thank the Directors of ICL for permission to publish and J.K. Iliffe for his support and for originating many of the ideas. The contribution of A.W. Walton is also gratefully acknowledged.

REFERENCES

1. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., and Stokes, R.A. "The ILLIAC IV Computer", IEEE Transaction on Computers, C-17, p. 746 (1968).
2. Flynn, M.J., "Some Computer Organisations and their Effectiveness", IEEE Transactions on Computers, C-21, p. 948 (1972).
3. Goodyear Aerospace "STARAN - A New Way of Thinking". A Goodyear Aerospace brochure, Akron, Ohio (1971).
4. Huttenhoff, J.H., and Shively, R.R. "Arithmetic Unit of a Computing Element in a Global, Highly Parallel Computer", IEEE Transactions on Computers, C-18, p. 695 (1969).
5. Kuck, D.J. "ILLIAC IV Software and Application Programming", IEEE Transactions on Computers, C-17, p. 758 (1968).
6. Murtha, J.C., "Highly Parallel Information Processing Systems" in "Advances in Computers". Vol.7, (1966).
7. Reddaway, S.F., "An Elementary Array with Processing and Storage Capabilities", International Workshop on Computer Architecture, Grenoble, June 1973.
8. Slotnick, D.L., Borck, W.C., and McReynolds, R.C., "The Solomon Computer", Fall Joint Computer Conference 1962, p. 97.
9. Wilkinson, J.H., "Rounding Errors in Algebraic Processes", H.M.S.O. London (1963).

TECHNICAL REPORT
REF: TR/7/ISSUE 1

High Performance Linear Algebra on the AMT DAP 510

Summary: The performance on a wide range of linear algebra can be greatly increased by defining at a high level a family of new subroutines and implementing them efficiently. The subroutines are essentially rank k updates to matrices, and have been used in matrix multiplication and equation solving. Performance improves more than six fold up to 50 MFLOPS in favourable cases.

This paper is to appear in the book of the December 1987 SIAM conference of Parallel Processing for Scientific Computing.

Author: Stewart F. Reddaway, Grant Bowgen, Sven van den Berghe
Date: 20 January 88

Distribution: Sofware Group and friends
G Manning
R Hornstein
R Humpleman
C Winckless
D J Hendley
B Alper
W Terry
J Litt
I Harding

High Performance Linear Algebra on the AMT DAP 510

STEWART F. REDDAWAY*, GRANT BOWGEN**, AND SVEN VAN DEN BERGHE**.

Abstract. The performance on linear algebra of bit-organised SIMD arrays such as the AMT DAP 510 can be greatly increased by defining at a high level new subroutines and implementing them efficiently. The subroutines are essentially rank k updates to matrices, and they exploit the fact that one vector is multiplied by a succession of numbers from another vector. The subroutines have been used in implementations of matrix multiplication and equation solving. Performance improves more than sixfold (to 50 MFLOPS) in favourable cases.

1. Introduction. SIMD arrays of simple bit-organised Processing Elements (PEs) perform floating point arithmetic by low level system software. The AMT DAP 510 produces an array of 1024 32-bit floating point results in about 1000 cycles of 100 nsec each to give a performance of about 10 MFLOPS.

This paper describes the definition of a family of subroutines at a higher level than single array operations, and their efficient implementation. They can be used to greatly increase the performance of many linear algebra problems, and example implementations of matrix multiplication and equation solving are described.

The DAP (Distributed Array Processor) has been described in many papers as it has evolved over the years; for example references 2, 3, 4 and 5. This paper refers to the current product, the AMT DAP 510 [1]. This has an SIMD array of 1024 PEs, and with arithmetic performed by low level software the time an operation takes is related to its complexity. Thus Boolean array operations are very fast (about 10,000 MOPS) but floating point arithmetic is much slower, with the most relevant 32-bit operations performing as:

	Cycles	MFLOPS
vector - vector:		
Add	860	12
Multiply	1400	7
Average Add, Multiply	1130	9
scalar - vector:		
Multiply	800	13

* 3 Woodforde Close, Ashwell, Baldock, Herts. SG7 5QE, England
** Active Memory Technology Ltd., 65 Suttons Park Avenue, Reading RG6 1AZ, England

The average is given because linear algebra uses an equal mix of add and multiply. Many other operations are much faster, and we will exploit integer add being much faster than floating point add. Every floating point precision from 24-bits to 64 bits in 8-bit intervals is available as standard from the high level language (FORTRAN-PLUS), with a smooth trade-off against speed and storage space.

2. Performance of Machines on Linear Algebra.

It is well known that on linear algebra problems, such as equation solving with pivoting, the measured MFLOPS of a machine typically falls a long way short of the rated peak MFLOPS. Without the use of the new subroutines of this paper, the DAP programmed (using a 2D mapping) in FORTRAN-PLUS on problems of order 100 - 1000 has system efficiencies in the ranges:

Matrix multiplication	60 - 90%
Equation solving	35 - 75%

This is good by industry standards, but can we do better still?

3. New Subroutines.

Most work in linear algebra can be expressed in terms of matrix multiplication (i.e. as an accumulation of outer products of pairs of vectors). This occurs, for example, in what is referred to as rank k updates of a matrix. The family of basic subroutines is:

MMk	"Matrix Multiply"
MMAk	"Matrix Multiply and Add"
MMSk	"Matrix Multiply and Subtract"

k gives the common dimension of the matrices being multiplied, e.g. 1 (outer product), 8, 16.

An example of calling a subroutine is:

```
CALL MMS8 (AVS, SCVS, VS, N, LM)
```

The Nx8 matrix SCVS is multiplied by the 8x1024 matrix VS and the result subtracted from an Nx1024 matrix AVS masked by LM, a 1024 element logical vector. SCVS and VS are sets of 8 DAP "long vectors" and AVS is a set of N long vectors; each long vector contains up to 1024 elements. N is up to 1024. The 1024 dimension in the above matrices can be converted into a dimension L (up to 1024) by LM containing L consecutive TRUE bits. (Other uses may be found for LM having arbitrary patterns.) The structure of MMS8 is shown in Fig. 1.

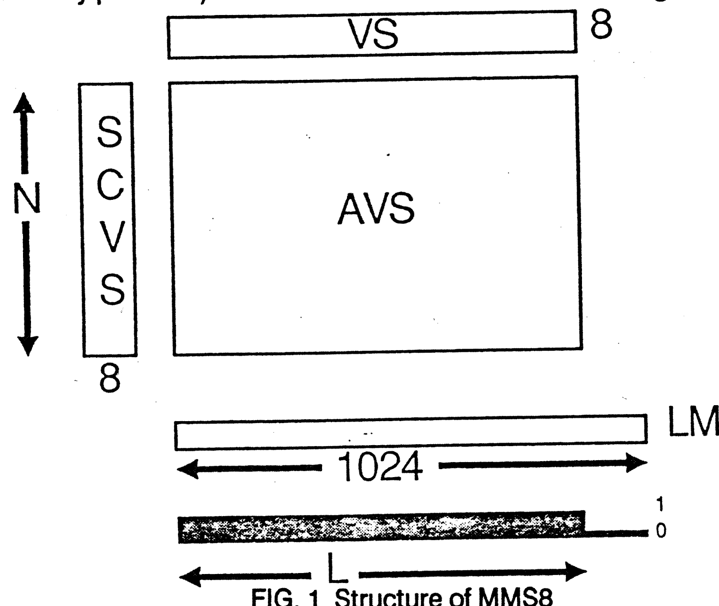


FIG. 1 Structure of MMS8

4. Ideas used inside the new subroutines. Multiplication can be speeded up by noting that in an outer product one vector is multiplied by a succession of scalars from the other vector, and doing appropriate preparation work on both vectors. Many (specifically 32) multiples of the vectors in VS are generated. The numbers in SCVS are analysed into "digits" that will be used serially to select, by global addressing, the appropriate multiple of the VS vector. In this way one vector add can advance the scalar multiplier one digit at a time rather than one bit. By permitting add or subtract and by allowing for gaps between digits, an average digit can deal with n + 3 multiplier bits for 2**n pre-computed multiples. (The code is being implemented in phases, and in the current code digits average n + 2 bits, or 7 bits with n = 5.) The 24-bit mantissa of a 32-bit floating point number requires at most 4 digits. The multiplication is shown schematically in Fig. 2.

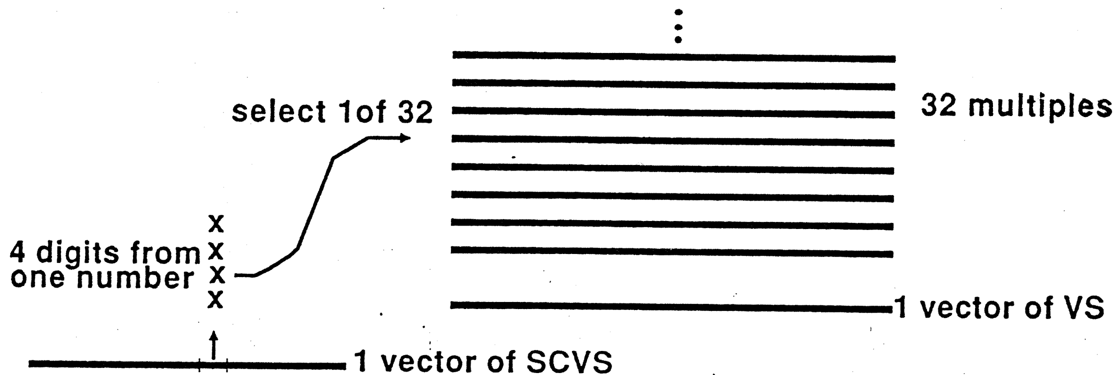


FIG. 2. Schematic for Multiplication in Outer Products

Normal element by element multiplication of 2 vectors with 24-bit mantissas requires 23 adds. On DAP the normal multiplication of a vector by a scalar requires about half as many adds because zero multiplier bits are skipped. The new outer product code requires at most 3 adds, because of the preparation work on both vectors. The technique can barely be used on single scalar-vector multiplies because the overhead of the preparation work cannot be spread over many multiplies.

An outer product contains no adds, but the function MMA1 contains as many adds as multiplies, and these adds are now the bottleneck. When k>1 (e.g. MMS8) the adds in the accumulation are speeded-up by using a form of local block floating point, and only the final subtract uses the standard floating point; in a later phase it is intended to move to a more sophisticated code that adapts to the needs of the data.

In MMS8 8 products are being accumulated, with each product consisting of up to 4 digit multiply and adds. There is advantage in sorting the (up to) 32 digits according to their significance as part of the preparation work, so that the accumulation is more regular. We are thus using sorting to speed up matrix multiplication!

The accumulation now consists of (at most) 4 adds per product. Ignoring the preparation work, all except one of the floating point adds have become one extra (pre-aligned) add; this is nearly an order of magnitude faster. There is a significant requirement for working memory. MMS8 needs 8K bits in every PE for the pre-computed multiples.

5. Examples of Use. The new subroutines have wide potential in linear algebra. We have implemented demonstration examples in matrix multiplication and equation solving via LU decomposition.

Suitable sized matrix multiplication is very easy to code. The FORTRAN-PLUS code to multiply a NxM matrix B by a Mx1024 matrix A to give an Nx1024 matrix C is in essence:

```
      DO 100 J = 1,M,8
      CALL MMA8 (C,A(,,J),B(,,J),N,MASK)
100   CONTINUE
```

The code (with column pivoting) for LU decomposition and equation solving is more complex and will not be described in detail. The main points are:

* 8 pivot columns are extracted and updated to find the next 8 pivot rows ("pivot look-ahead")
* these 8 pivot rows are updated
* the 8 pivot rows and 8 pivot columns are used to perform a rank 8 update of the main data with MMS8.

The above procedure means that if the problem order is large compared with 8, the bulk of the work is done by MMS8.

6. Performance. The rank 8 routines include 1 normal floating point add/subtract for every 8 multiplies and 7 fast adds. This normal add plus the normalisation at the end of the accumulation account for about 40% of the subroutine time. This means there is significant gain in going to rank 16 routines, which have not yet been implemented.

Matrix multiplication closely reflects the performance of the subroutines. The table below gives the performances for a 500 x 64 matrix multiplying a 64 x 1024 matrix to give a 500 x 1024 result matrix.

The performance on solving equations via LU decomposition for a problem of order 1024 is also given in the table. The code uses the new subroutines even in the later stages of decomposition when the matrix is getting small. The MFLOPS of the subroutines in that regime is getting poor, as most of the PEs are idle. A cross-over point occurs when the matrices are of order around 200, when it would be worthwhile switching to the previous 2D mapping that does not use scalar-vector multiplications. (Future work should reduce the cross-over point to below 100, see below.) However, the improvement achieved by switching for a problem of order 1024 is small.

	actual MFLOPS	% normal performance
Matrix multiplication	8	90%
Using MMA1	15.5	170%
Using MMA8	37.4	420%
(Using MMA16	50	560%)
Equation solving		
Old code 2D	6.5	72%
Old code 1D	4.5	50%
Using MMS1	6.6	73%
Using MMS8	19.8	220%
(Using MMS16	26	290%)

TABLE 1. Performance on DAP 510

The subroutines MMA16 and MMS16 have not yet been implemented. The figures given assume that as well as halving the contributions from the normal add and the normalisation, the accumulation advances 8 multiplier bits at a time (instead of 7) and some other minor improvements are made. The latter effects would improve MMA8 and MMS8 performance about 10%. For equation solving with MMS16, the pivot look-ahead work is about 10% of the total; this work could be approximately halved by use of MMS8 and MMS4.

The "% normal performance" column in the table is based on 9 MFLOPS being 100%; this is an equal mix of normal vector adds and multiplies in an ideal situation with no overheads. It is interesting that nearly 6 times that figure can be achieved with the new subroutines.

The accuracy of the equation solving was compared with a normal floating point implementation and found to be very similar on average.

7. **Further Work.** There are obvious extensions to rank 16, 4 and 2. Above rank 16 there are diminishing returns and temporary storage demands are large; in due course rank 32 will be worthwhile. More flexibility in the number of pre-computed multiples would also be helpful, for example for smaller N. It is intended that future codes advance n + 3 multiplier bits at a time instead of n + 2. Extending the floating point precisions to the full range offered by FORTRAN-PLUS is important. 64-bit will be approximately 3 times slower and have more severe storage demands; 24-bit will be nearly 2 times faster.

It is intended to investigate more fully making the subroutines adaptive to the data encountered. The aim is to improve accuracy for awkward data so that it is never worse than normal floating point (and is usually better), and to do so in such a way that extra time is used only when difficult data is encountered. It is thought that the effect on average speed will be nil.

The MFLOPS performance of the new subroutines drops proportionately as the current matrix order falls below 1024. It is hoped to produce (more complex) variants of the subroutines in which 2, 3 or 4 PEs are used for each accumulation; this will give higher performance when the matrix order is less than 512, 340 or 256 respectively. The cross-over point for switching to a 2D mapping could be as low as order 64. Performance on solving equations of order 1024 should improve about 15%, and equations of order 300 should more than double to nearly 20 MFLOPS.

The subroutines can, of course, be used for solving systems larger than 1024, although backing memory would need to be used for much bigger problems.

Acknowledgement. We should like to thank Peter Flanders of AMT for help with the data movement using Parallel Data Transforms [3] and with the sorting.

REFERENCES

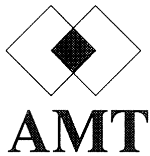
[1] Active Memory Technology, Inc. DAP 510 Attached Processor System, 16802 Aston St, Suite 103, Irvine, Ca.92714, USA.

[2] P.M. FLANDERS, D.J. HUNT, S.F. REDDAWAY and D. PARKINSON, Efficient High Speed Computing with the Distributed Array Processor, in High Speed Computer and Algorithm Organisation, D.J. Kuck, D.H. Lawrie and A.H. Sameh (eds), Academic Press, New York, 1977, pp. 113-128.

[3] P.M. FLANDERS and D. PARKINSON, Data Mapping and Routing for Highly Parallel Processor Arrays, to be published in Future Computing Systems (Oxford University Press), volume 1, number 1 (1988).

[4] S.F. REDDAWAY, DAP - a Distributed Array Processor, Proc. 1st Annual Symposium on Computer Architecture, G.J. Lipovski and S.A. Szygenda (eds), Computer Architecture News, 2, 4 (IEE Cat. No. 73CH0824-3C), 1973, pp. 61-65.

[5] S.F. REDDAWAY, Signal Processing on a Processor Array, in Les Houches, Session XLV, 1985, Traitement du Signal/Signal Processing, J.L. Lacoume, T.S. Durrani and R. Stora (eds), North-Holland, Amsterdam, 1987, pp. 831-858.



Fast Disks for the DAP Computer System

INTRODUCTION

Active Memory Technology supplies a fast disk system which attaches directly to DAP 510 or DAP 610 computer systems. It can handle sustained data transfer rates in excess of 16 Mbytes/sec and up to 45 Gbytes of storage capacity. The system offers outstanding reliability and performance for bulk data applications.

FAST DISK SYSTEM

Hardware

The AMT fast disk system is based upon the Concept 51[1] disk storage system. This high performance disk system is well matched to the DAP's fast input/output system, as described in the AMT publication *DAP Series Technical Overview.*

The disk system transfers data simultaneously from up to 9 high-density disk drives in parallel to achieve transfer rates of over 16 Mbytes/sec. The control unit can support up to 8 banks of drives, each containing up to 9 disk drives, providing a maximum of 45 Gbytes of storage. The system is based on the use of industry standard ESDI Winchester 5.25-inch disk drives.

The hardware configuration of the fast disk system is shown in figure 1. The disk system is controlled via an interface card plugged into the DAP VME bus. Data is transferred to the DAP array memory via a Fast I/O interface card. The I/O interface and all components of the disk system are provided and maintained by AMT.

Data Integrity

Safeguards are built into the disk system to provide high data integrity. A 'safe data disk' holding error correction codes is installed on each bank of disks to protect against the loss of a disk during system operation. The safeguards allow the system to continue uninterrupted service in the event of a disk failure without loss of data or performance.

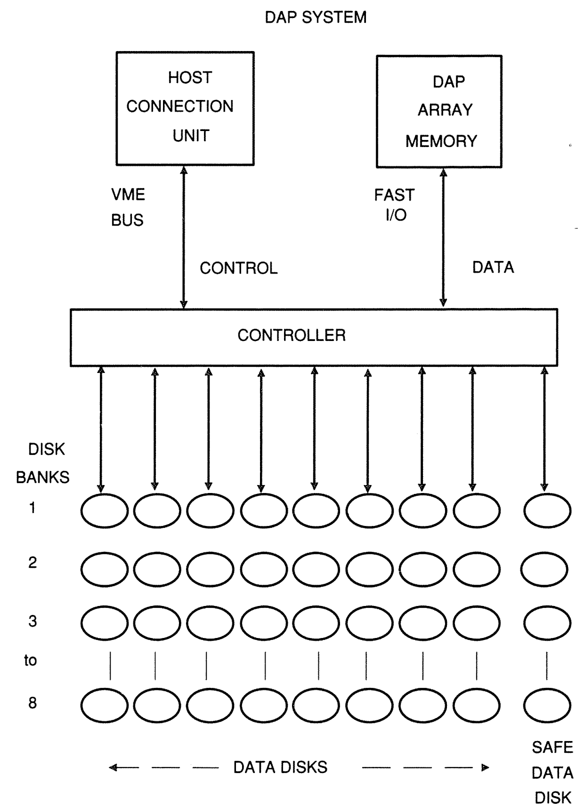


Figure 1. The hardware configuration of the fast disk system.

Software

The disk operating system is specially designed to handle very high speed sustained data transfer rates and to achieve maximum data throughput. From any DAP program, the user is able to create, read and write named files, access status information and perform housekeeping functions.

A software package on the host computer provides interactive command access to housekeeping functions such as disk formatting, compaction, status reporting, back-up, and copying (within the disk and also to or from host files).

A major feature of the interface library is the ability to continue processing while data is being transferred. Figure 2 illustrates a code segment in which one buffer is used to load in new data, while a second buffer is used to process existing data.

1 Concept 51 is a registered trademark of Storage Concepts, Inc.

Fast Disk System

```
          :
    do 100 i=1,1000
      index=index+recordsperimage
      if(imageptr.eq.1)then
        call amt_cdisk_read(fd,image2,index,recordsperimage,1,ierr)
        call process_image(image1)
        call amt_cdisk_wait(fd,ierr)
        imageptr=2
      else
        call amt_cdisk_read(fd,image1,index,recordsperimage,1,ierr)
        call process_image(image2)
        call amt_cdisk_wait(fd,ierr)
        imageptr=1
      endif
100   continue
          :
```

Figure 2. The example of code section shown above illustrates the operation of the disk interface while data is being processed in the DAP array.

In figure 2, the routine *amt_cdisk_read( )* initiates the transfer and *amt_cdisk_wait( )* confirms that the transfer has completed successfully. Between these two calls, data is processed with no reduction in the disk transfer rate and less than 1% reduction in processing performance.

Data is stored on the disk in a fixed-length record format, with each record in the range 4 Kbytes to 512 Kbytes.

SPECIFICATIONS

There are four basic options for the fast disk system:

Total number of drives per bank	Formatted Capacity Gbytes	Sustained Transfer Rate Mbytes/s	Disk drive model number
5	1.4	6.5	Hitachi DK514-38
5	2.8	8.8	Hitachi DK515-78
9	2.8	13.0	Hitachi DK514-38
9	5.6	16.5	Hitachi DK515-78

These figures are for a single bank of drives. Storage capacity can be increased by adding extra banks of drives; the maximum storage of 45 Gbytes is for 8 banks of disk drives.

System data buffering .... 512 Kbytes – 1 Mbytes of SRAM

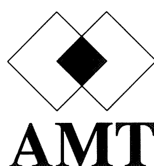
High-speed bus .... 16-bit differential fast bus (DFB), parity checked

Line voltage ....110/220 VAC, 50/60 Hz

Power dissipation .... 800 Watts (max) for 9 drives

Dimensions .... 19-inch rack mounting, 9 inches high and 30 inches deep for a bank of 9 disk drives

Weight .... 150 lbs (65 kg) for a bank of 9 disk drives



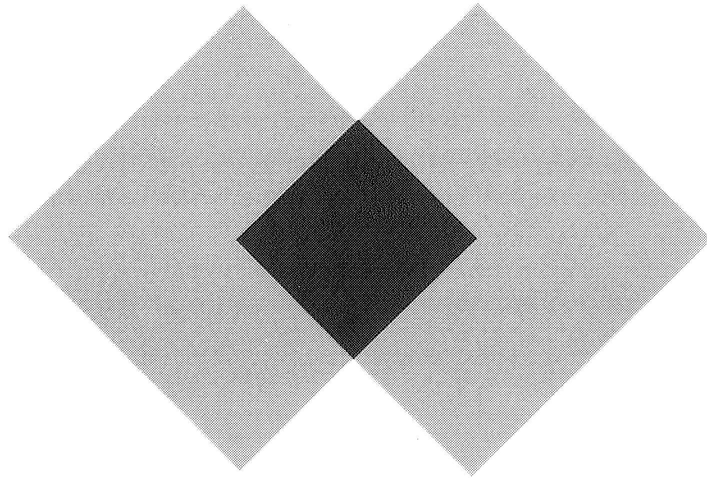
AMT

Active Memory Technology Inc
16802 Aston Street, Suite 103
Irvine
California 92714
U.S.A.
Tel (USA) : (714) 261-8901
Fax (USA) : (714) 261-8802

Active Memory Technology Ltd
65 Suttons Park Avenue
Reading RG6 1AZ
Berkshire
United Kingdom
Tel (UK) : 0734 661111
Fax (UK) : 0734 351395

AMT is a multinational manufacturer of massively parallel computer systems with Company headquarters in Irvine, California. AMT maintains Research and Development activities at its Irvine, California location and at its European headquarters in Reading, U.K.

NOW WITH
4096
PROCESSORS



AMT

ACTIVE MEMORY TECHNOLOGY

MASSIVELY
PARALLEL
COMPUTERS

INTRODUCING THE DAP

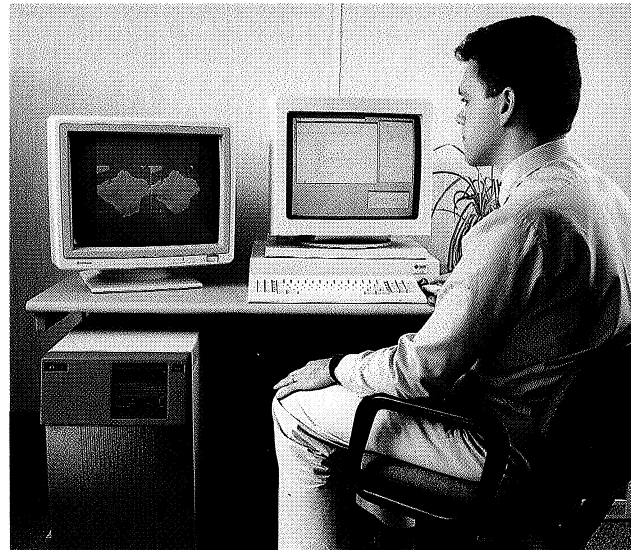
The DAP from Active Memory Technology is a massively parallel computer with thousands of processors which provides a breakthrough in computing price/performance.

As an attached processor for VAX* and Sun* computers, the DAP gives users easy access to very high speed massively parallel computing, achieving speed improvements in excess of 1000 for many applications. Powerful Fortran development tools are provided to program the DAP from these host systems under the standard VMS* and UNIX* operating systems.

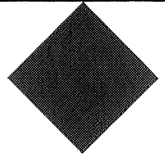
This very powerful computer can perform efficient computation and simultaneously display results on a high resolution colour monitor in real time.

The DAP provides very high speed parallel connections between processors for nearest neighbours, global access and complex routing.

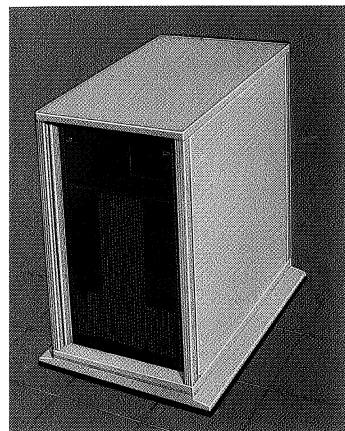
The DAP's very high processing and communication rates make this compact, affordable unit a powerful new engine for all data intensive computing tasks and applications such as CAD, neural networks, signal and image processing.



DAP 510 with a Sun workstation



FAST, POWERFUL AND AFFORDABLE



DAP 610

Performance

The DAP is available with 1024 processors (DAP 510) or 4096 processors (DAP 610) and has the power to cope with very large datasets at very high speeds. Examples of the performance of the DAP 610 system are:

Processors to memory transfer rates:	5,120 MByte/sec.
Logic and Boolean operations:	40,000 million/sec.
Character handling rates:	4,000 million/sec.
Integer operations:	1,600 million/sec.
Floating point operations:	200 million/sec.

A system programmable in Fortran with performance that rivals even specialist hardware.

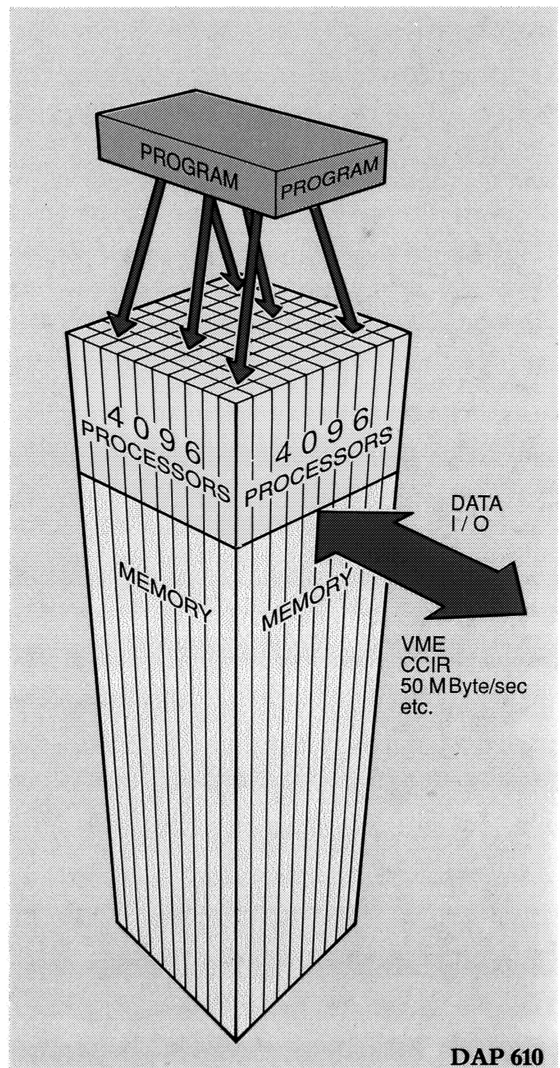
Data visualisation

The DAP offers the real time display of massive datasets with a display window containing over one million values being refreshed at 60Hz on a high quality, high resolution screen. Displays can be updated at video rate using only 1% of the available processing power of the DAP 610.

Rapid throughput

Data intensive processing requires high speed acquisition of data and high speed display. The DAP's key advantage over other parallel processors is the ability to transfer data in and out of the system at 400 million bits per second. Real time acquisition of signals, high speed processing and display of results all take place simultaneously.

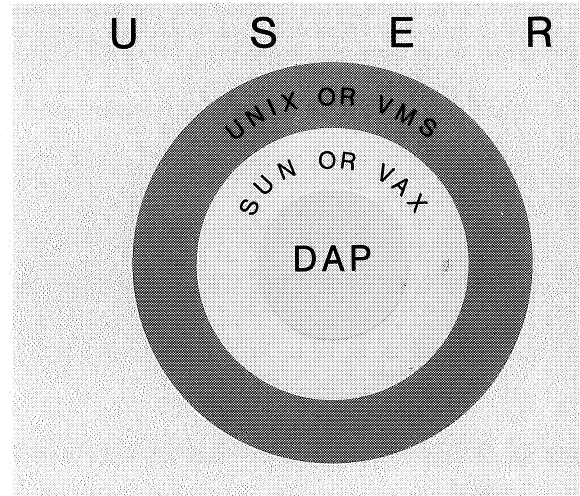
Standard interfaces are available for the DAP including CCIR-601 for digital TV signals and VME (including VSB) interfaces as well as high speed 32-bit parallel ports. High speed serial links using coaxial cables or optical couplers are also available.



DAP 610

USER FRIENDLY

The DAP is very easy to use. AMT provides software development tools which are fully integrated with Sun and VAX systems so that there is no significant learning curve in getting applications running with the DAP massively parallel computer. Users can develop parallel programs for the DAP on their host computer using high-level languages. Often only the computing intensive parts of applications need to be run on the DAP, leaving the host free for user interface and other tasks. The complete software support system for the DAP user runs on the host computer in a familiar environment.



Low cost system

Low cost with the DAP system means fast implementation of user applications as well as low cost hardware.

Its compact design uses low-power, highly reliable CMOS technology. A complete system with up to 4096 processors easily programmed in Fortran brings massively parallel computing within reach of even a small project team.

STRONG SUPPORT

AMT is committed to strong support of the DAP system in all our markets. The DAP is delivered with software tools for parallel computing and host computer connections which are simple and fast to install and use.

Powerful development environment

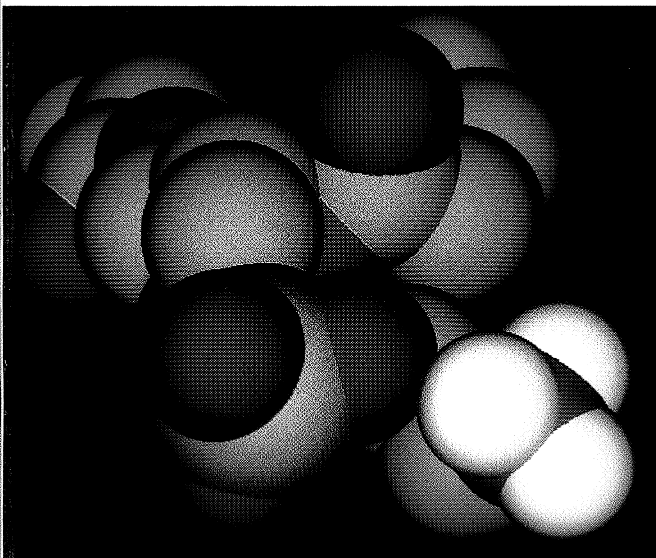
The DAP is easily programmed in Fortran enhanced by extensions to provide simple and efficient handling of arrays of data. Fortran 8X will be supported as soon as the standard is fully defined.

Powerful high level Fortran software and code optimisers are provided for data manipulation and high speed communication between processors.

This AMT-developed software is fundamental to the easy use of massively parallel computers.

There are extensive software libraries to enable the DAP system user to produce rapid solutions for applications. The libraries fully support the multi-processor system and contain a comprehensive set of functions to manipulate matrices and vectors, a wide range of mathematical functions as well as specialist routines for applications in the areas of image and signal processing.

A complete DAP simulator software package is available for VAX and Sun systems which gives accurate timings for off-line program development and testing.



Molecular graphics

The DAP has been successfully used in hundreds of different fields:

Area	Applications
Biotechnology, genetics	DNA sequencing
Command systems	Speech recognition
Electronics	Chip-routing and fault simulation
Fluid flow, aerospace	Navier-Stokes equations
Geophysics	Climatic modelling
High energy physics	Lattice gauge theory
Hydraulics research	Wave-equation in shallow water
Legal and patents	Syntactical retrieval from large files
Mechanical engineering	Finite element analysis
Neural networks	Image enhancement, pattern recognition
Oil exploration	Signal processing of seismographic data
Ordnance survey	Digital cartography
Pharmaceutical	Molecular modelling
Radar systems	Target detection and recognition
Robotics, vision	Image analysis, image processing
Solid state theory	Ising model

DAP SERIES SPECIFICATIONS

Processor array	DAP 510	DAP 610
No. of processors:	1024	4096
Connectivity:	Nearest neighbour connections 4-way Global connection to all processors	
Clock rate:	10 MHz	10 MHz
Processor to memory speed:	1280 MByte/sec	5120 MByte/sec

Memory		
Data memory:	4, 8, 12 or 16 MByte	16, 32, 48 or 64 MByte
Program memory:	0.5 to 2.0 MByte	0.5 to 2.0 MByte

Connections	
VME:	Internal 4 slot VME subsystem (optional)
VAX:	DR11W or DRB32 interface
SUN:	SCSI interface
High-speed serial port:	40 MByte/sec
High-speed parallel port:	50 MByte/sec
CCIR:	CCIR-601 standard 27 MByte/sec

Graphics	
Video display:	1024 x 1024 bit-mapped pixel display, with 1, 2, 4, 8 or 24 bits/pixel, high resolution, double buffered framestore, 60 Hz refreshed system

Physical data	DAP 510	DAP 610
Dimensions:	63cm high 34cm wide 79cm deep	114cm high 61cm wide 91cm deep
Weight:	50 kg	150 kg
Cooling:	Internal fan, office environment	
Power:	110/220 VAC 400 watts	220 VAC 1500 watts

Software	
Languages:	Fortran-Plus (Fortran with vector and matrix extensions for parallel processing) APAL Assembler
Data types:	Boolean Character 8 through 64-bit integers (in 8-bit steps) 24 through 64-bit floating point (in 8-bit steps)

General library:
Contains a wide range of functions to manipulate matrices and vectors, perform variable precision arithmetic, calculate eigenvalues and eigenvectors, solve simultaneous linear equations, perform random number generation, special functions, sorting and utility routines.

Image processing library:
Contains routines for image conversion, image processing primitives, low-level image processing tasks such as convolutions (Sobel, Laplacian, Prewitt, Kirsch, Roberts), neighbourhood averaging, thresholding, 2-dimensional fast Fourier transformations, and image analysis routines to search for particular features.

Signal processing library:
Contains routines for fast Fourier transformations, windowing functions (Hanning, Hamming, Dolph-Chebyshev, Blackman, Kaiser-Bessel, Gaussian), signal generators (sine, chirp and exponential decay) and array format conversions.

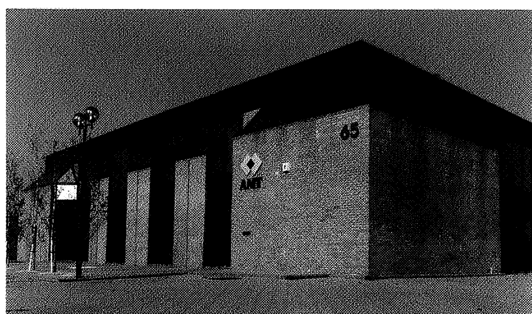
Connection software:
Parallel data transforms provide high level routines and code optimisers for data communications and routing between processors.

Simulator:
There is a complete software package to simulate the DAP system on Sun and VAX computers which provides accurate timings for off-line program development and testing.

OEM products

DAP systems are available as board sets for systems integrators.

Active Memory Technology is a manufacturer of massively parallel computer systems, with offices in the UK and USA.



For further information, please contact:

Marketing Manager,
Active Memory Technology Ltd,
65 Suttons Park Avenue,
READING RG6 1AZ,
United Kingdom.

Telephone (UK): 0734 661111
Fax (UK): 0734 351395

or

Marketing Manager,
Active Memory Technology Inc,
16802 Aston Street, Suite 103,
IRVINE, California 92714
U.S.A.

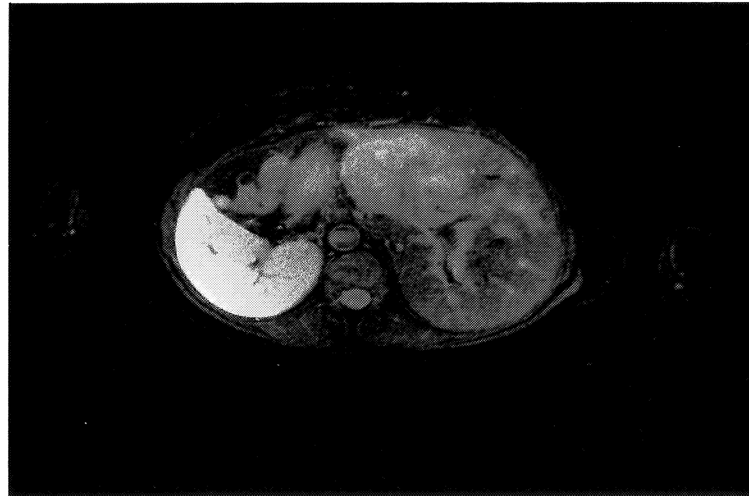
Telephone (USA): (714) 261-8901
Fax (USA): (714) 261-8802

•VAX is the trademark of Digital Equipment Corporation.
•UNIX is the trademark of Bell Telephone Laboratories Ltd.
•VMS is the trademark of Digital Equipment Corporation.
•Sun is the trademark of Sun Microsystems Inc.

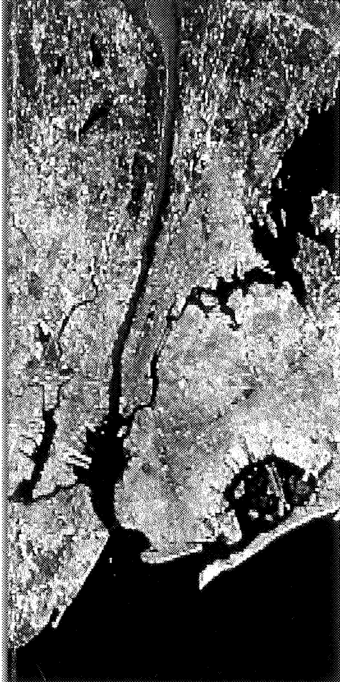
WIDELY USED

Image processing

The combination of high speed I/O and massively parallel computing means that the DAP improves the performance of the host computer by a factor in excess of 1000 for image processing. For example, the DAP can process image data calculations at 1600 million operations per second and simultaneously display the processed images.



Medical imaging



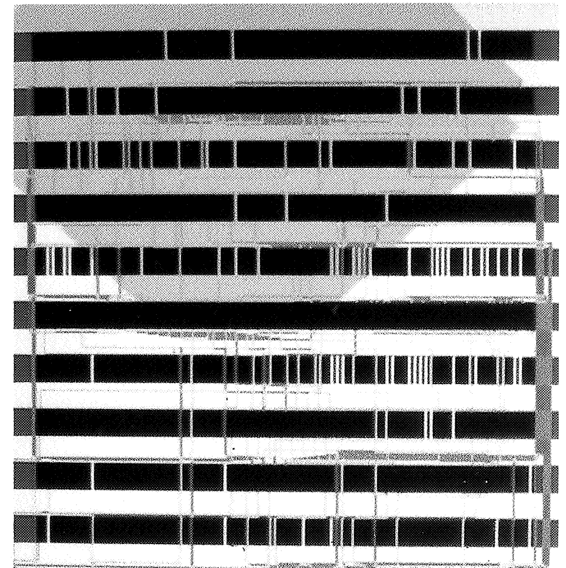
Satellite imaging

CAE engine

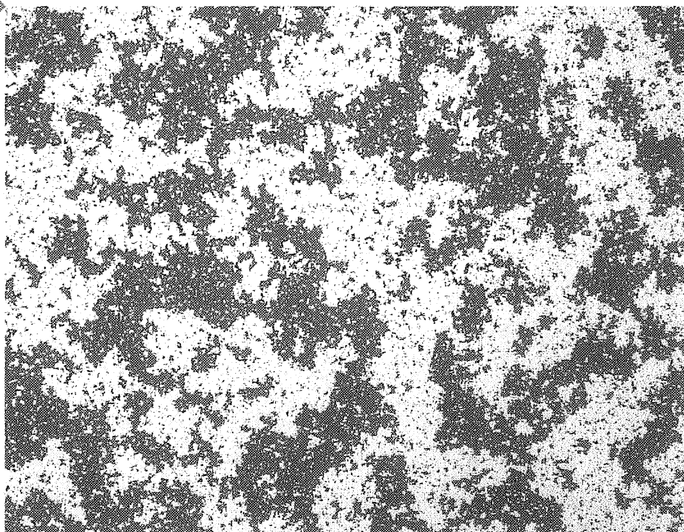
The fast processing speeds of the DAP make it ideal for circuit design and fault simulation. An example is the routing problem on a chip. The time taken for the physical routing of electrical connections on a gate array can be unacceptably long on a CAE workstation or mainframe computer. The solution involves millions of routing decisions. The DAP cuts the run time from hours to minutes.

Materials research

The simulation of ferromagnetic material involves billions of spin-flip operations. These Boolean operations are so fast on the DAP that it can run Ising model simulations at several times the speed of a Cray X-MP supercomputer. The fast I/O facility of the DAP has allowed researchers, for the first time, to change the parameters interactively and view the results in real time.



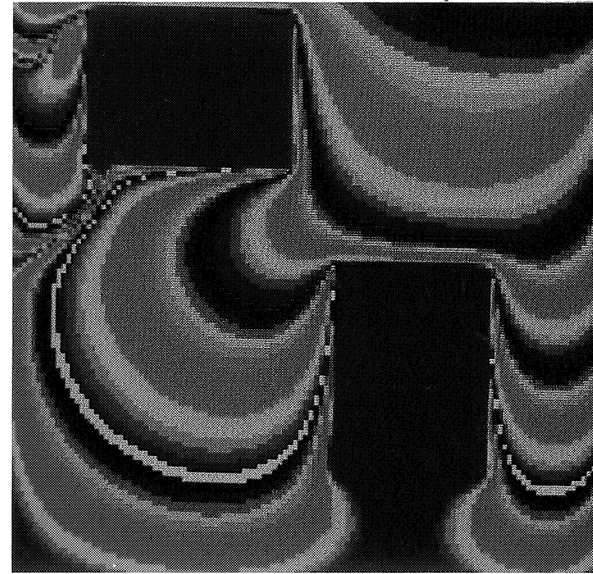
Chip routing



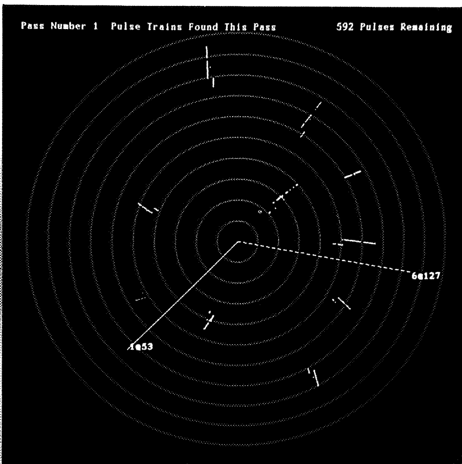
Ising model

Fluid flow

The DAP combines high power computing with the instant display of results. In the example, a large-scale computation simulates the flow of a fluid around objects in a square tank. The problem is solved on a 256 x 256 mesh, in which the boundaries between the colours indicate flow streamlines. The flow is calculated by solving Navier-Stokes equations at 65,000 co-ordinates every 0.3 second and the results are displayed in real time.



Fluid flow



Emitter identification

Electronic support measures

The DAP is a powerful tool for the complex signal processing ESM task of deinterleaving radar pulse trains. Real time capture of radar data is possible using the fast I/O system. The data analysis is supported by a suite of software which performs histogramming, harmonic analysis and pulse train removal. A leading systems consultancy, Systems Designers Scientific, has successfully implemented a fully functional deinterleaving system which shows that the DAP can deliver outstanding performance for these complex operations.

Analysing the simultaneous radar signals from various sources requires outstanding performance.

This chapter describes the syntax and function of APAL instructions.

Section 11.1 describes the functions of the various instruction fields in the internal representation of an APAL instruction. It also describes the modification of addresses and the stepping of addresses in APAL DO loops.

Section 11.2 describes the syntax of various types of operand that commonly appear in APAL instructions. These descriptions are referred to by the syntax descriptions of individual instructions in section 11.3.

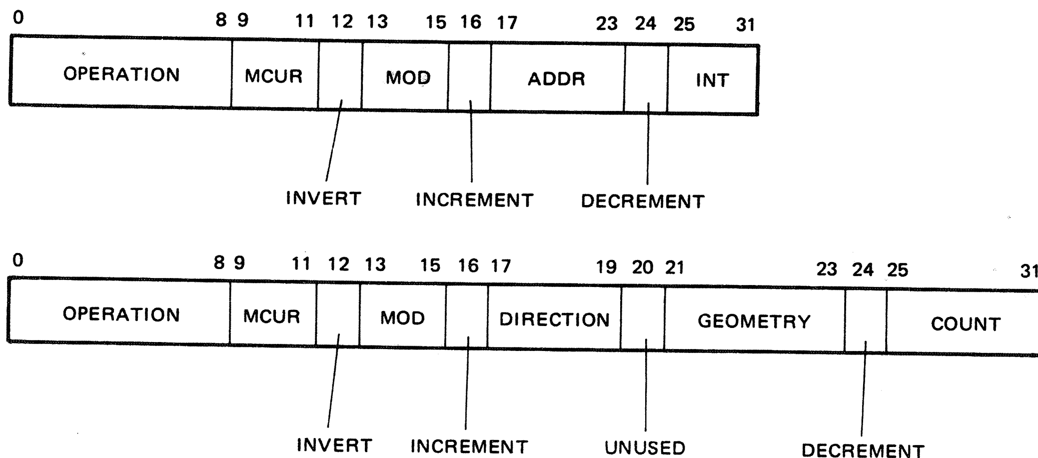
Section 11.3 describes the function and syntax of each APAL instruction.

11.1 Introduction to APAL instructions

11.1.1 Instruction fields in the internal representation of APAL instructions

Each APAL instruction is assembled into a 32-bit word. Assembled code begins on a store plane boundary and the assembled APAL instructions are packed two per row.

The internal representation of an assembled APAL instruction consists of a number of *instruction fields*; two of the most common subdivisions of an instruction are shown as examples below:



The significance of each of these instruction fields is described in the following sections.

11.1.1.1 *The OPERATION field*

The OPERATION field is a nine-bit field and occupies bits zero to eight of the assembled instruction. The contents of this field identify the operation to be performed by the instruction.

The bit pattern corresponding to each APAL instruction mnemonic is specified in the appropriate instruction description in section 11.3.4.

11.1.1.2 *The MCUR field*

The MCUR field is a three-bit field and may occupy either bits five to seven or bits nine to 11 of an assembled instruction; the position of the MCUR field depends on the type of instruction and is specified in the appropriate instruction description in section 11.3.4.

The contents of this field are interpreted as an unsigned integer in the range zero to seven and identify the corresponding MCU register M0, M1, M2, M3, M4, M5, M6, or M7.

The MCU register identified by the MCUR field may be:

1 The first operand of a register to register instruction. In this case the MCU register specified in this field will also contain the result of the instruction
2 The register used to form the R plane or orthogonal R plane (see section 11.3.3)
3 The operand of an instruction that addresses a single bit of that MCU register

11.1.1.3 *The INVERT field*

The INVERT field is a one-bit field and occupies bit 12 of the assembled instruction. This field may be considered as an extra bit of the OPERATION field (see section 11.1.1.1).

Many APAL instructions have companion instructions that perform the same function but first invert all the bits of data extracted from one of the operands of the instruction. The internal representations of such instruction pairs are identical except for the value of the INVERT field; if the field is set to one, one of the operands is inverted, otherwise the operands are unaffected.

Whether or not the INVERT field is set, and, if so, which operand is inverted, depends on the APAL instruction (see section 11.3.4).

11.1.1.4 *The MOD field*

The MOD field is a three-bit field and occupies bits 13 to 15 of the assembled instruction. The contents of this field are interpreted as an unsigned integer in the range zero to seven and identify the corresponding MCU register M0, M1, M2, M3, M4, M5, M6, or M7.

The MCU register identified by the MOD field may be:

1 The second operand of a register to register instruction
2 The modifier register to be used by an instruction to which address modification is applicable (see section 11.1.2.1). If M0 is specified in this case, no instruction modification takes place; that is, modification with M0 as the modifier register is equivalent to modification with a register all of whose bits are zero

11.1.1.5 *The INCREMENT and DECREMENT fields*

The INCREMENT and DECREMENT fields are both one-bit fields and occupy bits 16 and 24 respectively of the assembled instruction. It is convenient to regard these fields as a single two-bit INCREMENT/DECREMENT field.

The contents of this field specify how an address in the instruction is to be stepped if the instruction appears inside an APAL DO loop (see section 11.1.2.2), as follows:

INCREMENT	*DECREMENT*	*Effect*
0	1	Address is decremented
1	0	Address is incremented
0	0	Address is unaffected
1	1	The effect, within a DO loop, is undefined

For an instruction outside an APAL DO loop, the value of the INCREMENT/DECREMENT field is immaterial.

The amount by which an address is stepped may be one store plane, one row, one column, one MCU register bit, or both a store plane and an MCU register bit, depending on the type of the address.

11.1.1.6 *The ADDR and INT fields*

The ADDR and INT fields are both seven-bit fields and occupy bits 17 to 23 and bits 25 to 31 respectively of the assembled instruction. The contents of these fields are used to construct an address. The address may be that of a store plane, row, column, MCU register bit, or both a store plane and an MCU register bit, depending on the instruction in which the address appears.

The following table summarises how the ADDR and INT fields are interpreted for each type of address.

Address	*ADDR field*	*INT field*
Store plane	Displacement of store plane relative to start of DAP program block, or the displacement of the plane address held in an MCU register if address modification is specified	Not used
Store row	*Plane part* of row address; that is, displacement of a store plane relative to start of DAP program block, or the displacement of the plane address held in an MCU register, if address modification is specified	*Row part* of row address; that is, row number within store plane identified by plane part of row address
Store column	Plane part of address	*Column part* of address; that is, column number within store plane identified by plane part of column address
MCU register bit	Not used	Number of MCU register bit. The MCU register is specified in the MCUR field (see section 11.1.1.2)
Store plane and MCU register bit	Displacement of store plane relative to start of DAP program block, or the displacement of the plane address held in an MCU register, if address modification is specified	Number of MCU register-bit

Store planes are numbered from zero at the start of the DAP program block; rows, columns, and MCU register bits are numbered from zero to 63.

All of the above addresses may be modified (see section 11.1.2.1) and/or stepped (see section 11.2.2.2).

11.1.1.7 *The DIRECTION field*

The DIRECTION field, for an APAL shift instruction, is a three-bit field and occupies bits 17 to 19 of the assembled instruction. The contents of this field specify the direction of the shift, either independently of or relative to a direction specified in a modifier register, as follows:

DIRECTION field	*Direction of shift*	
000	Given by DIRECTION field of modifier register	
010	DIRECTION field of modifier register rotated 90° clockwise	
100	DIRECTION field of modifier register rotated 180° clockwise	
110	DIRECTION field of modifier register rotated 270° clockwise	
001	North	Without regard to DIRECTION field of modifier register, if specified
011	East	
101	South	
111	West	

11.1.1.8 *The GEOMETRY field*

The GEOMETRY field, for an APAL shift instruction, is a three-bit field and occupies bits 21 to 23 of the assembled instruction. The contents of the field specify the *geometry* to be applied to the shift; this may be independent of, or as given by, a geometry specified in a modifier register.

The geometry of a shift specifies the values that are shifted in at the edge of a plane that is being shifted. The geometry of a shift may be *plane* or *cyclic*. Plane geometry implies that zeros are shifted in at the edge of a plane; cyclic geometry implies that the values shifted off the edge of a plane are shifted in at the opposite edge.

The GEOMETRY field is interpreted as follows:

GEOMETRY field	*Geometry of shift*
100	Plane geometry for all shifts
101	Plane geometry for east and west shift, cyclic geometry for north and south shifts
110	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
111	Cyclic geometry for all shifts
0..	Given by GEOMETRY field of modifier register (. indicates that the corresponding bit may be zero or one)

11.1.1.9 *The COUNT field*

The COUNT field is a seven-bit field and occupies bits 25 to 31 of the assembled instruction. The contents of this field may specify:

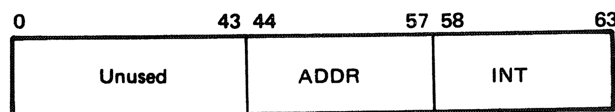
1 The number of places by which a register plane is to be shifted
2 The number of places by which an MCU register is to be shifted
3 The number of instruction cycles allowed for a ripple instruction

The contents of the COUNT field of an instruction may be modified by the COUNT field of a modifier register (see section 11.1.2.4).

11.1.2 Instruction modification and DO loop stepping

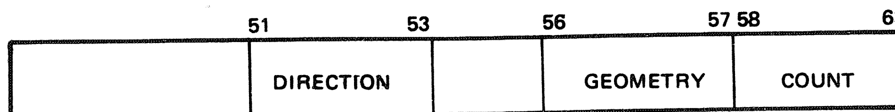
11.1.2.1 *Instruction modification*

When used as a modifier register for modifying addresses, an MCU register is divided into ADDR and INT fields as shown below.



The contents of the ADDR and INT fields of the modifier register are used to modify the ADDR and INT fields of the instruction as described in section 11.1.2.3.

An APAL shift instruction may also specify a modifier register in which case the MCU register used as the modifier is divided into DIRECTION, GEOMETRY, and COUNT fields as shown below.



The contents of the DIRECTION, GEOMETRY, and COUNT fields of the modifier register are used to modify the DIRECTION, GEOMETRY, and COUNT fields of the instruction as described in section 11.1.2.4.

Using only the seven-bit ADDR field of the instruction, an instruction may address 128 store planes, which is only a small fraction of the store planes in the DAP. The ADDR field of an instruction, which is fixed at assembly time, may be combined at run time with the ADDR

field of a *modifier register* to obtain a plane address that may range over the entire DAP store; this process is called *modification.* Modification may also be applied to the INT field of an instruction.

11.1.2.2 *DO loop stepping*

If an instruction appears inside an APAL DO loop (see section 11.3.4), the user may specify that the address referred to by the instruction is to be stepped; that is, the address is to be incremented or decremented each time it is executed during a pass through the DO loop.

DO loop stepping does not alter the address in the assembled instruction.

The effect of DO loop stepping on the various types of address is described in section 11.2.2.

11.1.2.3 *Effective ADDR and INT values*

In general, the address referred to in an APAL instruction has two components: an *effective ADDR value* and an *effective INT value.*

An effective ADDR or INT value is one that takes both modification and DO loop stepping into account.

The effective ADDR value represents the number of a store plane, relative to the start of the DAP program block.

The interpretation of the effective INT value depends on the type of instruction in which it appears. The following table shows how the effective ADDR and INT values are interpreted for the various types of address and also shows how they are derived.

Type of address	*Effective ADDR value*	*Effective INT value*
Store plane	Store plane displacement relative to start of DAP program block. = ADDR field of instruction + ADDR field of modifier (if specified) + DO loop step value (if specified)	Not relevant
Store row	Plane part of row address. = ADDR field of instructions + ADDR field of modifier (if specified) + carry from effective INT value (see below) + DO loop step value (if applicable)	Row part of address. = INT field of instructions + INT field of modifiers (if specified) + DO loop step value (if applicable)
Store column	Plane part of address. = ADDR field of instruction + ADDR field of modifier (if specified) + DO loop step value (if applicable)	Column part of address. = INT field of instruction + INT field of modifier (if specified) + DO loop step value (if applicable)
MCU register bit	Not relevant	Number of MCU register bit. = INT field of instruction + INT field of modifier (if specified) + DO loop step value (if applicable)
Store plane and MCU register bit	Store plane displacement relative to start of DAP program block. = ADDR field of instruction + ADDR field of modifier (if specified) + DO loop step value (if applicable)	Number of MCU register bit. = INT field of instruction + INT field of modifier (if specified) + DO loop step value (if applicable)

If DO loop stepping is specified for instructions inside an APAL DO loop, the appropriate increment or decrement is combined into either the effective ADDR or INT value (or both), depending on the type of instruction and on the way the stepping is specified (see section 11.2.2).

The effective ADDR value is truncated to 14 bits. The effective INT value is truncated to six bits. In the case of a store row address, any carry generated by producing the effective INT value is added to the effective ADDR value; this will occur in the event of the row part of the address being greater than 63, in which case the row address effectively crosses a store plane boundary. In all other cases the effective INT value must be in the range zero to 63, and is referred to as a *restricted effective INT value.*

Due to the structure of the DAP program block, there are a number of restrictions on the ranges of the effective ADDR value (see section 11.2.3.3).

11.1.2.4 *The effective DIRECTION, GEOMETRY, and COUNT values*

The direction, geometry, and magnitude of a shift caused by an APAL shift instruction are given by the effective DIRECTION, GEOMETRY, and COUNT values respectively.

The simplest case is when a shift instruction does not specifiy a modifier register, in this case, the effective DIRECTION, GEOMETRY, and COUNT values are given by the contents of the DIRECTION, GEOMETRY, and COUNT fields of the instruction (see sections 11.1.1.7, 11.1.1.8, and 11.1.1.9 respectively).

However, if a shift instruction does specify a modifier register, the DIRECTION, GEOMETRY, and COUNT fields of the instruction are combined with the DIRECTION, GEOMETRY, and COUNT fields of the modifier register.

The DIRECTION field of a modifier register is interpreted as follows:

DIRECTION field	*Effective DIRECTION value*
000	Self
001	North
010 011	East
100 101	South
110 111	West

(North, East, South, West:) The effective DIRECTION value may be a rotation of one of these directions, depending on the value of the DIRECTION field of the instruction (see section 11.1.1.7)

The GEOMETRY field of a modifier register is interpreted as follows:

GEOMETRY field	*Effective GEOMETRY value*
00	Plane geometry for all shifts
01	Cyclic geometry for north and south shifts, plane geometry for east and west shifts
10	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
11	Cyclic geometry for all shifts

The effective COUNT value of a modified shift instruction is the sum, modulo 64, of the COUNT fields of the instruction and the modifier register.

11.2 Addressing constructs

This section describes the syntax of a number of addressing constructs that frequently appear in APAL instructions.

An APAL instruction may address any of the following:

1 A store plane
2 A row within a store plane
3 A column within a store plane
4 A particular bit of a specified MCU register
5 An instruction in the same or another code section

The first four types of address may be both modified and, if within an APAL DO loop, stepped.

11.2.1 Specifying a modifier register

In general, an address in an APAL instruction (other than an instruction address) may be one of three types:

1 A data address, which is either of the following:
   (a) The name of a data variable, an optional plane and/or row offset, and a modifier register
   (b) A plane and/or row offset from an address held in a modifier register
2 An absolute address, consisting of a plane and/or row offset from the start of the DAP program block (that is, no modifier register is specified)
3 The number of a bit in a specified MCU register. This consists of the name of the MCU register and an integer value representing the bit number

Because of the structure of the DAP program block and the size of the instruction fields used to construct store addresses, the first type of address can only be constructed using address modification. The other types of address may use address modification, but it is not mandatory.

An instruction performs address modification by specifying a *modifier register*, as follows:

(*modifier*)

where *modifier* is one of the MCU registers M1, M2, ..., M7. Note that M0, when used as a modifier, is effectively a modifier of zeros; the assembler will flag an error if M0 is used as a modifier register. If DO loop stepping is also specified, *modifier* and *step* can be written in a short combined form (see section 11.2.2).

The effects of address modification on various types of address are described in the relevant sections.

11.2.2 Specifying DO loop stepping

The DAP allows the user to group up to 60 APAL instructions together in a *DO loop* and to cause this instruction sequence to be executed repeatedly a number of times (see section 11.3.4).

If an instruction addresses the store and/or an MCU register bit it may specify that the address is to be *stepped* (that is, incremented or decremented) each time the instruction is executed during a pass through the DO loop.

DO loop stepping is specified by a construct of the form:

(*step*)

where *step* may be written as +, −, + A, or − A. If address modification is also specified, the two constructs may be specified in either of the following ways:

(*modifier*) (*step*)
(*modifier step*)

The unit by which an address is stepped depends on the type of address (see below). The number of units by which an address is stepped depends on the *current DO loop step value.* Immediately after the DO instruction is executed to initialise the loop, the current DO loop step value is defined to be zero; at the end of each pass, it is incremented by one.

The following table summarises the effect of DO loop stepping on the various types of address (n is the current DO loop step value):

Type of address	*Stepped address*			
	+	−	*+A*	*−A*
Store plane address	Store plane address + n	Store plane address − n	Not valid in this context	Not valid in this context
Store row address (plane and row parts)	Plane part Row part + n	Plane part Row part − n	Plane part + n Row part	Plane part − n Row part
Store column address (plane and column parts)	Plane part Column part + n	Plane part Column part − n	Plane part + n Column part	Plane part − n Column part
MCU register bit address	Bit number + n	Bit number − n	Not valid in this context	Not valid in this context
Store plane address and MCU register bit address	Store plane address + n Bit number + n	Store plane address − n Bit number − n	Not valid in this context	Not valid in this context

If *step* is + or −, the current DO loop step value is effectively added to or subtracted from the INT field of the instruction. If *step* is +A or −A, this DO loop step value is added to or subtracted from the ADDR field of the instruction; if the address is that of a store plane and an MCU register bit, the current DO loop step value is added to or subtracted from the ADDR and INT fields of the instruction.

DO loop stepping may be specified for instructions outside APAL DO loops, but no stepping of addresses will be performed.

If a row or column address is to be stepped, the address may be stepped either by a single row or column or by an entire store plane. Bit 11 of such an instruction becomes a SELECT field. If step is + or −, the SELECT field is set to zero and the address is stepped by a single row or column; if step is + A or − A, the SELECT field is set to one and the address is stepped by an entire store plane. The MCUR field of such an instruction will occupy bits five to seven.

11.2.3 Store addresses

11.2.3.1 *Syntax*

This section describes the syntax of store addresses, the derivation of effective ADDR and INT values from these addresses, and the constraints on the parts of the DAP program block that may be addressed in this way.

⟨store address⟩ ::= ⟨store plane address⟩ | ⟨store row address⟩ | ⟨store column address⟩

⟨store plane address⟩ ::= ⟨plane⟩⟨modifier⟩?⟨step part⟩?

⟨store row address⟩ ::= ⟨row⟩⟨modifier⟩?⟨step part⟩?

⟨store column address⟩ ::= ⟨column⟩⟨modifier⟩?⟨step part⟩?

⟨data address⟩ ::= ⟨plane⟩ | ⟨row⟩ | ⟨column⟩

⟨plane⟩ ::= ⟨aligned data name⟩⟨plane offset⟩? | ⟨plane number⟩

⟨row⟩ ::= ⟨data name⟩⟨plane offset⟩?⟨row offset⟩? |
⟨plane number⟩⟨row offset⟩? |
⟨row offset⟩

⟨column⟩ ::= ⟨aligned data name⟩⟨plane offset⟩?⟨column offset⟩? |
⟨plane number⟩⟨column offset⟩? |
⟨column offset⟩

⟨modifier⟩ ::= (⟨mreg⟩)

⟨mreg⟩ ::= M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨step part⟩ ::= ⟨step⟩ | ⟨step A⟩

⟨step⟩ ::= (+)|(−)

⟨step A⟩ ::= (+A)|(−A)|⟨step⟩

⟨aligned data name⟩ ::= ⟨data name⟩

⟨data name⟩ ::= ⟨data section name⟩ | ⟨data variable name⟩ | ⟨identity name⟩

⟨plane offset⟩ ::= + ⟨plane number⟩

⟨row offset⟩ ::= .⟨number⟩

⟨column offset⟩ ::= .⟨number⟩

⟨plane number⟩ ::= ⟨number⟩

When ⟨modifier⟩ and ⟨step⟩ are both present in a store address, they may be combined by eliding parentheses; for example, the following are equivalent:

(M1)(+)
(M1 +)

11.2.3.2 *Semantics*

An instruction that addresses the DAP store may do so using a *data address*.

A data address may be formed in any of the following ways:

1 The name of a data section, a data variable, or a data identity that specifies an address within a data section. Plane and/or row/column offsets from the address corresponding to this name may optionally be specified.

  Such an address must specify a modifier register (see section 11.2.1). DO loop stepping (see section 11.2.2) may be specified for the address

2 An absolute plane number with an optional row/column offset, or an absolute row number. A modifier register and/or DO loop stepping may also be specified

3 A data identity name referring to an address within store planes zero to 35 of the DAP program block

An instruction may read from or write to a data address.

Examples

DATAVAR1 + 14.2 (M3)
DATAVAR2 + 10 (M4) (+)
20.14 (−)
.9 (M6)

As described in section 11.1.2.3, a run-time address consists of an effective ADDR value and an effective INT value. These values are generated from a store address, as it is written, as follows.

The effective ADDR value generated by a store address is the sum of:

1 The ADDR field of the modifier register, if any.

  For a data address in which a modifier register is specified, the ADDR field of the modifier register must contain the address of the data section referred to by the named data variable or data identity. Loading this value into the modifier register prior to executing an instruction that contains the address is the responsibility of the user, and may be done using the RASC instruction.

However, this only allows the user to address the first 128 store planes of the data section. If a greater displacement is required, the ADDR field of the modifier register must be loaded with an address within the data section, using the RAR instruction. For example:

```
DATA D
D1: 128*PLANE
D2: 1, 2, 3
.
.
.
END
CODE C
.
.
.
RAR M1 D2
.
.
.
CPCQS 0 (M1)
.
.
.
END
```

2 The ADDR field of the assembled instruction. This is itself the sum of:
   (a) An absolute plane number or plane offset, if any
   (b) The plane displacement of the named data variable or data identity within the data section
3 The current DO loop step value, provided that:
   (a) The instruction appears inside an APAL DO loop
   (b) DO loop stepping has been specified for the instruction
   (c) The instruction requires a plane aligned address
   Note that the current DO loop step value is not added into the ADDR field of the assembled instruction; the effective ADDR value is a notional run-time sum

The effective INT value generated by a store address is the sum of:

1 The INT field of the modifier register, if any
2 The INT field of the assembled instruction. This is itself the sum of:
   (a) An absolute row number or row/column offset, if any
   (b) The row number of the named data variable or data identity within its data section store plane
3 The current DO loop step value, provided that:
   (a) The instruction appears inside an APAL DO loop
   (b) DO loop stepping has been specified for the instruction
   (c) The instruction addresses a store row, store column, or an MCU register bit
   Note that the current DO loop step value is not added into the INT field of the assembled instruction; the effective INT value is a notional run-time sum

If an instruction addresses a store row, computation of the effective INT value may produce a value greater than 63. In such a case the effective INT value is taken modulo 64, and bits of the effective INT value thus removed are carried into the effective ADDR value; that is, a row address may cross plane boundaries. In cases where the effective INT value represents a column number or an MCU register bit number, a value greater than 63 causes an error.

11.2.3.3 *Assembler checking of store addresses*

Because of the structure of the DAP program block and the size of ADDR fields of instructions, the effective ADDR value generated by a store address is subject to the following constraints:

1 An instruction that addresses a data section must specify a modifier register
2 The ADDR field of an instruction that references a data section or a modified absolute address must be less than or equal to 127
3 In store addresses of the form

⟨plane number⟩?⟨row/column offset⟩?⟨modifier⟩
the value of ⟨plane number⟩ must be less than or equal to 127

4 Store addresses of the form:
⟨plane number⟩⟨row/column offset⟩?
or involving data identity names representing absolute store addresses must produce an effective ADDR value in the range zero to 35

11.2.4 MCU register bit addresses

This section describes the syntax of an MCU register bit address and describes the derivation of effective ADDR and INT values from the address.

11.2.4.1 *Syntax*

⟨MCU register bit address⟩ ::= ⟨MCU register⟩.⟨bit number⟩⟨modifier⟩?⟨step⟩?

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨bit number⟩ ::= ⟨number⟩

11.2.4.2 *Semantics*

A number of APAL instructions address a particular bit of a specified MCU register. An MCU register bit address specifies:

1 The MCU register whose bit is to be addressed; this may be any of the MCU registers M0, M1, ..., M7
2 The number of the bit of the specified MCU register to be addressed, which must be in the range zero to 63

An MCU register bit address generates a restricted effective INT value that is the sum of:

1 The INT field of the modifier register, if any
2 The INT field of the assembled instruction, which contains the bit number specified in the instruction
3 The current DO loop step value, provided that:
   (a) The instruction appears inside an APAL DO loop
   (b) DO loop stepping has been specified for the instruction
   Note that the current DO loop step value is not added into the INT field of the assembled instruction; the effective INT value is a notional run-time sum

The effective INT value must be in the range zero to 63.

Examples

M0.12
M1.20 (M2)
M6.14 (M3) (+)
M7.30 (−)

A number of APAL instructions address both an MCU register bit and a store plane. The syntax of such an address is:

⟨MCU register⟩.⟨bit number⟩⟨store address⟩

where ⟨store address⟩ is a plane aligned address as described in section 11.2.3. The effective INT value generated by such an address is as described above; the effective ADDR value is as described in section 11.2.3.

Examples

M0.12 VARI + 2 (M6) (+)
M1.20 (−)
M6.14 471 (M4)

11.2.5 Instruction addresses

This section describes the syntax of construction addresses as written in APAL jump instructions.

11.2.5.1 *Syntax*

⟨instruction address⟩ ::= ⟨within-section address⟩ | ⟨inter-section address⟩

⟨within-section address⟩ ::= ⟨code label name⟩⟨label offset⟩? | ⟨star⟩⟨label offset⟩

⟨inter-section address⟩ ::= ⟨code section name⟩⟨section offset⟩? | ⟨entry point name⟩ ⟨section offset⟩?

⟨label offset⟩ ::= + ⟨number⟩ | − ⟨number⟩

⟨section offset⟩ ::= + ⟨number⟩

⟨star⟩ ::= *

11.2.5.2 *Transferring control within a code section*

The J and JSL instructions (see section 11.3) transfer control to another instruction in the same code section by placing the address of the destination .instruction in the program counter.

The address of an instruction in the same code section may be either of the following:

1 The name of a code label in the same code section with an optional displacement (in 32-bit words) forwards or backwards

2 A displacement, forwards or backwards, from the current instruction; that is, the jump instruction. The current instruction is represented by the character *

Examples

LAB1	(transfer to instruction labelled LAB1)
LAB2+3	(transfer to third instruction after instruction labelled LAB2)
* − 2	(transfer to instruction two instructions before this instruction)

11.2.5.3 *Transferring control between code sections*

The JE and JESL instructions (see section 11.3) transfer control to an instruction in a different code section by placing the address of the destination instruction in the program counter. These instructions may also transfer control to the beginning of the current code section, or to a named entry point within it.

If the operand of a JE or JESL instruction has not been previously declared, the assembler will implicitly define it as a code section. When the name is actually declared in the same or a subsequent module it may be a code section or entry point name.

The address of an instruction in a different code section may be either of the following:

1 The name of another code section, with an optional forward displacement (in 32-bit words)

2 The name of an entry point within another code section, with an optional forward displacement.

Examples

CODESEC2 + 3	(transfer to fourth instruction in code section CODESEC2)
ENTPOINT1	(transfer control to first instruction following entry point ENTPOINT1)

All instruction addresses in an assembled jump instruction are relative to a program code datum, and must not exceed the program code limit.

11.3 The instruction set

The syntax and function of each APAL instruction is described in section 11.3.4. The instructions are described in alphabetical order, and where an instruction has a companion instruction that performs the same operation on an inverted operand these instructions are described together.

Each instruction description has the following general form:

1. A brief description of the function(s) performed by the instruction
2. The syntax of the instruction as it is written in an APAL code section. Many of these syntax descriptions refer to the appropriate part of section 11.2
3. The 32-bit binary format into which the instruction is assembled. Each instruction field is represented by an alphabetic character, repeated a number of times equal to the number of bits occupied by that field. Instruction fields are represented by the following characters:

Character	*Instruction field*
A	ADDR field
C	COUNT field
D	DIRECTION field
G	GEOMETRY field
I	INT field
J	Field containing an instruction address in a jump instruction
M	MOD field
R	MCUR field
S	SELECT field (for instructions that address store rows or columns)
+	INCREMENT field
−	DECREMENT field
Z	Field containing the address of a store plane each bit of which is zero

The OPERATION field is represented by the actual nine-bit binary pattern associated with the instruction mnemonic.

A period (.) represents an instruction bit that is ignored by the hardware and is set to zero by the assembler.

The binary format may be followed by notes describing how the APAL syntax is related to the binary format

4. Run-time program errors that may occur as a result of the execution of the instruction
5. In non-trivial cases, an example of the instruction

11.3.1 Pseudo and compound instructions

A number of pseudo instructions are provided that allow convenient handling of literals and addresses. Each pseudo instruction generates a single hardware instruction and may also generate a literal in the program literals area. The APAL pseudo instructions are:

LOOP
RAC
RACE
RALIT
RAPL
RAR
RASC
RDGC
RLIT

A *compound instruction* is a single hardware instruction that has a mnemonic consisting of two other instruction mnemonics separated by the _ character; for example, AMQ_QQ. The corresponding hardware instruction performs the functions of both of the instructions indicated in the compound mnemonic in the specified order (that is, leftmost first).

11.3.2 Activity control

A number of instructions that write into the DAP store do so under *activity control.* This means that only those bits of the addressed store plane corresponding to one bits in the A plane are written to. Store plane bits corresponding to zero bits in the A plane are not altered by writing to store under activity control.

11.3.3 The R plane and the orthogonal R plane

The function descriptions of a number of APAL instructions refer to the *R plane* or the *orthogonal R plane.*

The R plane may be regarded as a plane of bits, each row of which is equal to the contents of an MCU register specified in the instruction; that is, all the bits in column i are equal to bit i of the MCU register.

The orthogonal R plane may be regarded as a plane of bits, each column of which is equal to the contents of an MCU register specified in the instruction; that is all bits in row i are equal to bit i of the MCU register.

Note that these are only notionally store planes; that is, the planes are not formed explicitly in the DAP store.

11.3.4 APAL instructions

APALinstructions are described in alphabetical order in the following pages.

Function

The AB instruction sets every bit of the A plane to the value of one particular specified MCU register bit.

The ABN instruction sets each bit of the A plane to the inverse of the value of the same specified MCU register bit.

Syntax

AB ⟨MCU register⟩.⟨bit number⟩⟨modifier⟩?⟨step⟩?
ABN⟨MCU register⟩.⟨bit number⟩⟨modifier⟩?⟨step⟩?

where ⟨MCU register⟩, ⟨bit-number⟩, ⟨modifier⟩, and ⟨step⟩ together form an MCU register bit address (see section 11.2.4).

Binary instruction format

AB : 0101 .100 0RRR 0MMM +... .... –III IIII
ABN: 0101 .100 0RRR 1MMM +... .... –III IIII

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed by this instruction
2 *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the number of the MCU register bit addressed in this instruction
3 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the INT field of the instruction
5 The bit number to be addressed is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error occurs if an attempt is made to modify or step the effective INT value outside the range 0 to 63.

Example

```
AB M0.32 (M2)(+)  ! EACH BIT OF THE A PLANE IS SET TO
                  ! BIT i OF M0, WHERE i IS 32+ (INT
                  ! FIELD OF M2) + DO LOOP STEP VALUE.
```

ADD

Function

The ADD instruction adds the contents of two MCU registers, leaving the result in one of the registers. Arithmetic overflow is not detected.

Syntax

ADD<MCU register-1><MCU register-2>

where

<MCU register-1> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

<MCU register-2> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1110 1011 0RRR .MMM 01.. ....

Notes:

1. *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the register containing the first operand and the register that will contain the result
2. *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the register containing the second operand
3. Each operand is treated as a 64-bit unsigned integer. Any carry out of the most significant bit position is ignored, therefore arithmetic overflow is not detected

Possible run-time program errors

None.

Example

```
ADD M3 M5   !  M3 = M3 + M5
```

Function

The AEBS instruction sets each bit of the A plane to the logical equivalence of a specified MCU register bit and the corresponding bit of a specified store plane.

The AEBSN instruction sets each bit of the A plane to the logical non-equivalence (the inverse of the logical equivalence) of a specified MCU register bit and the corresponding bit of a specified store plane.

Syntax

AEBS ⟨MCU register⟩.⟨bit number⟩⟨plane⟩⟨modifier⟩?⟨step⟩?
AEBSN⟨MCU register⟩.⟨bit number⟩⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨MCU register⟩, ⟨bit number⟩, ⟨modifier⟩, and ⟨step⟩ together form an MCU register bit address (see section 11.2.4).

⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

AEBS : 0001 .100 0RRR 0MMM +AAA AAAA −III IIII
AESBN: 0001 .100 0RRR 1MMM +AAA AAAA −III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed by the instruction
2. *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the effective INT value
3. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
4. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
5. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how both the store plane address and MCU register bit number are to be stepped if the instruction appears in an APAL DO loop. If *step* is specified and is applicable, the current DO loop step value is effectively added to or subtracted from both the ADDR and INT fields of the instruction
6. The store plane addressed by the instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
7. The bit number addressed by the instruction is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to:

1. Modify or step the effective INT value outside the range zero to 63
2. Modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit

Example

```
AEBS M7.0 SPLANE+12   ! EACH BIT OF THE A PLANE IS SET
                      ! TO THE LOGICAL EQUIVALENCE OF
                      ! THE CORRESPONDING BIT OF STORE
                      ! PLANE SPLANE+12 AND BIT ZERO
                      ! OF M7
```

AF

Functions

The AF instruction sets each bit in the A plane to zero.

Syntax

AF

Binary instruction format

0100 .100 000. 0... ....

Possible run-time program errors

None.

Function

The AMB Instruction sets each bit of the A plane to the logical AND of itself and one particular specified MCU register bit.

The AMBN instruction sets each bit of the A plane to the logical AND of itself and the inverse of one particular specified MCU register bit.

Syntax

AMB <MCU register>.<bit number><modifier>?<step>?
AMBN<MCU register>.<bit number><modifier>?<step>?

where <MCU register>, <bit-number>, <modifier>, and <step> together form an MCU register bit address (see section 11.2.4).

Binary instruction format

AMB : 0101 .100 1RRR 0MMM +... .... −III IIII
AMBN: 0101 .100 1RRR 1MMM +... .... −III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed by this instruction
2. *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the number of the MCU register bit addressed in this instruction
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the INT field of the instruction
5. The bit number to be addressed is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective INT value outside the range zero to 63.

Example

```
AMB M2.30(-)  ! EACH BIT OF THE A PLANE IS SET TO THE
              ! LOGICAL AND OF ITSELF AND BIT i OF M3,
              ! WHERE i IS 30 - DO LOOP STEP VALUE
```

AMEBS
AMEBSN

Function

The AMEBS instruction sets each bit of the A plane to the logical AND of itself and the logical equivalence of the corresponding bit of a specified store plane with a specified MCU register bit.

The AMEBSN instruction sets each bit of the A plane to the logical AND of itself and the logical non-equivalence (inverse of logical equivalence) of the corresponding bit of a specified store plane with a specified MCU register bit.

Syntax

AMEBS <MCU register>.<bit number><plane><modifier> ?<step>?
AMEBSN<MCU register>.<bit number><plane><modifier> ?<step>?

where <MCU register>, <bit-number>, <modifier>, and <step> together form an MCU register bit address (see section 11.2.4).

<plane>, <modifier>, and <step> together form a store plane address (see section 11.2.3).

Binary instruction format

AMEBS : 0001 .100 1RRR 0MMM +AAA AAAA −III IIII
AMEBSN: 0001 .100 1RRR 1MMM +AAA AAAA −III IIII

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed in this section
2 *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the number of the MCU register bit addressed in this instruction
3 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the address of the store plane addressed in this instruction (see section 11.2.3)
4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
5 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how both the bit number and store plane address are to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from both the ADDR and INT fields of the instruction
6 The bit number addressed in this instruction is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified
7 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to:

1 Modify or step the effective INT value outside the range zero to 63
2 Modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit

Example

```
AMEBS M0.0 SPLANE (M3)  !  EACH BIT OF THE A PLANE IS
                        !  SET TO THE LOGICAL AND OF
                        !  ITSELF AND THE LOGICAL
                        !  EQUIVALENCE OF THE
                        !  CORRESPONDING BIT OF STORE
                        !  PLANE SPLANE WITH BIT i OF
                        !  M0, WHERE i IS THE INT FIELD
                        !  OF M3.
```

AMQ (non-shifting)
AMQN (non-shifting)

Function

The AMQ instruction sets each bit of the A plane to the logical AND of itself and the corresponding bit of the Q plane.

The AMQN instruction sets each bit of the A plane to the logical AND of itself and the inverse of the corresponding bit of the Q plane.

Syntax

AMQ
AMQN

Note that the instruction described in the next section also has the mnemonic AMQ but performs a different function. The assembler distinguishes between these instructions by using the fact that the non-shifting AMQ instructions have no operands.

Binary instruction format

AMQ : 1000 1100 1000 0... ....
AMQN: 1000 1100 1000 1... ....

Possible run-time program errors

None.

Function

The AMQ (shifting) instruction sets each bit of the A plane to the logical AND of itself and the corresponding bit of the Q plane as though the Q plane had first been shifted a single place in a specified direction, using a specified geometry. The Q plane is not changed by this instruction.

Syntax

AMQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩?

where

⟨direction⟩ ::= N | S | E | W | R0 | R1 | R2 | R3

⟨geometry⟩ ::= P | C | PC | CP

⟨count⟩ ::= ⟨number⟩

If all the operands are omitted or if the effective value of *count* is zero, the effect is the same as for the previously described AMQ (non-shifting) instruction.

Binary instruction format

1100 1100 1000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value. The values N, S, E, and W correspond to shifts to the north, south, east, and west respectively. The values R0, R1, R2, and R3 correspond to shifts in the direction specified in the DIRECTION field of *modifier*, rotated clockwise through zero, one, two, or three right angles respectively. The bit patterns corresponding to these options are given in sections 11.1.1.7. If *direction* is omitted, a zero value, corresponding to a shift specified by the DIRECTION field of *modifier*, is placed in the DIRECTION field of the instruction; *modifier* may not be omitted in this case

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value. The values of *geometry* are interpreted as follows:

geometry value	*Geometry of shift*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shifts

The bit patterns corresponding to these options are given in section 11.1.1.8.

If *geometry* is omitted, a zero value, indicating that the geometry of the shift is given by the GEOMETRY field of *modifier*, is placed in the GEOMETRY field of the instruction; modifier may not be omitted in such a case

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. If *count* is omitted, the COUNT field of the instruction is set to zero

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field of the instruction is set to zero

5 The direction of the shift is given by the effective DIRECTION value, which is the value in the DIRECTION field of the instruction modified by the DIRECTION field of *modifier*, if specified (see section 11.1.2.4)

6 The geometry of the shift is given by the effective GEOMETRY value, which is the value in the GEOMETRY field of the instruction modified by the GEOMETRY field of *modifier*, if specified (see section 11.1.2.4)

7 The magnitude of the shift is given by the effective COUNT value, which is the sum of the COUNT field of the instruction and the COUNT field of *modifier*, if specified, taken modulo 64. The effective COUNT value is interpreted as follows:

   (a) If the effective COUNT value is zero, the instruction is treated as a non-shifting instruction

   (b) If the effective COUNT value is not zero, a notional shift of one place is performed with the specified direction and geometry

8 The shift of the Q plane is only notional, that is, the Q plane is not physically shifted

Possible run-time program errors

A run-time program error will occur if the effective direction of the shift is a rotation of a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
AMQ E P 1   !  EACH BIT OF THE A PLANE IS SET TO THE
            !  LOGICAL AND OF ITSELF AND THE CORRESPONDING
            !  BIT OF THE Q PLANE SHIFTED ONE PLACE TO
            !  THE EAST USING PLANE GEOMETRY.
```

Function

The AMQ_QQ instruction is a compound instruction (see section 11.3.1), and is equivalent to an AMQ (shifting) instruction with a *count* of one followed by a QQ instruction with a *count* of one, the instruction pair being executed altogether *n* times, where *n* is the effective value of *count* in the AMQ_QQ instruction.

Syntax

AMQ__QQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩

AMQ__QQ ⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨direction⟩ ::= ⟨nesw⟩|R0|R1|R2|R3

⟨geometry⟩ ::= P|C|PC|CP

⟨count⟩ ::= ⟨number⟩

⟨nesw⟩ ::= N|E|S|W

Binary instruction format

1100 1110 1000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value.

The values N, S, E, and W correspond to north, south, east, and west shifts respectively. The values R0, R1, R2, and R3 correspond to a shift in the direction given by a clockwise rotation of zero, one, two, or three right angles of the direction specified by the DIRECTION field of *modifier*. The bit patterns corresponding to these options are given in section 11.1.1.7.

If *direction* is omitted, a zero value, implying a shift in the direction given by the DIRECTION field of *modifier*, is placed in the DIRECTION field of the instruction; *modifier* may not be omitted in this case

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

The *geometry* values are interpreted as follows:

Value	*Geometry of shift*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shifts

The bit patterns corresponding to these options are given in section 11.1.1.8. If *geometry* is omitted, a zero value, implying the geometry specified in the GEOMETRY field of *modifier*, is placed in the GEOMETRY field of the instruction; modifier may not be omitted in this case

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. If *count* is omitted, the COUNT field is set to zero

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

5 The direction of the shift is given by the effective DIRECTION value, which is the value in the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier*, if specified

6 The geometry of the shift is given by the effective GEOMETRY value, which is the value in the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier*, if specified

7 The magnitude of the shift is given by the effective COUNT value, which is the sum, modulo 64, of the COUNT field of the instruction and the INT field of *modifier*, if specified.

  If the effective COUNT value is zero, the instruction is treated as a non-shifting AMQ instruction; that is, no shift is performed

Possible run-time program errors

A run-time program error will occur if the effective direction is a rotation of a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

AMQ_QQ E C 10

Function

The AMR instruction sets each bit of the A plane to the logical AND of itself and the corresponding bit of the R plane (see section 11.3.3).

The AMRN instruction sets each bit of the A plane to the logical AND of itself and the inverse of the corresponding bit of the R plane

Syntax

AMR <MCU register>
AMRN<MCU register>

where

<MCU register> ::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

AMR : 0111 .100 1RRR 0... ....
AMRN: 0111 .100 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Example

```
AMR M4   ! EACH ROW OF THE A PLANE IS SET TO THE LOGICAL
         ! AND OF ITSELF AND M4.
```

AMRO
AMRNO

Function

The AMRO instruction sets each bit of the A plane to the logical AND of itself and the corresponding bit of the orthogonal R plane (see section 11.3.3).

The AMRNO instruction sets each bit of the A plane to the logical AND of itself and the inverse of the corresponding bit of the orthogonal R plane.

Syntax

AMRO ⟨MCU register⟩
AMRNO⟨MCU register⟩

where

⟨MCU register⟩ ::= M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

AMRO : 0011 .100 1RRR 0000 0ZZZ ZZZZ 0... ....
AMRNO: 0011 .100 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

Example

```
AMRO M4   !  EACH COLUMN OF THE A PLANE IS SET TO THE LOGICAL
          !  AND OF ITSELF AND M4.
```

Function

The AMS instruction sets each bit of the A plane to the logical AND of itself and the corresponding bit of a specified store plane.

The AMSN instruction sets each bit of the A plane to the logical AND of itself and the inverse of the corresponding bit of a specified store plane.

Syntax

AMS <plane><modifier>?<step>?
AMSN<plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 11.2.3).

Binary instruction format

AMS : 0000 .100 1.01 0MMM +AAA AAAA −... ....
AMSN: 0000 .100 1.01 1MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
AMS SPLANE (M1) (+)  ! EACH BIT OF THE A PLANE IS SET
                     ! TO THE LOGICAL AND OF ITSELF AND
                     ! THE CORRESPONDING BIT OF STORE
                     ! PLANE SPLANE + i, WHERE i IS THE
                     ! CURRENT DO LOOP STEP VALUE.
```

AND

Function

The AND instruction performs a logical AND on the contents of two MCU registers, leaving the result in one of the registers. The bits from either or both registers may optionally be inverted before participating in the logical AND.

Syntax

AND ⟨(inverted)MCU register-1⟩⟨(inverted)MCU register-2⟩

where

⟨(inverted)MCU register-1⟩ ::= ⟨(inverted)MCU register⟩

⟨(inverted)MCU register-2⟩ ::= ⟨(inverted)MCU register⟩

⟨(inverted)MCU register⟩ ::= ⟨MCU register⟩ | ⟨inverted MCU register⟩

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨inverted MCU register⟩ ::= M0N | M1N | M2N | M3N | M4N | M5N | M6N | M7N

Binary instruction format

1110 0000 1RRR .MMM 0... ....

See note 3 for possible variations on the OPERATION field value.

Notes:

1. *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the register containing the first operand and the register that will contain the result
2. *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the register containing the second operand
3. The OPERATION field of an assembled AND instruction may have any of four values, depending on which MCU operands are to be inverted, if at all:

OPERATION field	*Effect*
1110 0000 1	*MCU-register-1* AND *MCU-register-2*
1110 0001 0	*MCU-register-1* AND (NOT *MCU-register-2*)
1110 0011 1	(NOT *MCU-register-1*) AND *MCU-register-2*
1110 0101 1	(NOT *MCU-register-1*) AND (NOT *MCU-register-2*)

Possible run-time program errors

None.

Example

AND M2 M3N ! M2 = M2 AND (NOT M3).

AQ (non-shifting)
AQN (non-shifting)

Function

The AQ instruction sets each bit of the A plane to the corresponding bit of the Q plane.

The AQN instruction sets each bit of the A plane to the inverse of the corresponding bit of the Q plane

Syntax

AQ
AQN

Binary instruction format

AQ : 1000 1100 0000 0... ....
AQN: 1000 1100 0000 1... ....

Notes:

1 Note that the instruction described in the next section also has the mnemonic AQ but performs a different function. The assembler distinguishes between these instructions by using the fact that the non-shifting AQ instruction has no operands

Possible run-time program errors

None.

AQ (shifting)

Function

The AQ (shifting) instruction sets each bit of the A plane to the corresponding bit of the Q plane, as though the Q plane had first been shifted in a specified direction using a specified geometry. The Q plane is not altered by this instruction.

Syntax

AQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩?

where

⟨direction⟩ ::= N | S | E | W | R0 | R1 | R2 | R3

⟨geometry⟩ ::= P | C | PC | CP

⟨count⟩ ::= ⟨number⟩

If all the operands are omitted, or if the effective value of *count* is zero, the effect is the same as for the non-shifting AQ instruction.

Binary instruction format

1100 1100 0000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1. *direction* specifies the value in the DIRECTION field (D) of the construction, which is used to construct the effective DIRECTION value. The values N, S, E, and W correspond to shifts to the north, south, east, and west respectively. The values R0, R1, R2, and R3 correspond to shifts in the direction specified in the DIRECTION field of *modifier*, rotated clockwise through zero, one, two, or three right angles respectively. The bit patterns corresponding to these options are given in sections 11.1.1.7. If *direction* is omitted, a zero value, corresponding to a shift specified by the DIRECTION field of *modifier*, is placed in the DIRECTION field of the instruction; *modifier* must not be omitted in such a case
2. *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value. The values of *geometry* are interpreted as follows:

geometry value	*Geometry of shift*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shifts

   The bit patterns corresponding to these options are given in section 11.1.1.8.

   If *geometry* is omitted, a zero value, indicating that the geometry of the shift is given by the GEOMETRY field of *modifier*, is placed in the GEOMETRY field of the instruction; modifier must not be omitted in such a case
3. *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. If *count* is omitted, the COUNT field of the instruction is set to zero
4. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field of the instruction is set to zero
5. The direction of the shift is given by the effective DIRECTION value, which is the value in the DIRECTION field of the instruction modified by the DIRECTION field of *modifier*, if specified (see section 11.1.2.4)

6 The geometry of the shift is given by the effective GEOMETRY value, which is the value in the GEOMETRY field of the instruction modified by the GEOMETRY field of *modifier*, if specified (see section 11.1.2.4)

7 The magnitude of the shift is given by the effective COUNT value, which is the sum of the COUNT field of the instruction and the COUNT field of *modifier*, if specified, taken modulo 64. The effective COUNT value is interpreted as follows:
   (a) If the effective COUNT value is zero, the instruction is treated as a non-shifting instruction
   (b) If the effective COUNT value is not zero, a notional shift of one plane is performed with the specified direction and geometry

8 The shift of the Q plane is only notional; that is, the Q plane is not physically shifted

Possible run-time program errors

A run-time program error will occur if the effective direction of the shift is a rotation of a self modifier; that is, if *direction* is R1, R2 or R3 and *modifier* DIRECTION field is zero.

AQ_QQ

Function

The AQ_QQ instruction is a compound instruction (see section 11.3.1), and has the following effects:

1. The Q plane is shifted a specified number of places in a specified direction using a specified geometry
2. Each bit of both the A plane and Q plane is set to the corresponding bit of the shifted Q plane

Syntax

AQ__QQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩

AQ__QQ ⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨direction⟩ ::= ⟨nesw⟩|R0|R1|R2|R3

⟨geometry⟩ ::= P|C|PC|CP

⟨count⟩ ::= ⟨number⟩

⟨nesw⟩ ::= N|E|S|W

Binary instruction format

1100 1110 0000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1. *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value.

   The values N, S, E, and W correspond to north, south, east, and west shifts respectively. The values R0, R1, R2, and R3 correspond to a shift in the direction given by a clockwise rotation of zero, one, two, or three right angles of the direction specified by the DIRECTION field of *modifier*. The bit patterns corresponding to these options are given in section 11.1.1.7.

   If *direction* is omitted, a zero value, implying a shift in the direction given by the DIRECTION field of *modifier*, is placed in the DIRECTION field of the instruction. If *modifier* is omitted, *direction* must be specified
2. *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

   The *geometry* values are interpreted as follows:

Value	*Geometry of shift*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts

   Plane geometry for north and south shifts, cyclic geometry for east and west shifts

CP	Cyclic geometry for north and south shifts, plane geometry for east and west shifts

   The bit patterns corresponding to these options are given in section 11.1.1.8. If *geometry* is omitted, a zero value, implying the geometry specified in the GEOMETRY field of *modifier*, is placed in the GEOMETRY field of the instruction. If *modifier* is omitted, *geometry* must be specified.
3. *count* specifies the value of the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. If *count* is omitted, the COUNT field is set to zero
4. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

5 The direction of the shift is given by the effective DIRECTION value, which is the value in the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier*, if specified

6 The geometry of the shift is given by the effective GEOMETRY value, which is the value in the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier*, if specified

7 The magnitude of the shift is given by the effective COUNT value, which is the sum, modulo 64, of the COUNT field of the instruction and the INT field of *modifier*, if specified.

  If the effective COUNT value is zero, the instruction is treated as a non-shifting AMQ instruction; that is, no shift is performed

Possible run-time program errors

A run-time program error will occur if the effective direction is a rotation of a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
AQ_QQ E P 1   ! EACH BIT OF BOTH A AND Q PLANES IS
              ! SET TO THE CORRESPONDING BIT OF THE Q
              ! PLANE, SHIFTED ONE PLACE TO THE EAST
              ! USING PLANE GEOMETRY.
```

AR
ARN

Function

The AR instruction sets each bit of the A plane to the corresponding bit of the R plane (see section 11.3.3).

The ARN instruction sets each bit of the A plane to the inverse of the corresponding bit of the R plane.

Syntax

AR ⟨MCU register⟩
ARN⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

AR : 0111 .100 0RRR 0... ....
ARN: 0111 .100 0RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register used to form the R plane

Possible run-time program errors

None.

Example

```
AR M4   ! EACH ROW OF THE A PLANE IS SET EQUAL TO THE
        ! CONTENTS OF M4.
```

Function

The ARO instruction sets each bit of the A plane to the corresponding bit of the orthogonal R plane (see section 11.3.3).

The ARNO instruction sets each bit of the A plane to the inverse of the corresponding bit of the orthogonal R plane.

Syntax

ARO <MCU register>
ARNO<MCU register>

where

<MCU register>::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

ARO : 0011 .100 0RRR 0000 0ZZZ ZZZZ 0... ....
ARNO: 0011 .100 0RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register used to form the orthogonal R plane

Possible run-time program errors

None.

Example

```
ARO M4   !  EACH COLUMN OF THE A PLANE IS SET EQUAL TO
         !  THE CONTENTS OF M4.
```

AS
ASN

Function

The AS instruction sets each bit of the A plane to the corresponding bit of a specified store plane.

The ASN instruction sets each bit of the A plane to the inverse of the corresponding bit of a specified store plane.

Syntax

AS ⟨plane⟩⟨modifier⟩?⟨step⟩?
ASN⟨plane⟩⟨modifier⟩?⟨step⟩?
where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

AS : 0000 .100 0.01 0MMM +AAA AAAA –... ....
ASN: 0000 .100 0.01 1MMM +AAA AAAA –... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
AS SPLANE + 1 (M3) (–)   ! EACH BIT OF THE A PLANE IS SET
                         ! TO THE CORRESPONDING BIT OF
                         ! PLANE SPLANE + 1 – i, WHERE i IS
                         ! THE DO LOOP STEP VALUE.
```

AS_CF
ASN_CF

Function

The AS_CF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1. Each bit of the A plane is set to the corresponding bit of a specified store plane
2. Every bit of the C plane is set to zero

The ASN_CF is a compound instruction with the following effects:

1. Each bit of the A plane is set to the inverse of the corresponding bit of a specified store plane
2. Every bit of the C plane is set to zero

Syntax

AS_CF⟨plane⟩⟨modifier⟩?⟨step⟩?
ASN_CF⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section (11.2.3).

Binary instruction format

AS_CF: 0000 0101 0.01 0MMM +AAA AAAA –... ....
ASN_CF: 0000 0101 0.01 1MMM +AAA AAAA –... ....

Notes

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block, or between the program code datum and code limit.

Example

```
AS_CF 13 (+)  ! EACH BIT OF THE CPLANE IS SET TO ZERO,
              ! AND EACH BIT OF THE A PLANE IS SET
              ! TO THE CORRESPONDING BIT OF PLANE 13+i,
              ! WHERE i IS THE DO LOOP STEP VALUE.
```

AT

Function

The AT instruction sets every bit of the A plane to one.

Syntax

AT

Binary instruction format

0100 .100 000. 1... ....

Possible run-time program errors

None.

Function

The CF instruction sets every bit of the C plane to zero.

Syntax

CF

Binary instruction format

0100 1001 000. 0.. ....

Possible run-time program errors

None.

CPCA
CPCAN

Function

The CPCA instruction sets each bit of the C plane equal to the carry bit resulting from adding the corresponding bits of the C and A planes.

The CPCAN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the C plane and the inverse of the corresponding bit of the A plane.

The sums of these additions are discarded.

Syntax

CPCA
CPCAN

Binary instruction format

CPCA : 1001 0001 1000 1... ....
CPCAN; 1001 0001 1000 0... ....

Possible run-time program errors

None.

Function

The CPCQ instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q planes. The sums of these additions are discarded.

Syntax

CPCQ

Binary instruction format

0100 1001 100. 0... ... ...

Possible run-time program errors

None.

CPCQA
CPCQAN

Function

The CPCQA instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C, Q, and A planes.

The CPCQAN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q planes and the inverse of the corresponding bit of the A plane.

The sums of these additions are discarded.

Syntax

CPCQA
CPCQAN

Binary instruction format

CPCQA : 1001 1001 1000 1... ....
CPCQAN: 1001 1001 1000 0... ....

Possible run-time program errors

None.

Function

The CPCQR instructions sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C, Q, and R planes (see section 11.3.3).

The CPCQRN instructions sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q planes and the inverse of the corresponding bit of the R plane.

The sums of these additions are discarded.

Syntax

CPCQR⟨MCU register⟩
CPCQRN⟨MCU register⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

CPCQR : 0111 1001 1RRR 0... ....
CPCQRN: 0111 1001 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

CPCQRO
CPCQRNO

Function

The CPCQRO instruction sets each bit of the C plane to the carry bit resulting from the corresponding bits of the C, Q, and orthogonal R planes (see section 11.3.3).

The CPCQRNO instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q, planes and the inverse of the corresponding bit of the orthogonal R plane.

The sums of these additions are discarded.

Syntax

CPCQRO ⟨MCU register⟩
CPCQRNO⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CPCQRO : 0011 1001 1RRR 0000 0ZZZ ZZZZ 0... ....
CPCQRNO: 0011 1001 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

Function

The CPCQS instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q planes and a specified store plane.

The CPCQSN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the C and Q planes and the inverse of the corresponding bit of a specified store plane.

The sums of these additions are discarded.

Syntax

CPCQS <plane><modifier>?<step>?
CPCQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address, (see section 11.2.3.7).

Binary instruction format

CPCQS : 0000 1001 1.01 0MMM + AAA AAAA – ... .
CPCQSN: 0000 1001 1.01 1MMM + AAA AAAA – ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CPCQS 28 (M4)  ! EACH BIT OF THE C PLANE IS SET TO THE
               ! CARRY BIT GENERATED BY ADDING IT TO
               ! THE CORRESPONDING BITS OF THE Q PLANE
               ! AND STORE PLANE 28+(ADDR FIELD OF M4).
```

CPCQT

Function

The CPCQT instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and Q planes and the corresponding bit of a notional plane all of whose bits are one. The sums of these additions are discarded.

Syntax

CPCQT

Binary instruction format

0100 1001 100. 1... ....

Possible run-time program errors

None.

Function

The CPCR instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and R planes (see section 11.3.3).

The CPCRN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the C plane and the inverse of the corresponding bit of the R plane.

The sums of these additions are discarded.

Syntax

CPCR ⟨MCU register⟩
CPCRN ⟨MCU register⟩

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

CPCR : 0111 0001 1RRR 0... ....
CPCRN: 0111 0001 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

CPCRO
CPCRNO

Function

The CPCRO instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C and orthogonal R planes (see section 11.3.3).

The CPCRNO instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the C plane and the inverse of the corresponding bit of the orthogonal R plane.

The sums of these additions are discarded.

Syntax

CPCRO ⟨MCU register⟩
CPCRNO ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CPCRO : 0011 0001 1RRR 0000 0ZZZ ZZZZ 0... ....
CPCRNO: 0011 0001 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register used to form the orthogonal R plane

Possible run-time program errors

None.

Function

The CPCS instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the C plane and a specified store plane.

The CPCSN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the C plane and the inverse of the corresponding bit of a specified store plane.

The sums of these additions are discarded.

Syntax

CPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?
CPCSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

CPCS : 0000 0001 1.01 0MMM + AAA AAAA – ... .
CPCSN: 0000 0001 1.01 1MMM + AAA AAAA – ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CPCS SPLANE (M2)  !  EACH BIT OF THE C PLANE IS SET TO
                  !  THE CARRY BIT GENERATED BY ADDING
                  !  IT TO THE CORRESPONDING BIT OF
                  !  STORE PLANE SPLANE.
```

CPQA
CPQAN

Function

The CPQA instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the Q and A planes.

The CPQAN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the Q plane and the inverse of the corresponding bit of the A plane.

The sums of these additions are discarded.

Syntax

CPQA
CPQAN

Binary instruction format

```
CPQA  : 1001 1001 0000 1...  ....  ....  ....  ....
CPQAN: 1001 1001 0000 0...  ....  ....  ....  ....
```

Possible run-time program errors

None.

CPQR
CPQRN

Function

The CPQR instruction sets each bit of the C plane to the carry bit resulting from adding the correspondig bits of the Q and R planes (see s3ection 11.3.3).

The CPQRN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the Q plane and the inverse of the corresponding bit of the R plane.

The sums of these additions are discarded.

Syntax

CPQR ⟨MCU register⟩
CPQRN ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CPQR : 0111 1001 0RRR 0... ....
CPQRN: 0111 1001 0RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instructions, which specifies the MCU register used to form the R plane

Possible run-time program errors

None.

CPQRO
CPQRNO

Function

The CPQRO instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the Q and orthogonal R planes (see section 11.3.3).

The CPQRNO instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the Q plane and the inverse of the corresponding bit of the orthogonal R plane.

The sums of these additions are discarded.

Syntax

CPQRO   <MCU register>
CPQRNO <MCU register>

where

<MCU register>::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CPQRO : 0011 1001 0RRR 0000 0ZZZ ZZZZ 0... ....
CPQRNO: 0011 1001 0RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

Function

The CPQS instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bits of the Q plane and a specified store plane.

The CPQSN instruction sets each bit of the C plane to the carry bit resulting from adding the corresponding bit of the Q plane and the inverse of the corresponding bit of a specified store plane.

The sums of these additions are discarded.

Syntax

CPQS ⟨plane⟩⟨modifier⟩?⟨step⟩?
CPQSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

CPQS : 0000 1001 0.01 0MMM +AAA AAAA −... ....
CPQSN: 0000 1001 0.01 1MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CPQS SPLANE (M4) (+)  ! EACH BIT OF THE C PLANE IS SET
                      ! TO THE CARRY BIT GENERATED BY
                      ! ADDING THE CORRESPONDING BITS
                      ! OF THE Q PLANE AND STORE PLANE
                      ! SPLANE + i, WHERE i IS THE DO
                      ! LOOP STEP VALUE.
```

CQ

Function

The CQ instruction sets each bit of the C plane to the corresponding bit of the Q plane.

Syntax

CQ

Binary instruction format

0100 1001 000. 1... ....

Possible run-time program errors

None.

Function

The CQPCA instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and A planes; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCAN instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the A plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCA
CQPCAN

Binary instruction format

CQPCA : 1001 0011 1000 1... ....
CQPCAN: 1001 0011 1000 0... ....

Possible run-time program errors

None.

CQPCQ

Function

The CQPCQ instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the C plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQ

Binary instruction format

0100 1011 100. 0... ....

Possible run-time program errors

None.

Function

The CQPCQA instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and A planes; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCQAN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the A plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQA
CQPCQAN

Binary instruction format

CQPCQA : 1001 1011 1000 1... ....
CQPCQAN: 1001 1011 1000 0... ....

Possible run-time program errors

None.

CQPCQR
CQPCQRN

Function

The CQPCQR instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and R planes (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCQRN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQR ⟨MCU register⟩
CQPCQRN ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CQPCQR : 0111 1011 1RRR 0... ....
CQPCQRN: 0111 1011 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Function

The CQPCQRO instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and orthogonal R planes (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCQRNO instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the orthogonal R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQRO ⟨MCU register⟩
CQPCQRNO ⟨MCU register⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

CQPCQRO : 0011 1011 1RRR 0000 0ZZZ ZZZZ 0... ....
CQPCQRNO: 0011 1011 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

CQPCQS
CQPCQSN

Function

The CQPCQS instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C plane and a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCQSN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?
CQPCQSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

```
CQPCQS : 0000 1011 1.01 0MMM  +AAA AAAA  –... ....
CQPCQSN: 0000 1011 1.01 1MMM  +AAA AAAA  –... ....
```

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CQPCQS 26 (M6) (+)    ! EACH BIT OF THE Q PLANE IS SET
                      ! TO THE SUM OF ITSELF, THE
                      ! CORRESPONDING BIT OF THE C PLANE,
                      ! AND THE CORRESPONDING BIT OF
                      ! STORE PLANE 26 + (ADDR FIELD OF
                      ! M6) + i, WHERE i IS THE DO LOOP
                      ! STEP VALUE. THE CARRY BIT IS
                      ! PLACED IN THE CORRESPONDING BIT
                      ! OF THE C PLANE.
```

Function

The CQPCQT instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C plane and a notional plane all of whose bits are one; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCQT

Binary instruction format

0100 1011 100. 1... ....

Possible run-time program errors

None.

CQPCR
CQPCRN

Function

The CQPCR instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and R planes (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCRN instruction sets each bit of the Q plan to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCR <MCU register>
CQPCRN <MCU register>

where

<MCU register>::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CQPCR : 0111 0011 1RRR 0... ....
CQPCRN: 0111 0011 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

CQPCRO
CQPCRNO

Function

The CQPCRO instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and orthogonal R planes (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCRNO instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the orthogonal R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCRO ⟨MCU register⟩
CQPCRNO ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CQPCRO : 0011 0011 1RRR 0000 0ZZZ ZZZZ 0... ....
CQPCRNO: 0011 0011 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

CQPCS
CQPCSN

Function

The CQPCS instruction sets each bit of the Q plane to the sum of the corresponding bits of the C plane and a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPCSN instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPCS <plane><modifier>?<step>?
CQPCSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 11.2.3).

Binary instruction format

CQPCS : 0000 0011 1.01 0MMM +AAA AAAA −... ....
CQPCSN: 0000 0011 1.01 1MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CQPCS SPLANE (M4) (-)  ! EACH BIT OF THE Q PLANE IS SET
                       ! TO THE SUM OF THE CORRESPONDING
                       ! BITS OF THE C PLANE AND STORE
                       ! PLANE SPLANE-i, WHERE i IS
                       ! THE DO LOOP STEP VALUE. THE
                       ! CARRY BIT IS PLACED IN THE
                       ! CORRESPONDING BIT OF THE C
                       ! PLANE.
```

CQPQA
CQPQAN

Function

The CQPQA instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the A plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPQAN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the A plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPQA
CQPQAN

Binary instruction format

CQPQA : 1001 1011 0000 1... ....
CQPQAN: 1001 1011 0000 0... ....

Possible run-time program errors

None.

CQPQR
CQPQRN

Function

The CQPQR instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the R plane (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPQRN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPQR ⟨MCU register⟩
CQPQRN ⟨MCU register⟩

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

CQPQR : 0111 1011 0RRR 0... ....
CQPQRN: 0111 1011 0RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Function

The CQPQRO instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the orthogonal R plane (see section 11.3.3); the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPQRNO instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the orthogonal R plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPQRO 〈MCU register〉
CQPQRNO 〈MCU register〉

where

〈MUC register〉::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

CQPQRO : 0011 1011 0RRR 0000 0ZZZ ZZZZ 0... ....
CQPQRNO: 0011 1011 0RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

CQPQS
CQPQSN

Function

The CQPQS instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

The CQPQSN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of a specified store plane; the carry bit resulting from the addition is placed in the corresponding bit of the C plane.

Syntax

CQPQS 〈plane〉〈modifier〉?〈step〉?
CQPQSN 〈plane〉〈modifier〉?〈step〉?

where 〈plane〉, 〈modifier〉, and 〈step〉 together form a store plane address (see section 11.2.3).

Binary instruction format

CQPQS : 0000 1011 0.01 0MMM +AAA AAAA −... ....
CQPQSN: 0000 1011 0.01 1MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
CQPQS 23 (M5) (+)  ! EACH BIT OF THE Q PLANE IS SET TO
                   ! THE SUM OF ITSELF AND THE
                   ! CORRESPONDING BIT OF STORE
                   ! PLANE 23 + (ADDR FIELD OF M5) + i,
                   ! WHERE i IS THE DO LOOP STEP VALUE.
                   ! THE CARRY BIT IS PLACED IN THE
                   ! CORRESPONDING BIT OF THE C PLANE.
```

Function

The CQ_QQN is a compound instruction (see section 11.3.1), and has the following effects:

1 Each bit of the C plane is set to the corresponding bit of the Q plane
2 Every bit of the Q plane is then inverted

Syntax

CQ_QQN

Binary instruction format

0100 1011 000. 1... ....

Possible run-time program errors

None.

CQVCQ

Function

The CQVCQ instruction adds together corresponding rows or columns of the C and Q planes, treating each pair of rows or columns as 64-bit unsigned integers.

The effective direction for the instruction specifies whether C and Q are added by rows or by columns and, effectively, specifies the direction in which carries will be propagated, as follows:

Effective direction	*Effect*
North	The C and Q planes are added by columns; carry bits propagate northwards (bit 63 of each column is the least significant bit position)
South	The C and Q planes are added by columns; carry bits propagate southwards (bit 0 of each column is the least significant bit position)
East	The C and Q planes are added by rows; carry bits propagate eastwards (bit 0 of each row is the least significant bit position)
West	The C and Q planes are added by rows; carry bits propagate westwards (bit 63 of each row is the least significant bit position). This is the common case, since it corresponds to the mapping of integer data onto the DAP store

Therefore, considering a single pair of corresponding Q and C plane rows, and assuming an effective direction of west, the effect of the CQVCQ instruction is as follows.

In the *i*th bit position (*i* going from 63 to 0, east to west):

1 Q_i becomes the modulo 2 sum of:
   (a) The original Q_i (that is, the Q_i at the start of the instruction)
   (b) The original C_i (that is, the C_i at the start of the instruction)
   (c) The new C_{i+1} (that is, the carry result just formed in the next less significant bit position)

2 C_i becomes the carry produced in forming the above sum; that is, C_i becomes one if and only if two or more of the original Q_i, the original C_i, and the new C_{i+1} are one

The carry-in bit in the least significant bit position (C_{i+1} when $i = 63$), is determined by the effective geometry for the instruction, which may be plane or cyclic. The common case is plane geometry, implying that the carry-in bit at the least significant bit position is zero; cyclic geometry implies that the carry bit produced in the most significant bit position will be added in at the least significant bit position.

The instruction also specifies the number of instruction cycles during which carries propagate from less significant bit positions. The extent to which carry bits propagate determines the bit positions in which the result has a defined value. The number of carry propagation cycles is specified by the count (the sum of *count* in the instruction and the COUNT field in the modifier, if specified).

The usual way to use this instruction is with plane geometry and west direction, and with the data area (the bits which contain data) in the least significant portion of each row. In this simple case the count specified below will ensure that the result in the data area has a defined and correct value; result bits in other (more significant) bit positions may not be defined.

Data length (in bits)	*Count*
64 (entire row)	16
32	8
16	4

A complete description of carry propagation follows. In this description the term row means either row or column depending on the direction of carry; east and west directions correspond to row, north and south directions correspond to column.

Carries propagate along each row independently and simultaneously. In each row the carry-out bit is initially defined only in certain positions known as *carry starting points* (see below). Results are defined only in bit positions that are more significant than a *carry starting point*. The following positions are *carry starting points:*

1 In plane geometry, the least significant edge
2 In plane or cyclic geometry, any bit position in which the initial values of the Q and C bits are equal

The number of defined result bits is four times the count specified by the instruction. In cyclic geometry, a row in which the Q and C planes are exact inverses of each other has no carry starting point and, therefore, will have no defined result bits.

The following example shows the addition of a single pair of corresponding Q and C plane rows. The effective DIRECTION is west, the effective GEOMETRY is plane, and the count is 2. The Q and C rows have a logical data structure as follows:

Bit positions	*Contents*
0 to 17	Undefined
18 to 24	7-bit data field
25	0 (zero)
26 to 55	Undefined
56 to 63	8-bit data field

At the end of the instruction the results bits in the two data fields (bits 18 to 24 and bits 56 to 63) receive defined values. This occurs in the following way:

1 Starting from bit 25 (equal value in Q and C bits) two cycles give seven defined results bits
2 Starting from the least significant edge, two cycles give eight defined results bits
3 All other results bits (including bit 25) receive undefined values

The C plane results in bits 18 and 56 indicate whether there was overflow in the seven-bit or eight-bit data field sums respectively.

Syntax

CQVCQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩

CQVCQ ⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨direction⟩ ::= ⟨nesw⟩|R0|R1|R2|R3

⟨geometry⟩ ::= P|C|PC|CP

⟨count⟩ ::= ⟨number⟩

⟨nesw⟩ ::= N|E|S|W

Binary instruction format

1101 1011 1000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value.

The values N, S, E, and W correspond to the plane orientations defined in the function description. The values R0, R1, R2, and R3 specify an orientation that is a clockwise rotation of zero, one, two, or three right angles of the orientation specified in the DIRECTION field of *modifier*. The bit patterns corresponding to these values are given in section 11.1.1.7.

If *direction* is omitted, the value R0 is assumed as default; if *modifier* is also omitted W is assumed as the default

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

The values are interpreted as follows:

Value	*Meaning*
P	Carry in bits for least significant bit position are zero
C	Carry in bits for least significant bit position are carry out bits from most significant bit position
PC	Effect as for P if effective DIRECTION is N or S; effect as for C if effective DIRECTION is E or W
CP	Effect as for C if effective DIRECTION is N or S; effect as for P if effective DIRECTION is E or W

The bit patterns corresponding to these options are given in section 11.1.1.8.

If *geometry* is omitted, the effective geometry is given by the modifier register; if no modifier is specified, plane geometry is assumed

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. The value that is actually assembled into the COUNT field is *count* + 1.

If *count* is omitted, a zero value is assumed as default

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If modifier is omitted, the MOD field is set to zero

5 The orientation of the Q and C planes for the addition is given by the effective DIRECTION value, which is given by the contents of the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier* (see section 11.1.2.4), if specified. If the effective DIRECTION value is self, the result of the instruction is undefined

6 The value of the carry in bits at the least significant bit position is given by the effective GEOMETRY value, which is given by the contents of the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier* (see section 11.1.2.4), if specified

7 The number of instruction cycles for which carry bits are allowed to propagate is given by one less than the effective COUNT value, which is the sum of the COUNT field of the instruction and the COUNT field of *modifier*, if specified. An effective COUNT value of zero has the same effect as an effective COUNT value of one

Possible run-time program errors

A run-time program error will occur if an attempt is made to rotate a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
CQVCQ W P 8  ! Q AND C PLANES ARE ADDED AS 64 ROWS OF
             ! INTEGERS CONSIDERED AS APAL INTEGER
             ! VALUES. 8 INSTRUCTION CYCLES GUARANTEES
             ! THAT THE RIGHTMOST 32 BITS ARE CORRECT.
             ! THE Q PLANE IS SET TO THE SUMS; THE C
             ! PLANE IS SET TO THE CARRIES.
```

Function

The CVCQ instruction adds together corresponding rows or columns of the C and Q planes, treating each pair of rows or columns as 64-bit unsigned integers. The resulting carries appear in the C plane; the Q plane is unchanged by the instruction.

The effective direction for the instruction specifies whether the C and Q planes are added by rows or by columns and, effectively, specifies the direction in which carries will be propagated, as follows:

Effective direction	*Effect*
North	The C and Q planes are added by columns; carry bits propagate northwards (bit 63 of each column is the least significant bit position)
South	The C and Q planes are added by columns; carry bits propagate southwards (bit 0 of each column is the least significant bit position)
East	The C and Q planes are added by rows; carry bits propagate eastwards (bit 0 of each row is the least significant bit position)
West	The C and Q planes are added by rows; carry bits propagate westwards (bit 63 of each row is the least significant bit position). This is the common case, since it corresponds to the mapping of integer vector data onto the DAP store

Therefore, considering a single pair of corresponding Q and C plane rows, and assuming an effective direction of west, the effect of the CVCQ instruction is to set each bit of the C plane row to the carry bit produced by adding it to the corresponding bit of the Q plane row and the carry bit produced by the addition for the previous bit position; that is, in the *i*th bit position (i running from 0 to 63), C_i becomes one if and only if two or more one bits are present in the original Q_i, the original C_i, and the new C_{i+1}.

The carry-in bit in the least significant bit position (C_{i+1} when $i = 63$), is determined by the effective geometry for the instruction, which may be plane or cyclic. The usual option is plane geometry, implying that the carry-in bit at the least significant bit position is zero; cyclic geometry implies that the carry bit produced in the most significant bit position will be added in at the least significant bit position.

The instruction also specifies the number of instruction cycles during which carries propagate from less significant bit positions. The extent to which carry bits propagate determines the bit positions in which the result has a defined value. The number of carry propagation cycles is specified by the count (the sum of *count* in the instruction and the COUNT field in the modifier, if specified).

The usual way to use this instruction is with plane geometry and west direction, and with the data area (the bits which contain data) in the least significant portion of each row. In this simple case the count specified below will ensure that the result in the data area has a defined and correct value; result bits in other (more significant) bit positions may not be defined.

Data length (in bits)	*Count*
64 (entire row)	16
32	8
16	4

A complete description of carry propagation follows. In this description the term row means either row or column depending on the direction of carry; east and west directions correspond to row, north and south directions correspond to column.

Carries propagate along each row independantly and simultaneously. In each row the carry-out bit is initially defined only in certain positions known as *carry starting points* (see below). Results are defined only in bit positions that are more significant than a carry starting point. The following positions are carry starting points:

1 In plane geometry, the least significant edge

2 In plane or cyclic geometry, any bit position in which the initial values of the Q and C bits are equal

The number of defined result bits is four times the count specified by the instruction. In cyclic geometry, a row in which the Q and C planes are exact inverses of each other has no carry starting point and, therefore, will have no defined result bits.

The following example shows the addition of a single pair of corresponding Q and C plane rows. The effective DIRECTION is west, the effective GEOMETRY is plane and the count is 2. The Q and C rows have a logical data structure as follows:

Bit positions	*Contents*
0 to 17	Undefined
18 to 24	7-bit data field
25	0 (zero)
26 to 55	Undefined
56 to 63	8-bit data field

At the end of the instruction the results bits in the two data fields (bits 18 to 24 and bits 56 to 63) receive defined values. This occurs in the following way:

1 Starting from bit 25 (equal value in Q and C bits) two cycles give seven defined results bits

2 Starting from the least significant edge two cycles give eight defined results bits

3 All other results bits (including bit 25) receive undefined values

The C plane results in bits 18 and 56 indicate whether there was overflow in the seven-bit or eight-bit data field sums respectively.

Syntax

CVCQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩

CVCQ ⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨direction⟩ ::= ⟨nesw⟩|R0|R1|R2|R3

⟨geometry⟩ ::= P|C|PC|CP

⟨count⟩ ::=⟨number⟩

⟨nesw⟩ ::= N|E|S|W

Binary instruction format

1101 1001 1000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective GEOMETRY value.

The values N, S, E, and W correspond to the plane orientations defined in the function description. The values R0, R1, R2, and R3 specify an orientation that is a clockwise rotation of zero, one, two, or three right angles of the orientation specified in the DIRECTION field of *modifier*. The bit patterns corresponding to these values are given in section 11.1.1.7.

If *direction* is omitted, the value R0 is assumed as default; if modifier is also omitted, W is assumed as the default

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

The values are interpreted as follows:

Value	*Meaning*
P	Carry in bits for least significant bit position are zero
C	Carry in bits for least significant bit position are carry out bits from most significant bit position
PC	Effect as for P if effective DIRECTION is N or S; effect as for C if effective DIRECTION is E or W
CP	Effect as for C if effective DIRECTION is N or S; effect as for P if effective DIRECTION is E or W

The bit patterns corresponding to these options are given in section 11.1.1.8.

If *geometry* is omitted, the effective geometry is given by the modifier register; if no modifier is given, plane geometry is assumed

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. The value that is actually assembled into the COUNT field is *count* + 1.

If *count* is omitted, a zero value is assumed as default

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If modifier is omitted, the MOD field is set to zero

5 The orientation of the Q and C planes for the addition is given by the effective DIRECTION value, which is given by the contents of the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier* (see section 11.1.2.4), if specified. If the effective DIRECTION value is self, the result of the instruction is undefined

6 The value of the carry in bits at the least significant bit position is given by the effective GEOMETRY value, which is given by the contents of the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier* (see section 11.1.2.4), if specified

7 The number of instruction cycles for which carry bits are allowed to propagate is given one less than the effective COUNT value, which is the sum of the COUNT field of the instruction and the COUNT field of *modifier*, if specified. An effective COUNT value of zero has the same effect as an effective COUNT value of one

Possible run-time program errors

A run-time program error will occur if an attempt is made to rotate a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
CVCQ W P 8  ! Q AND C PLANES ARE ADDED AS 64 ROWS OF
            ! INTEGERS CONSIDERED AS APAL INTEGER
            ! VALUES 8. INSTRUCTION CYCLES GUARANTEES
            ! THAT THE RIGHTMOST 32 BITS ARE CORRECT.
            ! THE C PLANE IS SET TO THE RESULTING
            ! CARRIES; THE Q PLANE IS UNCHANGED.
```

DECR

Function

The DECR instruction decreases the value in a specified MCU register by one. Arithmetic overflow is not detected.

Syntax

DECR <MCU register>

where

<MCU register>: = M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1110 1111 1RRR .... 01.. ....

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose value is to be decremented
2. The value in the specified MCU register is treated as a 64-bit unsigned integer. Any carry out of the most significant bit position is ignored, therefore arithmetic overflow is not detected

Possible run-time program errors

None.

Example

DECR M7 ! M7 = M7 − 1

Function

The DO instruction indicates the start of an *APAL DO loop*, and also specifies how many times the instruction sequence within the loop is to be executed.

An APAL DO loop is a sequence of up to 60 APAL instructions, excluding the DO instruction itself, that is to be executed repeatedly the number of times specified in the DO instruction (see the syntax description below). The first instruction in the loop is the instruction that immediately follows the DO instruction.

The last instruction in a DO loop may be identified in either of two ways:

1. The first occurrence of the pseudo instruction LOOP following a DO instruction establishes that the instruction preceding LOOP is the last instruction in the loop. LOOP itself generates no code; it is merely an indicator of the end of a DO loop
2. The first occurrence of a labelled instruction following a DO instruction establishes that labelled instruction as the last instruction in the loop. A null label, consisting of a colon with no preceding identifier, may be used for this purpose. Note that the label need not occur on the same source line as the instruction that it labels; it may be separated from the instruction by comment lines, or lines containing only labels. For example, the following are all equivalent, the AMS instruction being the last instruction in the loop in every case:

```
    DO 32 TIMES    DO 32 TIMES    DO 32 TIMES          DO 32 TIMES
    QA             QA             QA                   QA
    AMS 0 (M1+)    :              :              LAST: AMS 0 (M1+)
    LOOP           AMS 0(M1+)     !NEXT IS LAST
                                  AMS 0 (M1+)

    DO 32 TIMES                   DO 32 TIMES
    QA                            QA
    :                             :AMS 0 (M1+)
LAST: AMS 0(M1+)
```

The instructions inside an APAL DO loop (that is, instructions after the DO instruction and up to and including the last instruction) may be any of the instructions described in this section except the DO, JSL, or JESL instructions. In particular, this implies that APAL DO loops can not be nested. The instructions have the same effect as if they were not in a DO loop, except that the INCREMENT/DECREMENT field may cause stepping of store addresses and/or MCU register bit numbers on each execution of the loop (see section 11.2.2). The J, JE, EXIT, or STOP instructions will prematurely terminate the DO loop.

Instructions in a DO loop are loaded into the MCU instruction buffer during the first pass through the loop. Consequently, the instructions do not have to be re-fetched from store on subsequent passes through the loop, thus reducing instruction fetch overheads.

Syntax

The DO instruction may have either of the following forms:

DO⟨number⟩⟨modifier⟩?⟨times⟩?
DO⟨modifier⟩⟨times⟩?

where

⟨times⟩ ::= TIMES

Binary instruction format

1111 0011 .... .MMM ...L LLLL LIII IIII

Notes:

1. *number* specifies the value in the INT field (I) of the instruction, which is used to determine the number of times that the DO loop is to be executed
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. The value in the L field of the instruction represents the number of instructions in the DO loop, excluding the DO instruction itself. This value is inserted by the assembler, which counts the number of instructions down to the end of the loop. If this value is greater than 60, the effect of the DO instruction is undefined
4. The number of times that the DO loop is to be executed is given by the effective INT value, which is the sum of the INT field of the instruction and bits 57 to 63 of *modifier*, if specified. The maximum number of times the loop can be executed is therefore 254 (that is, the sum of two seven-bit numbers)

Possible run-time program errors

A run-time program error will occur if an attempt is made to execute a DO loop inside another DO loop.

The effect of a DO loop whose effective INT value is zero is undefined.

Examples

```
  DO 32 TIMES         ! THE LOOP IS EXECUTED 32 TIMES.
  QS  0 (M2) (+)      ! THESE STORE PLANE ADDRESSES ARE INCREASED
: SQ  0 (M3) (+)      ! BY ONE PLANE EACH TIME THE LOOP IS
                      ! EXECUTED.

  DO   0 (M1) TIMES   ! THE LOOP IS EXECUTED i TIMES, WHERE i IS
: SF   31 (M2) (-)    ! IS THE VALUE OF THE INT FIELD OF M1.
```

Function

The EQV instruction sets each bit of a specified MCU register to the logical equivalence of itself and the corresponding bit of another MCU register.

Syntax

EQV⟨MCU register-1⟩⟨MCU register-2⟩

where

⟨MCU register-1⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨MCU register-2⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

1110 0100 1RRR .MMM 0... ....

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the MCU register containing the first operand and the register that will contain the result

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand

Possible run-time program errors

None.

EXIT

Function

The EXIT instruction loads an instruction address into the program counter from a specified MCU register, thereby causing a transfer of control.

The EXIT instruction is the normal way of returning control from a subroutine to the point from which the subroutine was entered (via a JE or JESL instruction). By software convention, all user written DAP programs are subroutines, and therefore EXIT is always used for normal return of control at the end of execution.

Syntax

EXIT⟨MCU register⟩?⟨number⟩?

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

1111 0110 0... .RRR .... XXXX XXXX

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register from which the instruction takes the value that is loaded into the program counter. If *MCU-register* is omitted, the program counter is loaded from M0 by default

2 *number*, which must be in the range zero to 255, specifies the value in the X field of the instruction, and represents an optional offset from the instruction in the saved program counter value. If *number* is omitted, a zero value is assumed

3 The value loaded into the program counter is:

number + (least significant 20 bits of *MCU-register*) + 1

Before transferring control, the JSL and JESL instructions save the correct program counter value (the address of the JSL or JESL instruction itself) in a specified MCU register. Therefore the EXIT instruction adds one to this value in order to return control to the instruction following the JSL or JESL instruction that caused the original transfer of control. *number* is specified if control is to be returned to an instruction more than one instruction beyond the JSL or JESL instruction

4 The transfer of control between APAL and DAP FORTRAN and/or 2900 FORTRAN is described in more detail in Chapter 9

Possible run-time program errors

A run-time error will occur if the address loaded into the program counter is less than the value in the program code datum or greater than the value in the program code limit; that is, the final address must be an instruction address.

Examples

```
EXIT        ! PROGRAM COUNTER = (LEAST SIGNIFICANT 20 BITS
            ! OF M0) +1.

EXIT 40     ! PROGRAM COUNTER = 40 + (LEAST SIGNIFICANT 20
            ! BITS OF M0) + 1.
```

INCR

Function

The INCR instruction increases the value in a specified MCU register by one. Arithmetic overflow is not detected.

Syntax

INCR⟨MCU register⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1110 1000 0RRR .... 00.. ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose value is to be incremented

2 The value of the specified MCU register is treated as a 64-bit unsigned integer. Any carry out of the most significant bit position is ignored, therefore arithmetic overflow is not detected

Possible run-time program errors

None.

Example

INCR M6 ! M6 = M6 + 1

Function

The J instruction loads an instruction address into the program counter, thereby causing control to be transferred to that instruction. The instruction to which control is transferred must be in the same code section as the J instruction.

Syntax

J<code label name><label offset>?

J<star><label offset>

where <code label name> (or <star>) and <label offset> together form an instruction address as described in section 11.2.5.

Binary instruction format

1111 0100 .... JJJJ JJJJ JJJJ JJJJ JJJJ

Notes

1 The instruction address specifies the value in the J field of the instruction, which is the value that will be loaded into the program counter (relative to the program code datum)

Possible run-time program errors

A run-time program error will occur if the address in the J field is not the address of an instruction in the same code section.

Examples

```
J LABI         ! CONTROL IS TRANSFERRED TO THE INSTRUCTION
               ! LABELLED LABI

J  LABI + 4    ! CONTROL IS TRANSFERRED TO THE FOURTH
               ! INSTRUCTION AFTER THE INSTRUCTION LABELLED
               ! LABI

J * - 12       ! CONTROL IS TRANSFERRED TO THE TWELFTH
               ! INSTRUCTION BEFORE THE J INSTRUCTION
```

JE

Function

The JE instruction loads an instruction address into the program counter, thereby causing control to be transferred to that instruction. The instruction to which control is transferred is identified relative to the first instruction or a named entry point in another code section.

Syntax

JE⟨code section name⟩⟨section offset⟩?

JE⟨entry point name⟩⟨section offset⟩?

where ⟨section offset⟩::= +⟨number⟩

⟨code section name⟩ (or ⟨entry point name⟩) and ⟨section offset⟩ together form an instruction address as described in section 11.2.5.

Binary instruction format

1111 0100 .... JJJJ JJJJ JJJJ JJJJ JJJJ

Notes:

1 The instruction address specifies the value in the J field of the instruction, which is the value that will be loaded into the program counter (relative to the program code datum)

Possible run-time program errors

A run-time program error will occur if the address in the J field is less than the value of the program code datum or greater than the value of the program code limit.

Examples

```
JE SECTION3 + 9  ! CONTROL IS TRANSFERRED TO THE 10TH
                 ! INSTRUCTION OF CODE SECTION SECTION3.

JE EP2           ! CONTROL IS TRANSFERRED TO THE INSTRUCTION
                 ! AT ENTRY POINT EP2.
```

Function

The JESL instruction transfers control to a named code section or entry point (see the JE instruction), but before loading the appropriate instruction address into the program counter, the current value of the program counter is saved in a specified MCU register. This value can be used by a subsequent EXIT instruction to return control to the instruction following the JESL instruction that caused the transfer.

The JESL or JSL instructions are the normal means of calling a subroutine.

Syntax

JESL⟨MCU register⟩?⟨code section name⟩⟨section offset⟩?

JESL⟨MCU register⟩?⟨entry point name⟩⟨section offset⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

and ⟨code section name⟩ (or ⟨entry point name⟩) and ⟨section offset⟩ together form an instruction address as described in section 11.2.5.

Binary instruction format

1111 0101 0RRR JJJJ JJJJ JJJJ JJJJ JJJJ

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register in which the current program counter value is to be saved. The program counter value is held in the least significant 20 bits of *MCU-register*; all remaining bits are set to zero.

   If *MCU-register* is omitted, the current program counter value is saved in M0 by default

2. The instruction address specifies the value in the J field of the instruction, which is the value that will be loaded into the program counter (relative to the program code datum)

Possible run-time program errors

A run-time program error will occur if the address in the J field is less than the value of the program code datum or greater than the value of the program code limit.

Example

```
JESL SECTION3 + 9  ! AS FOR JE, EXCEPT THAT THE CURRENT VALUE
                   ! OF THE PROGRAM COUNTER IS SAVED IN M0.
```

JSL

Function

The JSL instruction transfers control to another instruction in the same code section (see the J instruction), but before loading the appropriate instruction address into the program counter the current value of the program counter is saved in a specified MCU register. The value may be used in a subsequent EXIT instruction to return control to the instruction following the JSL instruction.

The JSL or JESL instructions are the normal means of calling a subroutine.

Syntax

JSL⟨MCU register⟩?⟨code label name⟩⟨label offset⟩?

JSL⟨MCU register⟩?⟨star⟩⟨label offset⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨code label name⟩ (or ⟨star⟩) and ⟨label offset⟩ together form an instruction address as described in section 11.2.5.

Binary instruction format

1111 0101 0RRR JJJJ JJJJ JJJJ JJJJ JJJJ

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register in which the current program counter value is to be saved. The program counter value is held in the least significant 20 bits of *MCU-register*; all remaining bits are set to zero.

  If *MCU-register* is omitted, the current program counter value is saved in M0 by default

2 The instruction address specifies the value in the J field of the instruction, which specifies the value that will be loaded into the program counter (relative to the program code datum)

Possible run-time program errors

A run-time program error will occur if the address in the J field is not the address of an instruction in the same code section.

Function

The LOOP instruction is a pseudo instruction that is used to indicate the end of an APAL DO LOOP (see the DO instruction).

LOOP generates no code and may not be labelled.

Syntax

LOOP

NAND

Function

The NAND instruction performs a logical NAND on the contents of two MCU registers, leaving the result in one of the registers. The bits from either or both of the registers may be optionally inverted before they participate in the logical NAND.

Syntax

NAND<(inverted) MCU register-1><(inverted) MCU register-2>

where

<(inverted) MCU register-1> ::= <(inverted) MCU register>

<(inverted) MCU register-2> ::= <(inverted) MCU register>

<(inverted) MCU register> ::= <MCU register> | <inverted MCU register>

<MCU register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

<inverted MCU register> ::= M0N | M1N | M2N | M3N | M4N | M5N | M6N | M7N

Binary instruction format

1110 0111 0RRR .MMM 0... ....

See note 3 for possible variations on the OPERATION field value.

Notes:

1. *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the register containing the first operand and the register that will contain the result
2. *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand
3. The OPERATION field of the assembled NAND instruction may have any of four values, depending on whether either of the MCU registers is to be inverted:

OPERATION field	*Effect*
1110 0111 0	*MCU-register-1* NAND *MCU-register-2*
1110 0110 1	*MCU-register-1* NAND (NOT *MCU-register-2*)
1110 0100 0	(NOT *MCU-register-1*) NAND *MCU-register-2*
1110 0010 0	(NOT *MCU-register-1*) NAND (NOT *MCU-register-2*)

Possible run-time program errors

None.

Example

NAND M1N M4N ! M1 = (NOT M1) NAND (NOT M4)

Function

The NEQ instruction sets each bit of a specified MCU register to the logical non-equivalence of itself and the corresponding bit of another MCU register.

Syntax

NEQ<MCU register-1><MCU register-2>

where

<MCU register-1>::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

<MCU register-2>::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1110 0011 0RRR .MMM 0... ....

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the MCU register containing the first operand and the register that will contain the result

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand

Possible run-time program errors

None.

Example

NEQ M0 M2 ! M0 = M0 NEQ M2

NOR

Function

The NOR instruction performs a logical NOR on two MCU registers, leaving the result in one of the registers. The bits from either or both of the registers may optionally be inverted before they participate in the logical NOR.

Syntax

NOR⟨(inverted) MCU register-1⟩⟨(inverted) MCU register-2⟩

where

⟨(inverted) MCU register-1⟩ ::=⟨(inverted) MCU register⟩

⟨(inverted) MCU register-2⟩ ::=⟨(inverted) MCU register⟩

⟨(inverted) MCU register⟩ ::=⟨MCU register⟩|⟨inverted MCU register⟩

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨inverted MCU register⟩::=M0N|M1N|M2N|M3N|M4N|M5N|M6N|M7N

Binary instruction format

1110 0101 1RRR .MMM 0... ....

See note 3 for possible variations on the OPERATION field value.

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the MCU register containing the first operand and the register that will contain the result

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand

3 The OPERATION field of an assembled NOR instruction may have any of four values, depending on which MCU registers are to be inverted, if at all:

OPERATION field	*Effect*
1110 0101 1	*MCU-register-1* NOR *MCU-register-2*
1110 0011 1	*MCU-register-1* NOR (NOT *MCU-register-2*)
1110 0001 0	(NOT *MCU-register-1*) NOR *MCU-register-2*
1110 0000 1	(NOT *MCU-register-1*) NOR (NOT *MCU-register-2*)

Possible run-time program errors

None.

Example

NOR M3 M6N ! M3 = M3 NOR (NOT M6)

Function

The NULL instruction has no effect.

Syntax

NULL

Binary instruction format

1111 1111 1111 ... .

Notes:

1 The instruction whose bits are all zero is also a NULL instruction

Possible run-time program errors

None.

OR

Function

The OR instruction performs a logical OR on two MCU registers, leaving the result in one of the registers. The bits from either or both registers may optionally be inverted before they participate in the logical OR.

Syntax

OR〈(inverted) MCU register-1〉〈(inverted) MCU register-2〉

where

〈(inverted) MCU register-1〉::=〈(inverted) MCU register〉

〈(inverted) MCU register-2〉 ::=〈(inverted) MCU register〉

〈(inverted) MCU register〉::=〈MCU register〉|〈inverted MCU register〉

〈MCU register〉::=M0|M1|M2|M3|M4|M5|M6|M7

〈inverted MCU register〉::=M0N|M1N|M2N|M3N|M4N|M5N|M6N|M7N

Binary instruction format

1110 0010 0RRR .MMM 0... ....

See note 3 for possible variations on the OPERATION field value.

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the MCU register containing the first operand and the register that will contain the result

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand

3 The OPERATION field of an assembled OR instruction may have any of four values, depending on which MCU registers are to be inverted, if at all:

OPERATION field	*Effect*
1110 0010 0	*MCU-register-1* OR *MCU-register-2*
1110 0100 0	*MCU-register-1* *OR* (NOT *MCU-register-2*)
1110 0110 1	(NOT *MCU-register-1*) OR *MCU-register-2*
1110 0111 0	(NOT *MCU-register-1*) OR (NOT *MCU-register-2*)

Possible run-time program errors

None.

Example

OR M0N M1 ! M0 = (NOT M0) OR M1

Function

The PCHECK instruction checks that a specified store plane is stored with correct parity. If correct parity is found, the PCHECK instruction is effectively a null instruction; if incorrect parity is found, DAP processing is interrupted and a parity failure is signalled.

Syntax

PCHECK⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address as described in section 11.2.3.

Binary instruction format

0000 .000 ..01 .MMM +AAA AAAA ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

Store parity failure.

Example

PCHECK 0 (M6)

QA
QAN

Function

The QA instruction sets each bit of the Q plane to the corresponding bit of the A plane.

The QAN instruction sets each bit of the Q plane to the inverse of the corresponding bit of the A plane.

Syntax

QA
QAN

Binary instruction format

QA : 1001 0010 0000 1... ....
QAN: 1001 0010 0000 0... ....

Possible run-time program errors

None.

Function

The QA_CF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1. Each bit of the Q plane is set to the corresponding bit of the A plane
2. Each bit of the C plane is set to zero

The QAN_CF is a compound instruction, and has the following effects:

1. Each bit of the Q plane is set to the inverse of the corresponding bit of the A plane
2. Each bit of the C plane is set to zero

Syntax

QA_CF
QAN_CF

Binary instruction format

QA__CF : 1001 0011 0000 1... ....
QAN__CF: 1001 0011 0000 0... ....

Possible run-time program errors

None.

QB
QBN

Function

The QB instruction sets every bit of the Q plane to the value of one particular specified MCU register bit.

The QBN instruction sets every bit of the Q plane to the inverse of one particular specified MCU register bit.

Syntax

QB ⟨MCU register⟩.⟨bit number⟩⟨modifier⟩?⟨step⟩?
QBN⟨MCU register⟩.⟨bit number⟩⟨modifier⟩?⟨step⟩?

where ⟨MCU register⟩, ⟨bit-number⟩, ⟨modifier⟩, and ⟨step⟩ together form an MCU register bit address (see section 11.2.4).

Binary instruction format

```
QB  : 0101 0010 0RRR 0MMM +...  .... -III IIII
QBN: 0101 0010 0RRR 1MMM +...  .... -III IIII
```

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed by this instruction
2. *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the effective INT value
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the INT field of the instruction
5. The bit number to be addressed is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective INT value outside the range zero to 63.

Example

```
QB M4.0 (M1) (+)  !  EACH BIT OF THE Q PLANE IS SET TO
                  !  BIT (INT FIELD OF M1) + i OF M4,
                  !  WHERE i IS THE DO LOOP STEP VALUE
```

Function

The QC instruction sets each bit of the Q plane to the corresponding bit of the C plane.

The QCN instruction sets each bit of the Q plane to the inverse of the corresponding bit of the C plane.

Syntax

QC
QCN

Binary instruction format

QC : 0100 0010 100. 0... ....
QCN: 0100 0011 100. 1... ....

Possible run-time program errors

None.

QC_CF

Function

The QC_CF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1. Each bit of the Q plane is set to the corresponding bit of the C plane
2. Every bit of the C plane is then set to zero

Syntax

QC_CF

Binary instruction format

0100 0011 100. 0... ....

Possible run-time program errors

None.

QEBS
QEBSN

Function

The QEBS instruction sets each bit of the Q plane to the logical equivalence of the corresponding bit of a specified store plane and a specified MCU register bit.

The QEBSN instruction sets each bit of the Q plane to the logical equivalence of the inverted corresponding bit of a specified store plane and a specified MCU register bit.

Syntax

QEBS ⟨MCU register⟩.⟨bit number⟩⟨plane⟩⟨modifier⟩?⟨step⟩?
QEBSN ⟨MCU register⟩.⟨bit number⟩⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨MCU register⟩, ⟨bit-number⟩, ⟨modifier⟩, and ⟨step⟩ together form an MCU register bit address (see section 11.2.4).

⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

QEBS : 0001 0010 0RRR 0MMM +AAA AAAA −III IIII
QEBSN: 0001 0010 0RRR 1MMM +AAA AAAA −III IIII

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed in this instruction
2 *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the effective INT value
3 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
5 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how both the bit number and store plane address are to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from both the ADDR and INT fields of the instruction
6 The bit number addressed in this instruction is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified
7 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to:

1 Modify or step the effective INT value outside the range zero to 63
2 Modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit

Example

```
QEBS M4.0 SPLANE (M3) (+)   ! EACH BIT OF THE Q PLANE IS
                            ! SET TO THE LOGICAL
                            ! EQUIVALENCE OF THE
                            ! CORRESPONDING BIT OF STORE
                            ! PLANE SPLANE + i AND BIT
                            ! (INT FIELD OF M3) + i OF M4,
                            ! WHERE i IS THE DO LOOP
                            ! STEP VALUE.
```

Function

The QF instruction sets every bit of the Q plane to zero.

Syntax

QF

Binary instruction format

0100 0010 000. 0... ....

Possible run-time program errors

None.

QF_CF

Function

The QF_CF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1 Every bit of the Q plane is set to zero
2 Every bit of the C plane is set to zero

Syntax

QF_CF

Binary instruction format

0100 0011 000. 0... ....

Possible run-time program errors

None.

QPCA
QPCAN

Function

The QPCA instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and A planes.

The QPCAN instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the A plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCA
QPCAN

Binary instruction format

QPCA : 1001 0010 1000 1... ....
QPCAN: 1001 0010 1000 0... ....

Possible run-time program errors

None.

QPCQ

Function

The QPCQ instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the C plane. The carry bits resulting from these additions are discarded.

Syntax

QPCQ

Binary instruction format

0100 1010 100. 0... ....

Possible run-time program errors

None.

QPCQA
QPCQAN

Function

The QPCQA instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and A planes.

The QPCQAN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the A plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCQA
QPCQAN

Binary instruction format

QPCQA : 1001 1010 1000 1... ....
QPCQAN: 1001 1010 1000 0... ....

Possible run-time program errors

None.

QPCQR
QPCQRN

Function

The QPCQR instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and R planes (see section 11.3.3).

The QPCQRN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCQR ⟨MCU register⟩
QPCQRN ⟨MCU register⟩

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

```
QPCQR  : 0111 1010 1RRR 0... .... .... .... ....
QPCQRN: 0111 1010 1RRR 1... .... .... .... ....
```

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

QPCQRO
QPCQRNO

Function

The QPCQRO instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C and orthogonal R planes (see section 11.3.3).

The QPCQRNO instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of the orthogonal R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCQRO 〈MCU register〉
QPCQRNO 〈MCU register〉

where

〈MCU register〉 ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QPCQRO : 0011 1010 1RRR 0000 0ZZZ ZZZZ 0... ....
QPCQRNO: 0011 1010 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

QPCQS
QPCQSN

Function

The QPCQS instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C plane and a specified store plane.

The QPCQSN instruction sets each bit of the Q plane to the sum of itself, the corresponding bit of the C plane, and the inverse of the corresponding bit of a specified store plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?
QPCQSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

QPCQS : 0000 1010 1.01 0MMM +AAA AAAA –... ....
QPCQSN: 0000 1010 1.01 1MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier* if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QPCQS SPLANE (M2)  ! EACH BIT OF THE Q PLANE IS SET TO
                   ! THE SUM OF ITSELF AND THE
                   ! CORRESPONDING BITS OF THE C PLANE
                   ! AND STORE PLANE SPLANE.
```

Function

The QPCQT instruction sets each bit of the Q plane to the sum of itself and the corresponding bits of the C plane and a notional plane all of whose bits are one. The carry bits resulting from these additions are discarded.

Syntax

QPCQT

Binary instruction format

0100 1010 100. 1... ....

Possible run-time program errors

None.

QPCR
QPCRN

Function

The QPCR instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and R planes (see section 11.3.3).

The QPCRN instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCR <MCU register>
QPCRN <MCU register>

where

<MCU register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QPCR : 0111 0010 1RRR 0... ....
QPCRN: 0111 0010 1RRR 1... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Function

The QPCRO instruction sets each bit of the Q plane to the sum of the corresponding bits of the C and orthogonal R planes (see section 11.3.3).

The QPCRNO instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of the orthogonal R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCRO <MCU register>
QPCRNO <MCU register>

where

<MCU register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QPCRO : 0011 0010 1RRR 0000 0ZZZ ZZZZ 0... ....
QPCRNO: 0011 0010 1RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value of the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

QPCS
QPCSN

Function

The QPCS instruction sets each bit of the Q plane to the sum of the corresponding bits of the C plane and a specified store plane.

The QPCSN instruction sets each bit of the Q plane to the sum of the corresponding bit of the C plane and the inverse of the corresponding bit of a specified store plane.

The carry bits resulting from these additions are discarded.

Syntax

QPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?
QPCSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

QPCS : 0000 0010 1.01 0MMM + AAA AAAA – ... .
QPCSN: 0000 0010 1.01 1MMM + AAA AAAA – ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value of the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which in the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QPCS S1 (–)  ! EACH BIT OF THE Q PLANE IS SET TO THE
             ! SUM OF THE CORRESPONDING BITS OF THE C
             ! PLANE AND STORE PLANE S1-i, WHERE i IS
             ! THE DO LOOP STEP VALUE.
```

Function

The QPQA instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the A plane.

The QPQAN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the A plane.

The carry bits resulting from these additions are discarded.

Syntax

QPQA
QPQAN

Binary instruction format

QPQA : 1001 1010 0000 1... ....
QPQAN: 1001 1010 0000 0... ....

Possible run-time program errors

None.

QPQR
QPQRN

Function

The QPQR instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the R plane (see section 11.3.3).

The QPQRN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPQR ⟨MCU register⟩
QPQRN ⟨MCU register⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QPQR : 0111 1010 0RRR 0... ....
QPQRN: 0111 1010 0RRR 1... ....

Notes:

1 *MCU-register* specifies the value of the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Function

The QPQRO instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of the orthogonal R plane (see section 11.3.3).

The QPQRNO instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of the orthogonal R plane.

The carry bits resulting from these additions are discarded.

Syntax

QPQRO <MCU register>
QPQRNO <MCU register>

where

<MCU register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QPQRO : 0011 1010 0RRR 0000 0ZZZ ZZZZ 0... ....
QPQRNO: 0011 1010 0RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

QPQS
QPQSN

Function

The QPQS instruction sets each bit of the Q plane to the sum of itself and the corresponding bit of a specified store plane.

The QPQSN instruction sets each bit of the Q plane to the sum of itself and the inverse of the corresponding bit of a specified store plane.

The carry bits resulting from these additions are discarded.

Syntax

QPQS <plane><modifier>?<step>?
QPQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 11.2.3).

Binary instruction format

QPQS : 0000 1010 0.01 0MMM +AAA AAAA −... ....
QPQSN: 0000 1010 0.01 1MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, and specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified (and applicable), the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed by the instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QPQS 49 (M6)  ! EACH BIT OF THE Q PLANE IS SET TO THE
              ! SUM OF ITSELF AND THE CORRESPONDING BIT
              ! OF STORE PLANE 49 + (ADDR FIELD OF M6).
```

Function

The QQ instruction shifts the Q plane a specified number of places in a specified direction, using a specified geometry.

Syntax

QQ <direction>?<geometry>?<count>?<modifier>

QQ <nesw><geometry><count>?

where

<direction> ::= <nesw>|R0|R1|R2|R3

<geometry> ::= P|C|PC|CP

<count> ::= <number>

<nesw> ::= N|E|S|W

Binary instruction format

1100 1010 .000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value.

The values N, S, E, and W correspond to north, south, east, and west shifts respectively. The values R0, R1, R2, and R3 specify a direction that is a clockwise rotation of zero, one, two or three right angles of the direction specified in the DIRECTION field of *modifier*. The bit patterns corresponding to these values are given in section 11.1.1.7.

If *direction* is omitted, a zero value, representing a shift in the direction specified in the DIRECTION field of *modifier*, is placed in the DIRECTION field of the instruction; *modifier* may not be omitted in this case

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

The values are interpreted as follows:

Value	*Geometry of shift*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shifts
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shifts

The bit patterns corresponding to these options are given in section 11.1.1.8.

If *geometry* is omitted, a zero value, representing the geometry specified in the GEOMETRY field of *modifier*, is placed in the GEOMETRY field of the instruction; *modifier* may not be omitted in this case

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. If *count* is omitted, the COUNT field is set to zero

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

5 The direction of the shift is given by the effective DIRECTION value, which is given by the contents of the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier* (see section 11.1.2.4), if specified.

If the effective direction of the shift is self, the instruction has no effect

6 The geometry of the shift is given by the effective GEOMETRY value, which is given by the content of the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier* (see section 11.1.2.4), if specified

7 The magnitude of the shift is given by the effective COUNT value, which is the sum, modulo 64, of the COUNT field of the instruction and the INT field of *modifier*, if specified.

   If the effective COUNT value is zero, the instruction has no effect

Possible run-time program errors

A run-time program error will occur if an attempt is made to rotate a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
QQ W P 1 (M2)  !  THE Q PLANE IS SHIFTED 1 + (INT FIELD OF
               !  M2) PLACES TO THE WEST, USING PLANE
               !  GEOMETRY.
```

Function

The QQN instruction inverts every bit of the Q plane.

Syntax

QQN

Binary instruction format

0100 1010 000. 1... ....

Possible run-time program errors

None.

QR
QRN

Function

The QR instruction sets each bit of the Q plane to the corresponding bit of the R plane (see section 11.3.3).

The QRN instruction sets each bit of the Q plane to the inverse of the corresponding bit of the R plane.

Syntax

QR ⟨MCU register⟩
QRN ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

```
QR  : 0111 0010 0RRR 0... .... .... .... ....
QRN: 0111 0010 0RRR 1... .... .... .... ....
```

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane

Possible run-time program errors

None.

Example

```
QR M3  ! EACH ROW OF THE Q PLANE IS SET EQUAL TO THE
       ! CONTENTS OF M3.
```

QRO
QRNO

Function

The QRO instruction sets each bit of the Q plane to the corresponding bit of the orthogonal R plane (see section 11.3.3).

The QRNO instruction sets each bit of the Q plane to the inverse of the corresponding bit of the orthogonal R plane.

Syntax

QRO 〈MCU register〉
QRNO 〈MCU register〉

where

〈MCU register〉::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

QRO : 0011 0010 0RRR 0000 0ZZZ ZZZZ 0... ....
QRNO: 0011 0010 0RRR 1000 0ZZZ ZZZZ 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the orthogonal R plane

Possible run-time program errors

None.

Example

```
QRO M5   ! EACH COLUMN OF THE Q PLANE IS SET TO THE
         ! CONTENTS OF M5.
```

QS
QSN

Function

The QS instruction sets each bit of the Q plane to the corresponding bit of a specified store plane.

The QSN instruction sets each bit of the Q plane to the inverse of the corresponding bit of a specified store plane.

Syntax

QS ⟨plane⟩⟨modifier⟩?⟨step⟩?
QSN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address as described in section 11.2.3.

Binary instruction format

QS : 0000 0010 0.01 0MMM +AAA AAAA −... ....
QSN: 0000 0010 0.01 1MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, and specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified (and applicable), the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed by the instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QS SPLANE (M2) (+)  ! EACH BIT OF THE Q PLANE IS SET TO
                    ! THE CORRESPONDING BIT OF STORE
                    ! PLANE SPLANE + i, WHERE i IS THE
                    ! DO LOOP STEP VALUE.
```

Function

The QS_AS instruction is a compound instruction (see section 11.3.1), and has the following effects:

1 Each bit of the Q plane is set to the corresponding bit of a specified store plane
2 Each bit of the A plane is set to the corresponding bit of the same store plane

The QSN_ASN instruction is a compound instruction with the following effects:

1 Each bit of the Q plane is set to the inverse of the corresponding bit of a specified store plane
2 Each bit of the A plane is set to the inverse of the corresponding bit of the same store plane

Syntax

```
QS_AS    ⟨plane⟩⟨modifier⟩?⟨step⟩?
QSN̄_ASN  ⟨plane⟩⟨modifier⟩?⟨step⟩?
```

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

```
QS_AS   : 0000 0110 0.01 0MMM  +AAA AAAA  –...  ....
QSN̄_ASN: 0000 0110 0.01 1MMM  +AAA AAAA  –...  ....
```

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QS_AS 20 (M2)   ! EACH BIT OF THE Q AND A PLANES IS SET
                ! TO THE CORRESPONDING BIT OF STORE PLANE
                ! 20 + (ADDR FIELD OF M2).
```

QS_CF
QSN̄_CF

Function

The QS_CF instruction is a compound instruction (see section 11.3.3), and has the following effects:

1 Each bit of the Q plane is set to the corresponding bit of a specified store plane
2 Every bit of the C plane is set to zero

The QSN_CF instruction is a compound instruction with the following effects:

1 Each bit of the Q plane is set to the inverse of the corresponding bit of a specified store plane
2 Every bit of the C plane is set to zero

Syntax

QS_CF ⟨plane⟩⟨modifier⟩?⟨step⟩?
QSN̄_CF ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

QS_CF : 0000 0011 0.01 0MMM +AAA AAAA –... ....
QSN̄_CF: 0000 0011 0.01 1MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
QS_CF SPLANE + 2 (M3) (–)  ! EACH BIT OF THE C PLANE IS
                           ! SET TO ZERO, AND EACH BIT OF
                           ! THE Q PLANE IS SET TO THE
                           ! CORRESPONDING BIT OF STORE
                           ! PLANE SPLANE + 2 – i, WHERE i IS
                           ! THE DO LOOP STEP VALUE.
```

Function

The QT instruction sets every bit of the Q plane to one.

Syntax

QT

Binary instruction format

0100 0010 000. 1... ....

Possible run-time program errors

None.

QT_CF

Function

The QT_CF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1 Every bit of the Q plane is set to one

2 Every bit of the C plane is set to zero

Syntax

QT_CF

Binary instruction format

0100 0011 000. 1... ....

Possible run-time program errors

None.

Function

The QVCQ instruction adds together corresponding rows or columns of the C and Q planes, treating each pair of rows or columns as 64-bit unsigned integers. The sums appear in the Q plane; the C plane is not altered by this instruction.

The effective direction for the instruction specifies whether the C and Q planes are added by rows or by columns and, effectively, specifies the direction in which carries will be propagated, as follows:

Effective direction	*Effect*
North	The C and Q planes are added by columns; carry bits propagate northwards (bit 63 of each column is the least significant bit position)
South	The C and Q planes are added by columns; carry bits propagate southwards (bit 0 of each column is the least significant bit position)
East	The C and Q planes are added by rows; carry bits propagate eastwards (bit 0 of each row is the least significant bit position)
West	The C and Q planes are added by rows; carry bits propagate westwards (bit 63 of each row is the least significant bit position). This is the common case, since it corresponds to the mapping of integer vector data onto the DAP store

Therefore, considering a single pair of corresponding Q and C plane rows, and assuming an effective direction of west, the effect of the QVCQ instruction is to set each bit of the Q plane row to the sum of itself, the corresponding bit of the C plane row, and the carry bit produced by the addition in the previous bit position; that is, in the *i*th bit position (*i* running from 63 to 0), Q_i becomes the modulo 2 sum of:

1. The original Q_i (that is, the Q_i at the start of the instruction)
2. The original C_i (that is, the C_i at the start of the instruction)
3. The new C_{i+1} (that is, the carry bit just formed in the next less significant bit position)

The carry-in bit in the least significant bit position (C_{i+1} when $i = 63$), is determined by the effective geometry for the instruction, which may be plane or cyclic. The common case is plane geometry, implying that the carry-in bit at the least significant bit position is zero; cyclic geometry implies that the carry bit produced in the most significant bit position will be added in at the least significant bit position.

The instruction also specifies the number of instruction cycles during which carries propagate from less significant bit positions. The extent to which carry bits propagate determines the bit positions in which the result has a defined value. The number of carry propagation cycles is specified by the count (the sum of *count* in the instruction and the COUNT field in the modifier, if specified).

The usual way to use this instruction is with plane geometry and west direction, and with the data area (the bits which contain data) in the least significant position of each row. In this simple case the count specified below will ensure that the result in the data area has a defined and correct value; result bits in other (more significant) bit positions may not be defined.

Data length (in bits)	*Count*
64 (entire row)	16
32	8
16	4

A complete description of carry propagation follows. In this description the term row means either row or column depending on the direction of carry; east and west directions correspond to row, north and south directions correspond to column.

Carries propagate along each row independantly and simultaneously. In each row the carry-out bit is initially defined only in certain positions known as *carry starting points* (see below). Results are defined only in bit positions that are more significant than a carry starting point. The following positions are carry starting points:

1 In plane geometry, the least significant edge
2 In plane or cyclic geometry, any bit position in which the initial values of the C and Q bits are equal

The number of defined result bits is four times the count specified by the instruction. In cyclic geometry, a row in which the Q and C planes are exact inverses of each other has no carry starting point and, therefore, will have no defined result bits.

The following example shows the addition of a single pair of corresponding Q and C plane rows. The effective DIRECTION is west, the effective GEOMETRY is plane and the count is 2. The Q and C rows have a logical data structure as follows:

Bit positions	*Contents*
0 to 17	Undefined
18 to 24	7-bit data field
25	0 (zero)
26 to 55	Undefined
56 to 63	8-bit data field

At the end of the instruction the results bits in the two data fields (bits 18 to 24 and bits 56 to 63) receive defined values. This occurs in the following way:

1 Starting from bit 25 (equal value in Q and C bits) two cycles give seven defined results bits
2 Starting from the least significant edge two cycles give eight defined results bits
3 All other results bits (including bit 25) receive undefined values

Syntax

QVCQ ⟨direction⟩?⟨geometry⟩?⟨count⟩?⟨modifier⟩

QVCQ ⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨direction⟩ ::= ⟨nesw⟩|R0|R1|R2|R3

⟨geometry⟩ ::= P|C|PC|CP

⟨count⟩ ::= ⟨number⟩

⟨nesw⟩ ::= N|E|S|W

Binary instruction format

1101 1010 1000 0MMM 0DDD .GGG 0CCC CCCC

Notes:

1 *direction* specifies the value in the DIRECTION field (D) of the instruction, which is used to construct the effective DIRECTION value.

The values N, S, E, and W correspond to the plane orientations defined in the function description. The values R0, R1, R2, and R3 specify an orientation that is a clockwise rotation of zero, one, two, or three right angles of the orientation specified in the DIRECTION field of *modifier*. The bit patterns corresponding to these values are given in section 11.1.1.7.

If *direction* is omitted, the value R0 is assumed as default

2 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which is used to construct the effective GEOMETRY value.

The values are interpreted as follows:

Value	*Meaning*
P	Carry in bits for least significant bit position or zero
C	Carry in bits for least significant bit position or carry out bits from most significant bit position
PC	Effect as for P if effective DIRECTION is N or S; effect as for C if effective DIRECTION is E or W
CP	Effect as for C if effective DIRECTION is N or S; effect as for P if effective DIRECTION is E or W

The bit patterns corresponding to these options are given in section 11.1.1.8.

If *geometry* is omitted, the effective geometry is given by the modifier register; if no modifier is given, plane geometry is assumed

3 *count* specifies the value in the COUNT field (C) of the instruction, which is used to construct the effective COUNT value. The value that is actually assembled into the COUNT field is *count* + 1.

If *count* is omitted, a zero value is assumed as default

4 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If modifier is omitted, the MOD field is set to zero

5 The orientation of the Q and C planes for the addition is given by the effective DIRECTION value, which is given by the contents of the DIRECTION field of the instruction, modified by the DIRECTION field of *modifier* (see section 11.1.2.4), if specified. If the effective DIRECTION value is self, the result of the instruction is undefined

6 The value of the carry in bits at the least significant bit position is given by the effective GEOMETRY value, which is given by the contents of the GEOMETRY field of the instruction, modified by the GEOMETRY field of *modifier* (see section 11.1.2.4), if specified

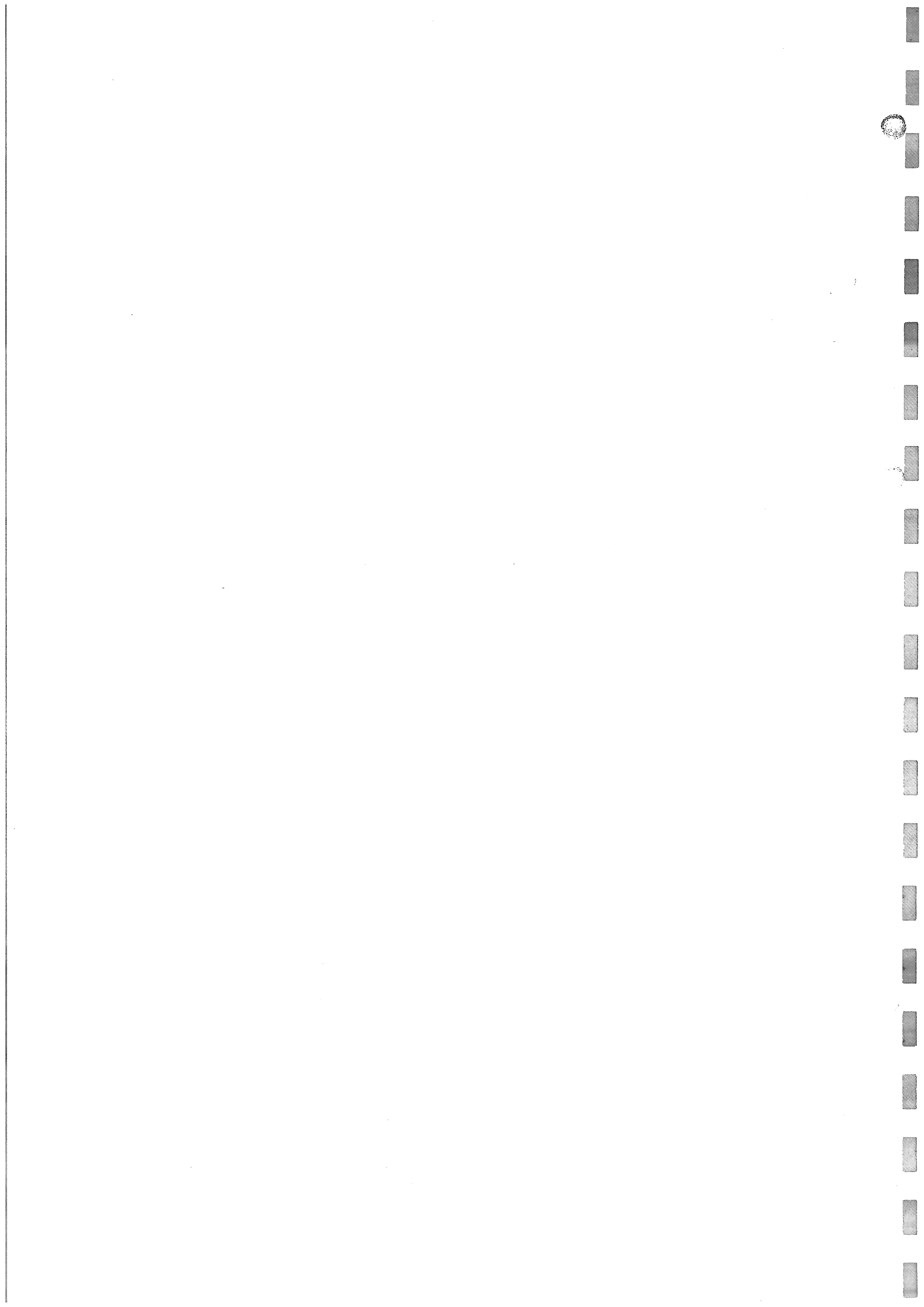
7 The number of instruction cycles for which carry bits are allowed to propagate is given by one less than the effective COUNT value, which is the sum of the COUNT field of the instruction and the COUNT field of *modifier*, if required. An effective COUNT value of zero has the same effect as an effective COUNT value of one

Possible run-time program errors

A run-time program error will occur if an attempt is made to rotate a self modifier; that is, *direction* and *modifier* may not both be omitted.

Example

```
QVCQ W P 8  !  Q AND C PLANES ARE ADDED AS 64 ROWS OF
            !  INTEGERS CONSIDERED AS APAL INTEGER
            !  VALUES. 8 INSTRUCTION CYCLES GUARANTEES
            !  THAT THE RIGHTMOST 32 BITS ARE CORRECT.
            !  THE Q PLANE IS SET TO THE SUMS; THE C
            !  PLANE IS UNCHANGED.
```



RAC

Function

The RAC instruction is a pseudo instruction (see section 11.3.1), and loads the address of an instruction in the same code section into a specified MCU register.

A literal is created to hold the address and an RX instruction is generated to load it from the literals area.

Syntax

RAC ⟨MCU register⟩⟨code label name⟩⟨label offset⟩?
RAC ⟨MCU register⟩⟨star⟩⟨label offset⟩?

where ⟨label offset⟩ ::= + ⟨number⟩ | − ⟨number⟩

Binary instruction format

See the RX instruction.

Possible run-time program errors

None.

Examples

```
RAC M4 LAB1 +2 !  LOADS THE ADDRESS OF THE SECOND
               !  INSTRUCTION AFTER THE INSTRUCTION
               !  LABELLED LAB1 INTO M4.

RAC M4 * + 2   !  LOADS THE ADDRESS OF THE SECOND
               !  INSTRUCTION FOLLOWING THIS RAC
               !  INSTRUCTION INTO M4.
```

RACE

Function

The RACE instruction is a pseudo instruction (see section 11.3.1), and loads the address of an instruction in a different code section into a specified MCU register.

As with the RAC instruction, a literal is created to hold the address, and an RX instruction is generated to load the address from the literals area.

Syntax

RACE ⟨MCU register⟩⟨code section name⟩⟨section offset⟩?
RACE ⟨MCU register⟩⟨entry point name⟩⟨section offset⟩?

where ⟨section offset⟩ ::= + ⟨number⟩

Binary instruction format

See the RX instruction.

Possible run-time program errors

None.

Examples

RACE M2 CODESEC + 1	! LOADS THE ADDRESS OF THE SECOND ! INSTRUCTION IN CODE SECTION CODESEC ! INTO M2.
RACE M0 ENTPT1	! LOADS THE ADDRESS OF ENTRY POINT ! ENTPT1 INTO M0.

RALIT

Function

The RALIT instruction is a pseudo instruction (see section 11.3.1), and loads the store address associated with a literal into a specified MCU register. The instruction creates a literal with the specified value and generates a RAX instruction to load the address of the literal into the specified MCU register.

Syntax

RALIT ⟨MCU register⟩⟨value⟩⟨size⟩?

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨value⟩⟨size⟩? is as defined in Chapter 4.

Binary instruction format

See the RAX instruction.

Possible run-time program errors

None.

Examples

```
RALIT M2 1247 (32)  ! THE ADDRESS OF THE LITERAL 1247 IS
                    ! LOADED INTO M2.
RALIT M7 18.42      ! THE ADDRESS OF LITERAL 18.42 IS
                      LOADED INTO M7.
```

RANO

Function

The RANO instruction sets a specified MCU register to the logical AND of the inverse of all 64 columns of the A plane.

Syntax

RANO <MCU register>

where

<MCU register> ::= M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

1001 .RRR .001 ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register into which the result is placed

Possible run-time program errors

None.

Example

RANO M4

RAPL

Function

The RAPL instruction is a pseudo instruction (see section 11.3.1), and loads the plane part of the data address associated with a specified row into the ADDR field of a specified MCU register. Remaining bits of the MCU register are set to zero.

A literal is created to hold the address, and an RX instruction is generated to load the address from the literals area.

Syntax

RAPL ⟨MCU register⟩⟨row⟩

where

⟨MCU register⟩ ::= M0|M1|M2|M3|M4|M5|M6|M7

⟨row⟩ is defined in section 11.2.3.

Binary instruction format

See the RX instruction.

Possible run-time program errors

None.

Example

```
DATA SECT1
     FRED: PLANE*3
     TOM: 3.142
END
.
.
.
RAPL M3 TOM + 2   !  LOADS THE VALUE n + 5 INTO ADDR FIELD
                  !  OF M3. n IS THE PLANE ADDRESS OF DATA
                  !  SECTION SECT1, TOM HAS A PLANE
                  !  DISPLACEMENT OF 3 WITHIN SECT1, AND
                  !  ANOTHER DISPLACEMENT OF 2 PLANES IS
                  !  SPECIFIED IN THE INSTRUCTION.
```

RAR

Function

The RAR instruction is a pseudo instruction (see section 11.3.1), and loads the store address of a specified item into a specified MCU register. The item is a variable identified by a row.

If necessary, a literal is created to hold the address, and an RX instruction is generated to load the address from the literals area.

Syntax

RAR ⟨MCU register⟩⟨row⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨row⟩ is defined in section 11.2.3.

Binary instruction format

See the RX instruction.

Possible run-time program errors

None.

Example

```
DATA DATASEC
     FRED: 200*PLANE
     JOE: PLANE
END
RAR M4 JOE .37   !  LOADS THE ADDRESS OF ROW 37 OF JOE
                 !  INTO M4.
```

Function

The RASC instruction is a pseudo instruction (see section 11.3.1), and loads the address of the start of the data section, or data part of a mixed section, associated with a specified row into a specified MCU register.

A literal is created to hold the address, and an RX instruction is generated to load the address from the literals area.

Syntax

RASC ⟨MCU register⟩⟨row⟩

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨row⟩ is defined in section 11.2.3.

Binary instruction format

See the RX instruction.

Possible run-time program errors

None.

Example

```
RASC M6 VAR1  !  THE START ADDRESS OF THE DATA SECTION IN
              !  WHICH VAR1 IS DECLARED IS LOADED INTO M6.
```

RAX

Function

The RAX instruction loads the address of a row into a specified MCU register. The plane part of the address is loaded into the ADDR field (bits 44 to 57) of the specified MCU register; the row part of the address is loaded into the INT field (bits 58 to 63). All other bits of the specified MCU register are set to zero.

Syntax

RAX ⟨MCU register⟩⟨row⟩⟨modifier⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨row⟩ and ⟨modifier⟩ together form a store row address (see section 11.2.3).

Binary instruction format

0010 .RRR 0110 0MMM 1AAA AAAA 1III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the register into which the store address is to be loaded
2. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. The plane part of the store address, loaded into the ADDR field of *MCU-register*, is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified. The effective ADDR value is not checked to see if it is a valid plane number, and is truncated to 14 bits
5. The row part of the store address, loaded into the INT field of *MCU-register*, is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified.

   If the effective INT value is greater than 63, carry across a store plane boundary is performed; that is, the effective INT value is taken modulo 64 and carry bits are added into the effective ADDR value

Possible run-time program errors

None.

Example

```
RAX M2 14.2 (M6)  !  ADDR FIELD OF M2 = 14 + (ADDR FIELD OF
                  !  M6) + CARRY OUT OF INT FIELD
                  !  OF M2 = 2 + (INT FIELD OF M6)
```

RAY

Function

The RAY instruction loads an address into a specified MCU register. The instruction interprets the address as a column address. The plane part of the column address is loaded into the ADDR field (bits 44 to 57) of the register, and the column part is loaded into the INT field (bits 58 to 63). All other bits of the specified MCU register are set to zero.

Syntax

RAY ⟨MCU register⟩⟨column⟩⟨modifier⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨column⟩ and ⟨modifier⟩ together form a store column address (see section 11.2.3).

Binary instruction format

0010 .RRR 0010 0MMM 1AAA AAAA 1III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the register into which the store address is to be loaded
2. *column* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. The plane part of the store address, loaded into the ADDR field of *MCU-register*, is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified. The effective ADDR value is not checked to see if it is a valid plane number, and is truncated to 14 bits
5. The column part of the store address, loaded into the INT field of *MCU-register*, is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if the effective INT value is greater than 63; that is, a store column address may not carry across a store plane boundary.

Example

```
RAY M4 12.6 (M5)  !  ADDR FIELD OF M4 = 12 + (ADDR FIELD OF
                  !  M5) INT FIELD OF M4 = 6 + (INT FIELD
                  !  OF M5).
```

RD

Function

The RD instruction loads the effective ADDR and INT values corresponding to a store address into the ADDR and INT fields of a specified MCU register. All other bits of the MCU register are set to zero.

Syntax

RD ⟨MCU register⟩⟨plane⟩⟨integer offset⟩?⟨modifier⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨integer offset⟩ ::= .⟨number⟩

⟨plane⟩ and ⟨modifier⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

0010 .RRR 0.00 0MMM 1AAA AAAA 1III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register into which the address is loaded
2. *plane* specifies the values in the ADDR field (A) and INT field (I) of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
3. *integer offset* specifies a value, taken modulo 64, that is placed into the INT field (I) of the instruction
4. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
5. The plane part of the store address loaded into the ADDR field of *MCU register* is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified. The effective ADDR value is not checked to ensure that it is a valid plane number, and is truncated to 14 bits
6. The effective INT value, loaded into the INT field of *MCU register*, is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified. If the effective INT value is greater than 63, the carry into the ADDR field is ignored

Possible run-time program errors

None.

Example

```
RD M4 SPLANE + 6.19 (M3)  !  ADDR FIELD OF M4 = PLANE ADDRESS
                          !  OF SPLANE + 6 + (ADDR FIELD OF
                          !  M3) .INT FIELD OF M4 = 19 + INT FIELD OF M3).
```

Function

THE RDGC instruction is a pseudo instruction (see section 11.3.1), and loads a specified MCU register with effective direction, geometry, and count fields. The instruction generates an RD instruction.

Syntax

RDGC ⟨MCU register⟩⟨nesw⟩⟨geometry⟩⟨count⟩?

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨geometry⟩ ::= P | C | PC | CP

⟨count⟩ ::= ⟨number⟩

⟨nesw⟩ ::= N | E | S | W

Binary instruction format

0010 .RRR 0.00 0000 1DDD 0GGG 1CCC CCCC

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the register into which the effective DIRECTION, GEOMETRY, and COUNT fields are to be loaded

2 *nesw* specifies the value in the DIRECTION field (D) of the instruction, which specifies the effective DIRECTION value

The values N, S, E, and W correspond to north, south, east and west shifts respectively. The bit patterns corresponding to these options are given in section 11.1.1.7

3 *geometry* specifies the value in the GEOMETRY field (G) of the instruction, which specifies the effective GEOMETRY value.

The values are interpreted as follows:

Value	*Geometry*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shift
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shift

The bit patterns corresponding to these options are given in section 11.1.1.8

4 *count* specifies the value in the COUNT field (C) of the instruction, which specifies the effective COUNT value. If *count* is omitted, a zero count is assumed

Possible run-time program errors

None.

Example

RDGC M1 N P 3

RF

Function

The RF instruction sets every bit in a specified MCU register to zero.

Syntax

RF ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

1110 0001 1RRR .... 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose contents are set to zero

Possible run-time program errors

None.

Function

The RLIT instruction is a pseudo instruction (see section 11.3.1), and allows the user to load a literal value directly into a specified MCU register.

Syntax

RLIT <MCU register><value><size>?

where

<MCU register>::=M0|M1|M2|M3|M4|M5|M6|M7

<value><size>? is as defined in Chapter 4.

Binary instruction format

If *value* is less than 14 bits long, an RD instruction will be generated, otherwise a literal of the stated size is created and an RX instruction is generated to load it from the literals area.

Possible run-time program errors

None.

Example

```
RLIT M5 0.0 (64)   !  M5 IS LOADED WITH THE 64-BIT REAL
                   !  VALUE 0.0. THE VALUE WILL HAVE BEEN
                   !  GENERATED AS A LITERAL.
```

RQO

Function

The RQO instruction sets a specified MCU register to the logical AND of all columns in the Q plane.

Syntax

RQO ⟨MCU register⟩

where

⟨MCU register⟩::=M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1000 1RRR .001 ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that will contain the result

Possible run-time program errors

None.

RR
RRN

Function

The RR instruction sets one MCU register to the value of another MCU register.

The RRN instruction sets one MCU register to the inverse of the value of another register.

Syntax

RR ⟨MCU register-1⟩⟨MCU register-2⟩
RRN ⟨MCU register-1⟩⟨MCU register-2⟩?

where

⟨MCU register-1⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨MCU register-2⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

```
RR : 1110 0010 1RRR .MMM 0...  ....  .000  0000
RRN: 1110 0101 0RRR .MMM 0...  ....  ....  ....
```

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is to contain the result

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register holding the required value. If *MCU-register-2* is omitted in an RRN instruction, it is assumed to be the same as *MCU-register-1*; that is, the instruction will invert the specified MCU register

Possible run-time program errors

None.

Example

```
RR M0 M6     ! M0 = M6
RRN M2 M3    ! M2 = NOT (M3)
RRN M4       ! M4 = NOT (M4)
```

RS
RSO

Function

The RS instruction sets a specified MCU register to the logical AND of all the rows of a specified store plane.

The RSO instruction sets a specified MCU register to the logical AND of all the columns of a specified store plane.

Syntax

RS ⟨MCU register⟩⟨plane⟩⟨modifier⟩?⟨step⟩?
RSO ⟨MCU register⟩⟨plane⟩⟨modifier⟩?⟨step⟩?

where

⟨MCU register⟩ ::= M0|M1|M2|M3|M4|M5|M6|M7

⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

```
RS : 0010 .RRR 0101 0MMM  +AAA AAAA  -... ....
RSO: 0010 .RRR 0001 0MMM  +AAA AAAA  -... ....
```

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is to contain the result
2. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
5. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
RS M4 62 (M3) (-)  ! M4 IS SET TO THE LOGICAL AND OF ALL
                   ! COLUMNS IN STORE PLANE 62 + (ADDR
                   ! FIELD OF M3) - i, WHERE i IS THE DO
                   ! LOOP STEP VALUE
```

Function

The RT instruction sets every bit of a specified MCU register to one.

Syntax

RT ⟨MCU register⟩

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

Binary instruction format

1110 0110 0RRR .... 0... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bits are to be set to ones

Possible run-time program errors

None.

RX

Function

The RX instruction loads the contents of a specified store row into a specified MCU register.

Syntax

RX ⟨MCU register⟩⟨row⟩⟨modifier⟩?⟨step A⟩?

where

⟨MCU register⟩::=M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨row⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store row address (see section 11.2.3).

Binary instruction format

0010 .RRR 011S 0MMM +AAA AAAA −III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the register into which the contents of the addressed row are to be loaded
2. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.4)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies whether the store address is to be incremented or decremented if it appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which specifies whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the store address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the store address is stepped by a single plane)
5. The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
6. The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified.

   If the effective INT value is greater than 63, carry across a store plane boundary is performed; that is, the effective INT value is taken modulo 64 and carry bits are added into the effective ADDR value

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit.

Example

```
RX M1 SPLANE + 16.32 (M3) (+A)  ! M1 IS SET TO CONTENTS
                                ! OF ROW i OF STORE PLANE
                                ! j, WHERE:
                                !
                                ! i = 32 + (INT FIELD OF M3)
                                !
                                ! j = PLANE ADDRESS OF
                                ! SPLANE + 16 + n, WHERE n
                                ! IS THE CURRENT DO LOOP
                                ! STEP VALUE.
```

Function

The RY instruction loads the contents of a specified store column into a specified MCU register.

Syntax

RY ⟨MCU register⟩⟨column⟩⟨modifier⟩?⟨step A⟩?

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨column⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store column address (see section 11.2.3).

Binary instruction format

0010 .RRR 001S 0MMM +AAA AAAA −III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the register into which the contents of the addressed column are to be loaded
2. *column* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.5)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies whether the store address is to be incremented or decremented if it appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which specifies whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or − the SELECT field is set to 1 and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the store address is stepped by a single column).

   If *step A* is + A or − A, the SELECT field is set to zero and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the store address is stepped by a single plane)
5. The plane part of the required column address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
6. The column part of the required column address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to:

1 Modify or step the effective ADDR value outside the DAP program block or between the program code datum and code limit
2 Modify or step the effective INT value outside the range zero to 63

Example

```
RY M7 26.12 (M2)  !  M7 IS SET TO THE CONTENTS OF COLUMN
                  !  i OF STORE PLANE j, WHERE:
                  !
                  !  i = 12 + (INT FIELD OF M2)
                  !
                  !  j = 26 + (ADDR FIELD OF M2)
```

SAN

Function

The SAN instruction sets each bit of a specified store plane to the inverse of the corresponding bit of the A plane.

Syntax

SAN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1001 .000 .101 .MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area between plane 36 and the program code limit.

Example

```
SAN SPLANE + 9 (M6) (+)  ! EACH BIT OF STORE SPLANE + 9 + i
                         ! (WHERE i IS THE DO LOOP STEP
                         ! VALUE) IS SET TO THE INVERSE
                         ! OF THE CORRESPONDING BIT OF
                         ! THE A PLANE
```

Function

The SF instruction sets every bit in a specified store plane to zero.

Syntax

SF ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

0100 .000 .101 .MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area between plane 36 and the program code limit.

Example

```
SF 27 (M3)  !  EACH BIT OF STORE PLANE 27 + (ADDR FIELD
            !  OF M3) IS SET TO ZERO.
```

SHL

Function

The SHL instruction copies the value in a specified MCU register into another MCU register and shifts the latter register a specified number of places to the left. Zeros are shifted in on the right.

Syntax

SHL ⟨MCU register-1⟩⟨MCU register-2⟩?⟨count⟩

where

⟨MCU register-1⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨MCU register-2⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨count⟩::=⟨number⟩

Binary instruction format

1110 0010 1RRR .MMM 1.10 .... .CCC CCCC

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that will hold the result. If *MCU-register-2* is omitted, this register also holds the operand to be shifted

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, and if specified determines the MCU register from which the operand is taken. If *MCU-register-2* is omitted, the MOD field has the same value as the MCUR field

3 *count* specifies the number of places by which *MCU-register-1* is to be shifted. The value of *count* should be in the range 1 to 127.

The assembler adds one to *count* and places the resulting value in the COUNT field of the instruction. When the instruction is executed, the number of places shifted is one less than the value in the COUNT field; that is, the value written by the user is the value that is finally used, but is one less than the value assembled into the instruction.

Possible run-time program errors

None.

Example

```
SHL M1 M2 17  ! M1 = M2 SHIFTED 17 PLACES TO THE LEFT
SHL M4 12     ! M4 = M4 SHIFTED 12 PLACES TO THE LEFT
```

Function

The SHLC instruction copies the value in a specified MCU register into another MCU register and shifts the latter register cyclically a number of places to the left. Bits shifted off the left end of the MCU register are shifted in at the right.

Syntax

SHLC <MCU register-1><MCU register-2>?<count>

where

<MCU register-1>::=M0|M1|M2|M3|M4|M5|M6|M7

<MCU register-2>::=M0|M1|M2|M3|M4|M5|M6|M7

<count>::=<number>

Binary instruction format

1110 0010 1RRR .MMM 1.11 .... .CCC CCCC

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that will contain the result. If *MCU-register-2* is omitted, this register also contains the operand to be shifted

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, and if specified determines the MCU register from which the operand is taken. If *MCU-register-2* is omitted, the MOD field has the same value as the MCUR field

3 *count* specifies the number of places by which *MCU-register-1* is to be shifted. The value of *count* should be in the range 1 to 127.

The assembler adds one to *count* and places the resulting value in the COUNT field of the instruction. When the instruction is executed, the number of places shifted is one less than the value in the COUNT field; that is, the value written by the user is the value that is finally used, but is one less than the value assembled into the instruction

Possible run-time program errors

None.

Examples

```
SHLC M3 14        ! M3 = M3 ROTATED 14 PLACES TO THE LEFT
SHLC M1 M6 - 7    ! M1 = M6 ROTATED 7 PLACES TO THE LEFT.
```

SHR

Function

The SHR instruction copies the value in a specified MCU register into another MCU register and shifts the latter register a number of places to the right. Zeros are shifted in at the left of the register.

Syntax

SHR ⟨MCU register-1⟩⟨MCU register-2⟩?⟨count⟩

where

⟨MCU register-1⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨MCU register-2⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨count⟩::=⟨number⟩

Binary instruction format

1110 0010 1RRR .MMM 1.00 .... .CCC CCCC

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that will contain the result. If *MCU-register-2* is omitted, this register will also contain the operand to be shifted

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, and if specified determines the MCU register from which the operand is taken. If *MCU-register-2* is omitted, the MOD field has the same value as the MCUR field

3 *count* specifies the number of places by which *MCU-register-1* is to be shifted. The value of *count* should be in the range 1 to 127.

The assembler adds one to *count* and places the resulting value in the COUNT field of the instruction. When the instruction is executed, the number of places shifted is one less than the value in the COUNT field; that is, the value written by the user is the value that is finally used, but is one less than the value assembled into the instruction

Possible run-time program errors

None.

Example

SHR M1 M4 12 ! M1 = M4 SHIFTED 12 PLACES TO THE RIGHT

Function

The SHRC instruction copies the value in a specified MCU register into another MCU register and shifts the latter register cyclically a number of places to the right. Bits shifted out at the right of the MCU register are shifted in at the left.

Syntax

SHRC ⟨MCU register-1⟩⟨MCU register-2⟩?⟨count⟩

where

⟨MCU register-1⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨MCU register-2⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨count⟩::=⟨number⟩

Binary instruction format

1110 0010 1RRR .MMM 1.01 .... .CCC CCCC

Notes:

1 *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that will contain the result. If *MCU-register-2* is omitted, this register also contains the operand to be shifted

2 *MCU-register-2* specifies the value in the MOD field (M) of the instruction, and if specified determines the MCU register from which the operand is taken.

If *MCU-register-2* is omitted, the MOD field has the same value as the MCUR field

3 *count* specifies the number of places by which *MCU-register-1* is to be shifted. The value of *count* should be in the range 1 to 127.

The assembler adds one to *count* and places the resulting value in the COUNT field of the instruction. When the instruction is executed, the number of places shifted is one less than the value in the COUNT field; that is, the value written by the user is the value that is finally used, but is one less than the value assembled into the instruction

Possible run-time program errors

None.

Example

SHRC M1 10 ! M1 = M1 SHIFTED 10 PLACES TO THE RIGHT.

SIC

Function

The SIC instruction sets each bit of a specified store plane to the corresponding bit of the C plane under activity control (see section 11.3.2). That is, the assignment only takes place where the corresponding bit of the A plane is one; store plane bits corresponding to zero A plane bits are unchanged.

Syntax

SIC ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1011 0000 1101 .MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Example

SIC SPLANE (M4) (+) ! EACH BIT OF STORE PLANE SPLANE
! + *i* (WHERE *i* IS THE DO LOOP
! STEP VALUE) CORRESPONDING TO A
! ONE BIT IN THE A PLANE IS SET
! TO THE CORRESPONDING BIT OF
! THE C PLANE.

Function

The SICPCQS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bits of the C and Q planes under activity control (see section 11.3.2). That is, the store plane is written to only for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bit generated by each addition is placed in the corresponding bit of the C plane. The carries are not under activity control; that is, every bit of the C plane receives the carry bit resulting from the corresponding addition.

Syntax

SICPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1001 1101 0MMM +AAA AAAA –... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Example

```
SICPCQS 28 (M4)  ! EACH BIT OF STORE PLANE 28 + (ADDR
                 ! FIELD OF M4) THAT CORRESPONDS TO A
                 ! ONE BIT IN THE A PLANE IS SET TO
                 ! THE SUM OF ITSELF AND THE
                 ! CORRESPONDING BITS OF THE C AND Q
                 ! PLANES. EACH BIT OF THE C PLANE IS
                 ! SET TO THE CARRY BIT GENERATED BY
                 ! THE CORRESPONDING ADDITION.
```

SICPCS

Function

The SICPCS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bit of the C plane under activity control (see section 11.3.2). That is, the store plane is written to only for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bits generated by these additions are placed in the corresponding bits of the C plane. Carries are not under activity control; that is, every C plane bit receives the carry bit resulting from the corresponding addition.

Syntax

SICPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 0001 1101 0MMM +AAA AAAA – ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Example

```
SICPCS SPLANE + 9 (M4) (+)  ! EACH BIT OF STORE PLANE
                            ! SPLANE + 9 + i (WHERE i IS
                            ! THE DO LOOP STEP VALUE)
                            ! THAT CORRESPONDS TO A ONE
                            ! IN THE A PLANE IS SET TO
                            ! THE SUM OF ITSELF AND THE
                            ! CORRESPONDING BIT OF THE
                            ! C PLANE. EACH BIT OF THE
                            ! C PLANE IS SET TO THE
                            ! CARRY BIT GENERATED BY
                            ! THE CORRESPONDING ADDITION.
```

Function

The SICPQS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bit of the Q plane under activity control (see section 11.3.2). That is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bits generated by these additions are placed in the corresponding bits of the C plane. Carries are not under activity control; that is, every C plane bit receives the carry bit resulting from the corresponding addition.

Syntax

SICPQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1001 0101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Example

```
SICPQS 28 (-)  ! EACH BIT OF STORE PLANE 28 - i (WHERE i
               ! IS THE DO LOOP STEP VALUE) THAT
               ! CORRESPONDING TO A ONE BIT IN THE A
               ! PLANE IS SET TO THE SUM OF ITSELF AND
               ! THE CORRESPONDING BIT OF THE Q PLANE.
               ! EACH BIT OF THE C PLANE IS SET TO THE
               ! CARRY BIT GENERATED BY THE
               ! CORRESPONDING ADDITIONS.
```

SICQPCQS

Function

The SICQPCQS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bits of the Q and C planes. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not written to under activity control; that is, each Q plane bit receives the sum bit resulting from the corresponding addition.

The carry bits generated by these additions are placed in the corresponding bits of the C plane. Carries are not under activity control; that is, every C plane bit receives the carry bit resulting from the corresponding addition.

Syntax

SICQPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1011 1101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Function

The SICQPCS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bit of the C plane. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not under activity control; that is, each Q plane bit receives the sum bit resulting from the corresponding addition.

The carry bits generated by these additions are placed in the corresponding bits of the C plane. Carries are not under activity control; that is, every C plane bit receives the carry bit resulting from the corresponding addition.

Syntax

SICQPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 0011 1101 0MMM + AAA AAAA – ... .

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

SICQPQS

Function

The SICQPQS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bit of the Q plane. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not written to under activity control; that is, each Q plane bit receives the sum bit resulting from the corresponding addition.

The carry bits generated by these additions are placed in the corresponding bits of the C plane. Carries are not performed under activity control; that is, every C plane bit receives the carry bit resulting from the corresponding addition.

Syntax

SICQPQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1011 0101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 to the program code limit.

Example

```
SICQPQS SPLANE (M6)  ! EACH BIT OF SPLANE CORRESPONDING
                     ! TO AN A PLANE ONE BIT IS SET TO
                     ! THE SUM OF ITSELF AND THE
                     ! CORRESPONDING BIT OF THE Q
                     ! PLANE. EVERY BIT OF THE Q PLANE
                     ! IS SET TO THE SUM OF ITSELF AND
                     ! THE CORRESPONDING BIT OF SPLANE.
                     ! THE CARRY BITS ARE PLACE IN
                     ! IN THE C PLANE.
```

Function

The SIF instruction sets every bit of a specified store plane to zero under activity control (see section 11.3.2); that is, only store plane bits corresponding to one bits in the A plane are set to zero; store plane bits corresponding to zero A plane bits are unchanged.

Syntax

SIF ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1011 0000 0101 .MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value of the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIF SPLANE + 3 (M2)  !  EACH BIT OF PLANE SPLANE + 3 THAT
                     !  CORRESPONDS TO A ONE BIT IN THE A
                     !  PLANE IS SET TO ZERO.
```

SIPCQS

Function

The SIPCQS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bits of the C and Q planes under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bits resulting from these additions are discarded.

Syntax

SIPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1000 1101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIPCQS SPLANE (M3)  ! EACH BIT OF SPLANE THAT
                    ! CORRESPONDS TO A ONE BIT IN THE A
                    ! PLANE IS SET TO THE SUM OF ITSELF
                    ! AND THE CORRESPONDING BITS OF THE
                    ! C AND Q PLANES.
```

Function

The SIPCS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bit of the C plane under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bits resulting from these additions are discarded.

Syntax

SIPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 0000 1101 0MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIPCS SPLANE + 1 (M4)  ! EACH BIT OF SPLANE +1 THAT
                       ! CORRESPONDS TO A ONE BIT IN THE
                       ! A PLANE IS SET TO THE SUM OF
                       ! ITSELF AND THE CORRESPONDING BIT
                       ! OF THE C PLANE.
```

SIPQS

Function

The SIPQS instruction sets each bit of a specified store plane to the sum of itself and the corresponding bit of the Q plane under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

The carry bits resulting from these additions are discarded.

Syntax

SIPQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1000 0101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIPQS SPLANE (M4) (+)   ! EACH BIT OF SPALNE +i (WHERE i
                        ! IS THE CORRECT DO LOOP STEP
                        ! VALUE) THAT CORRESPONDS TO A
                        ! ONE BIT IN THE A PLANE IS SET
                        ! TO THE SUM OF ITSELF AND THE
                        ! CORRESPONDING BIT OF THE Q
                        ! PLANE
```

SIQ

Function

The SIQ instruction sets each bit of a specified store plane to the corresponding bit of the Q plane under activity control (see section 11.3.2), that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged.

Syntax

SIQ ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1011 1000 0101 .MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIQ SPLANE (M4)  !  EACH BIT OF SPLANE THAT CORRESPONDS
                 !  TO A ONE BIT IN THE A PLANE IS SET
                 !  TO THE CORRESPONDING BIT OF THE Q
                 !  PLANE.
```

SIQPCQS

Function

The SIQPCQS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bits of the C and Q planes. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not written to under activity control; that is, each Q plane bit receives the sum bit resulting from the corresponding addition.

The carry bits resulting from these additions are discarded.

Syntax

SIQPCQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1010 1101 0MMM +AAA AAAA −... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane address in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, is specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIQPCQS SPLANE +5 (M7)  !! EACH BIT OF SPLANES +5 IS SET
                        !  TO THE SUM OF ITSELF, THE C
                        !  AND Q PLANES.
```

SIQPCS

Function

The SIQPCS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bit of the C plane. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not written to under activity control; that is, each bit of the Q plane receives the sum bit resulting from the corresponding addition.

The carry bits resulting from these additions are discarded.

Syntax

SIQPCS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 0010 1101 0MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIQPCS SPLANE (M4)  !  EACH BIT OF SPLANE IS SET TO
                    !  THE SUM OF ITSELF AND THE C
                    !  PLANE, WHERE THE A PLANE IS
                    !  TRUE, THE SUMS ARE WRITTEN TO
                    !  EVERY BIT OF THE Q PLANE'
```

SIQPQS

Function

The SIQPQS instruction sets each bit of both a specified store plane and the Q plane to the sum of the store plane bit and the corresponding bit of the Q plane. The bits of the store plane are written to under activity control (see section 11.3.2); that is, the store plane is only written to for bits corresponding to one bits in the A plane; store plane bits corresponding to zero A plane bits are unchanged. The bits of the Q plane are not written to under activity control.

The carry bits resulting from these additions are discarded.

Syntax

SIQPQS ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1010 1010 0101 0MMM + AAA AAAA − ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction

4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SIQPQS 12  ! THE Q PLANE AND PLANE 12 ARE ADDEP; THE
           ! SUM BITS ARE WRITTEN TO PLANE 12 WHERE
           ! THE A PLANE IS TRUE AND ARE WRITTEN TO
           ! EVERY BIT OF THE Q PLANE.
```

Function

The SKIP instruction skips the next instruction if a specified MCU register bit has a certain value. If the SKIP instruction is the last instruction in a DO loop, the instruction that is skipped is therefore the first instruction in the DO loop, except when on the last pass of the DO loop, in which case the SKIP instruction has no effect. Skipping an instruction in a DO loop does not inhibit the stepping of addresses in that instruction and any errors that this may cause.

See also the SKIP ALL and SKIP ANY instructions.

Syntax

SKIP ⟨MCU register⟩. ⟨bit number⟩⟨modifier⟩?⟨step⟩?⟨truth value⟩

where

⟨MCU register⟩, ⟨bit number⟩, ⟨modifier⟩, and ⟨step⟩ together form an MCU register bit address (see section 11.2.6).

⟨truth value⟩ ::= 0|1|T|F

The truth values 0 and F are equivalent, as are the values 1 and T.

Binary instruction format

1111 0010 .RRR NMMM +... .... −III IIII

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bit is addressed by this instruction
2 *bit-number* specifies the value in the INT field (I) of the instruction, which is used to construct the effective INT value
3 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4 *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the INT field of the instruction
5 The bit number to be addressed is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified
6 *truth-value* specifies the inverse of the value in the N field of the instruction. An instruction will be skipped if the selected MCU register bit is the inverse of the bit in the N field

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective INT value outside the range zero to 63.

Example

```
SKIP M4.12 F     ! THE NEXT INSTRUCTION IS SKIPPED IF BIT
                 ! 12 OF M4 IS ZERO.
```

SKIP ALL

Function

The SKIP ALL instruction skips the next instruction if every bit of a specified MCU register has a certain value. If the SKIP ALL instruction is the last instruction in a DO loop, the instruction that is skipped is therefore the first instruction in the DO loop, except when on the last pass of the DO loop, in which case the SKIP ALL instruction has no effect. Skipping an instruction in a DO loop does not inhibit the stepping of addresses in that instruction, or the errors that may be so caused.

See also the SKIP and SKIP ANY instructions.

Syntax

SKIP ⟨MCU register⟩ ALL ⟨truth value⟩

where

⟨MCU register⟩ ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨truth value⟩ ::= 0 | 1 | F | T

The truth values 0 and F are equivalent, as are the values 1 and T

Binary instruction format

1111 0000 .RRR N... ....

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bits are to be tested
2. *truth-value* specifies the inverse of the value in the N field of the instruction. The next instruction is skipped if every bit of the specified MCU register is equal to the inverse of the N field

Possible run-time program errors

None.

Example

```
SKIP M1 ALL 0  !  SKIP THE NEXT INSTRUCTION IF EVERY BIT
               !  OF M1 IS 0.
```

SKIP ANY

Function

The SKIP ANY instruction skips the next instruction if any bit of a specified MCU register has a certain value. If the SKIP ANY instruction is the last instruction in a DO loop, the instruction that is skipped is therefore the first instruction in the DO loop, except when on the last pass of the DO loop, in which case the SKIP ANY instruction has no effect. Skipping an instruction in a DO loop does not inhibit the stepping of addresses in that instruction, or the errors that may be so caused.

See also the SKIP and SKIP ALL instructions.

Syntax

SKIP ⟨MCU register⟩ ANY ⟨truth value⟩

where

⟨MCU register⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨truth value⟩::= 0 | 1 | F | T

The truth values 0 and F are equivalent, as are the values 1 and T.

Binary instruction format

1111 0001 .RRR N... ....

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register whose bits are to be tested
2. *truth-value* specifies the inverse of the value in the N field of the instruction. The next instruction will be skipped if any bits of the specified MCU register are equal to the inverse of the N field

Possible run-time program errors

None.

Example

```
SKIP M4 ANY T  !  SKIPS THE NEXT INSTRUCTION IF ANY BIT
               !  OF M4 IS 1
```

SQ

Function

The SQ instruction sets each bit of a specified store plane to the corresponding bit of the Q plane.

Syntax

SQ ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

1000 1000 .101 .MMM +AAA AAAA −... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SQ 22 (M6) (+)  ! EACH BIT OF STORE PLANE 22 + (ADDR FIELD
                ! OF M6) + i (WHERE i IS THE THE DO LOOP step
                ! VALUE) IS SET TO THE CORRESPONDING BIT
                ! OF THE Q PLANE.
```

Function

The SQ_AQ instruction is a compound instruction (see section 11.3.1), and has the following effects:

1 Each bit of a specified store plane is set to the corresponding bit of the Q plane
2 Each bit of the A plane is set to the corresponding bit of the Q plane

Syntax

SQ_AQ 〈plane〉〈modifier〉?〈step〉?

where 〈plane〉, 〈modifier〉, and 〈step〉 together form a store plane address (see section 11.2.3).

Binary instruction format

1000 1100 0101 0MMM + AAA AAAA – ... .

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SQ_AQ 0(M6)  !  EACH BIT OF BOTH STORE PLANE SPLANE AND
             !  THE A PLANE IS SET TO THE CORRESPONDING
             !  BIT OF THE Q PLANE
```

SQ_QC
SQ_QCN

Function

The SQ_QC instruction is a compound instruction (see section 11.3.1) and has the following effects:

1. Each bit of a specified store plane is set to the corresponding bit of the Q plane
2. Each bit of the Q plane is then set to the corresponding bit of the C plane

The SQ_QCN instruction is a compound instruction with the following effects:

1. Each bit of a specified store plane is set to the corresponding bit of the Q plane
2. Each bit of the Q plane is then set to the inverse of the corresponding bit of the C plane

Syntax

SQ_QC ⟨plane⟩⟨modifier⟩?⟨step⟩?
SQ_QCN ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

SQ_QC : 1000 1010 1101 0MMM +AAA AAAA –... ....
SQ_QCN: 1000 1010 1101 1MMM +AAA AAAA –... ....

Notes:

1. *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4. The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SQ_QC SPLANE (M2) (–)  ! EACH BIT OF STORE PLANE SPLANE
                       ! –i (WHERE i IS THE
                       ! DO LOOP STEP
                       ! VALUE) IS SET TO THE
                       ! CORRESPONDING BIT
                       ! OF THE Q
                       ! PLANE. EACH BIT OF THE Q PLANE
                       ! IS THEN SET
                       ! TO THE
                       ! CORRESPONDING BIT OF THE C
                       ! PLANE
```

Function

The SQ_QF instruction is a compound instruction (see section 11.3.1), and has the following effects:

1 Each bit of a specified store plane is set to the corresponding bit of the Q plane
2 Every bit of the Q plane is then set to zero

The SQ_QT instruction is a compound instruction, and has the following effects:

1 Each bit of a specified store plane is set to the corresponding bit of the Q plane
2 Every bit of the Q plane is then set to one

Syntax

SQ_QF ⟨plane⟩⟨modifier⟩?⟨step⟩?
SQ_QT ⟨plane⟩⟨modifier⟩?⟨step⟩?

where ⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

SQ_QF: 1000 1010 0101 0MMM +AAA AAAA –... ....
SQ_QT: 1000 1010 0101 1MMM +AAA AAAA –... ....

Notes:

1 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used if any. If *modifier* is omitted, the MOD field is set to zero
3 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
4 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

SR
SRN

Function

The SR instruction sets each bit of a specified store plane to the corresponding bit of the R plane (see section 11.3.3).

The SRN instruction sets each bit of a specified store plane to the inverse of the corresponding bit of the R plane.

Syntax

SR ⟨MCU register⟩⟨plane⟩⟨modifier⟩?⟨step⟩?
SRN ⟨MCU register⟩⟨plane⟩⟨modifier⟩?⟨step⟩?

where

⟨MCU register⟩::=M0|M1|M2|M3|M4|M5|M6|M7

⟨plane⟩, ⟨modifier⟩, and ⟨step⟩ together form a store plane address (see section 11.2.3).

Binary instruction format

SR : 0110 .RRR .101 0MMM +AAA AAAA –... ....
SRN: 0110 .RRR .101 1MMM +AAA AAAA –... ....

Notes:

1 *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register that is used to form the R plane
2 *plane* specifies the value in the ADDR field (A) of the instruction, which is used to construct the effective ADDR value (see section 11.2.3)
3 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4 *step* specifies the value in the INCREMENT/DECREMENT field (+/–) of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop. If *step* is specified, the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction
5 The store plane addressed in this instruction is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
SR M6 SPLANE (M2)  !  EACH ROW OF STORE PLANE SPLANE IS
                   !  SET TO THE CONTENTS OF M6.
```

Function

The STOP instruction causes the DAP to stop, interrupting the host processor. The instruction is normally used to indicate error or exception conditions.

Syntax

STOP ⟨type⟩?⟨error number⟩

where

⟨type⟩ ::= TYPE ⟨basic integer⟩

⟨error number⟩::=⟨number⟩

Binary instruction format

1111 0111 YYYY|YYYY YYYY YYYY YYYY YYYY

Notes:

1. *type* specifies the first hexadecimal digit in the Y field (bits 8 to 11), and determines the type of interrupt generated, as follows:

Hexadecimal digit	*Type of interrupt*
0	User
1 to 9	ICL superstructure
A	Fixed code (ROM)
B to D	Reserved
E to F	ICL test engineering software

   If *type* is omitted, TYPE 0 is assumed

2. *error-number* specifies the value in the rest of the Y field. For user error stops (TYPE 0), *error-number* must be in the range 1 to 9999; for other error types it must be less than or equal to hexadecimal £FFFFF
3. The STOP instruction causes a user defined APAL error. To generate a non-error exit, the EXIT instruction should be used. An EXIT in the top level DAP subroutine will cause return to the calling host program
4. If run-time parameters indicate that execution is to be restarted after a user defined error, execution will restart at the instruction following the STOP instruction

Possible run-time program errors

None.

Example

```
STOP 2000  !  CAUSES A USER DEFINED APAL ERROR WITH
           !  ERROR NUMBER 2000.
```

SUB

Function

The SUB instruction subtracts the contents of one MCU register from another, leaving the result in the latter register. Arithmetic overflow is not detected.

Syntax

SUB ⟨MCU register-1⟩⟨MCU register-2⟩

where

⟨MCU register-1⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

⟨MCU register-2⟩::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7

Binary instruction format

1110 1100 1RRR .MMM 00... ....

Notes:

1. *MCU-register-1* specifies the value in the MCUR field (R) of the instruction, which specifies both the MCU register containing the first operand and the register that will contain the result
2. *MCU-register-2* specifies the value in the MOD field (M) of the instruction, which specifies the MCU register containing the second operand
3. Each operand is treated as a 64-bit unsigned integer. Any carry out of the most significant bit position is ignored, therefore arithmetic overflow is not detected

Possible run-time program errors

None.

Example

SUB M3 M4 ! M3 = M3 – M4.

XAN

Function

The XAN instruction sets a specified store row to the inverse of the corresponding row of the A plane.

Syntax

XAN ⟨row⟩⟨modifier⟩?⟨step A⟩?

where ⟨row⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store row address (see section 11.2.3).

Binary instruction format

1001 .000 .11S .MMM + AAA AAAA − III IIII

Notes:

1 *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero, and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)

4 The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

5 The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XAN SROW + 12.9 (M4)   !  ROW 9 +  (INT FIELD OF M4) OF
                       !  STORE PLANE SROW + 12 IS SET TO
                       !  THE INVERSE OF ROW 9 + (INT
                       !  FIELD OF M4) OF THE A PLANE.
```

XF

Function

The XF instruction sets every bit in a specified store row to zero.

Syntax

XF ⟨row⟩⟨modifier⟩?⟨step A⟩?

where ⟨row⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store row address (see section 11.2.3).

Binary instruction format

0100 .000 .11S .MMM +AAA AAAA −III IIII

Notes:

1. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)
4. The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
5. The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

XF 18.4 ! ROW 4 OF STORE PLANE 18 IS SET TO ALL ZEROS.

Function

The XIC instruction sets a specified store row to the corresponding row of the C plane under activity control; that is, only the values of store row bits corresponding to one bits in the corresponding row of the A plane are changed; store row bits corresponding to zero A plane bits are unchanged.

Syntax

XIC <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 11.2.3).

Binary instruction format

1011 0000 111S .MMM +AAA AAAA −III IIII

Notes:

1. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero, and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)
4. The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
5. The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XIC SPLANE + 2.12 (M4)  !  EACH BIT OF ROW 12 + (INT FIELD
                        !  OF M4) IN STORE PLANE SPLANE
                        !  +2 IS SET TO THE CORRESPONDING
                        !  BIT OF ROW 12 + (INT FIELD OF
                        !  M4) OF THE C PLANE, PROVIDED
                        !  THE CORRESPONDING BIT OF THE
                        !  A PLANE IS ONE.
```

XIF

Function

The XIF instruction sets each bit in a specified store row to zero under activity control (see section 11.3.2) that is, only the values of store row bits corresponding to one bits in the corresponding row of the A plane are changed; store row bits corresponding to zero A plane bits are unchanged.

Syntax

XIF <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 11.2.3).

Binary instruction format

1011 0000 011S .MMM +AAA AAAA −III IIII

Notes:

1. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
2. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
3. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)
4. The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
5. The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XIF 21.10  ! ROW 10 OF STORE PLANE 21 IS SET TO ZERO
           ! WHEREVER THE CORRESPONDING BIT
           ! OF THE A PLANE IS ONE.
```

Function

The XIQ instruction sets a specified store row to the contents of the corresponding row of the Q plane under activity control (see section 11.3.2); that is, only the values of store row bits corresponding to one bits in the corresponding row of the A plane are changed; store row bits corresponding to zero A plane bits are unchanged.

Syntax

XIQ <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 11.2.3).

Binary instruction format

1011 1000 011S .MMM +AAA AAAA −III IIII

Notes:

1 *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero, and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)

4 The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

5 The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XIQ SPLANE + 10.14 (M2) (−A)  ! ROW 14 + (INT FIELD OF M2)
                              ! OF STORE PLANE SPLANE
                              ! + 10 − i (WHERE i IS THE
                              ! DO LOOP STEP VALUE) IS
                              ! SET TO THE CORRESPONDING
                              ! ROW OF THE Q PLANE
                              ! WHEREVER THE CORRESPONDING
                              ! BITS OF THE A PLANE ARE
                              ! ONES.
```

XQ

Function

The XQ instruction sets a specified store row to the corresponding row of the Q plane.

Syntax

XQ ⟨row⟩⟨modifier⟩?⟨step A⟩?

where ⟨row⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store row address (see section 11.2.3).

Binary instruction format

1000 1000 .11S .MMM +AAA AAAA -III IIII

Notes:

1 *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)

2 *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero

3 *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

If *step A* is + or −, the SELECT field is set to zero, and the current DO loop value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)

4 The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified

5 The row part of the required row address is given by the effective INT value, which is the sum of the INT field of the instruction and the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XQ 12.13 (M2) (+)  !  ROW 13 + (INT FIELD OF M2) + i
                   !  (WHERE i IS THE DO LOOP STEP VALUE)
                   !  OF STORE PLANE 12 + (ADDR FIELD OF
                   !  M2) IS SET TO THE CORRESPONDING
                   !  ROW OF THE Q PLANE.
```

XR
XRN

Function

The XR instruction sets a specified store row to the contents of a specified MCU register.

The XRN instruction sets a specified store row to the inverse of the contents of a specified MCU register.

Syntax

XR ⟨MCU register⟩⟨row⟩⟨modifier⟩?⟨step A⟩?
XRN ⟨MCU register⟩⟨row⟩⟨modifier⟩?⟨step A⟩?

where

⟨MCU register⟩ ::= M0|M1|M2|M3|M4|M5|M6|M7

⟨row⟩, ⟨modifier⟩, and ⟨step A⟩ together form a store row address (see section 11.2.3).

Binary instruction format

XR : 0110 .RRR .11S 0MMM +AAA AAAA −III IIII
XRN: 0110 .RRR .11S 1MMM +AAA AAAA −III IIII

Notes:

1. *MCU-register* specifies the value in the MCUR field (R) of the instruction, which specifies the MCU register used in the instruction
2. *row* specifies the values in the ADDR (A) and INT (I) fields of the instruction, which are used to construct the effective ADDR and INT values (see section 11.2.3)
3. *modifier* specifies the value in the MOD field (M) of the instruction, which specifies the modifier register to be used, if any. If *modifier* is omitted, the MOD field is set to zero
4. *step A* specifies the value in the INCREMENT/DECREMENT field (+/−) of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the SELECT field (S) of the instruction, which determines whether the INT or ADDR part of the address is to be stepped.

   If *step A* is + or −, the SELECT field is set to zero and the current DO loop step value is effectively added to or subtracted from the INT field of the instruction (that is, the address is stepped by a single row).

   If *step A* is +A or −A, the SELECT field is set to one and the current DO loop step value is effectively added to or subtracted from the ADDR field of the instruction (that is, the address is stepped by a single plane)
5. The plane part of the required row address is given by the effective ADDR value, which is the sum of the ADDR field of the instruction and the ADDR field of *modifier*, if specified
6. The row part of the request row address is given by the effective INT value, which is the sum of the INT field of *modifier*, if specified

Possible run-time program errors

A run-time program error will occur if an attempt is made to modify or step the effective ADDR value outside the DAP program block or into the read only area from plane 36 up to the value of the program code limit.

Example

```
XR M4 28.10 (-)  !  ROW 10 - i (WHERE i IS THE DO LOOP
                 !  STEP VALUE) OF STORE PLANE 28 IS
                 !  SET TO THE CONTENTS OF M4.
```

Computer Physics Communications 37 (1985) 351–356
North-Holland, Amsterdam

A VERY HIGH SPEED MONTE CARLO SIMULATION ON DAP

S.F. REDDAWAY

ICL, Stevenage, Hertfordshire, UK

D.M. SCOTT

Dept. of Physics, Edinburgh University, Edinburgh, UK

and

K.A. SMITH

DAP Support Unit, Queen Mary College, University of London, Mile End Road, London E1 4NS, UK

The Ising ferromagnet is a well-known model of a system of interacting spins which is amenable to Monte Carlo calculation involving:
(a) Boolean work and low precision counting;
(b) conditional choice of transition probabilities;
(c) high quality 24-bit random number generation (RNG);
(d) comparison of transition probabilities with random numbers.
An assembler implementation on the 64×64 DAP has achieved 42×10^6 spin updates per second on a $64 \times 64 \times 64$ problem using a standard DAP random number generator (RNG) that has recently been speeded up by a factor of eight. A later implementation involves:
(a) a new RNG that is an order of magnitude faster still, mainly achieved by producing data in greater bulk;
(b) a rewrite of the code including a form of parallel table look up for the transition probabilities;
(c) an increase in the problem size to $128 \times 128 \times 144$.
The performance is 218×10^6 spin updates per second. This compares with 22×10^6 on a CYBER 205 and 24×10^6 on special purpose hardware built by a group in Santa Barbara, both on $64 \times 64 \times 64$ problems.

Our results can be extrapolated to about 40×10^6 spin updates per second for a possible 32×32 DAP which would be about two times faster than a CYBER 205 and two orders of magnitude cheaper.

1. Introduction

Much effort has been put into developing efficient algorithms for the Monte Carlo simulation of the Ising model. This model is of some interest to theoretical physicists, and – for better or worse – computer simulation of this system has become one of the yardsticks by which the suitability of different computers (and more generally computer architectures) for scientific calculations is measured.

The first work was performed by Fosdick who introduced multi-spin coding techniques [1]. Since then multi-spin coding techniques have been used on serial machines [2,3], and on a vector machine [4]. Also code has been written for the DAP [5], and some workers have gone so far as to construct special purpose processors [6,7]. It seems worthwhile, therefore, to examine what sort of performance can be achieved by special coding on existing processors.

Two pieces of code have been developed for the 64×64 DAP. One (which is already in use in applications) is for a $64 \times 64 \times 64$ lattice and uses the random number generator GO5FASTINT2 (a speeded up version of GO5XORINT2). The other is for a $128 \times 128 \times 144$ lattice (which is easily adaptable to a $128 \times 128 \times 128$ lattice with a small

0010-4655/85/$03.30 © Elsevier Science Publishers B.V.
(North-Holland Physics Publishing Division)

Table 1

Machine	Update rate [a]	Ref.
CDC Cyber 76	0.94	[3]
CDC Cyber 205	22	[12]
ICL DAP [b]	6	[5]
SBIMP [c]	25	[6]
DISP [d]	1.5	[7]
ICL DAP [e]	218	

[a] In millions of spin updates per second.
[b] Coded in DAP Fortran with old RNG.
[c] Santa Barbara Ising Model Processor.
[d] Delft Ising System Processor.
[e] Coded in APAL and with fast RNG.

drop in performance) and is substantially faster, mainly due to a faster random number generator. The larger lattice also leads to a simpler mapping of the problem onto store. They are written in the DAP assembly language APAL [8].

Only the generation of configurations is considered – measurements are not included. The generation of new configurations is an important part of the problem as a configuration may have to be updated many times to produce a configuration which is sufficiently different (that is more or less independent, in the statistical sense, of its predecessor) to justify measurements being made on it.

The increase in performance obtained is such that the DAP is very much faster, for this algorithm, than any other machine for which timings have been published – including the special purpose machines (see table 1).

2. The DAP

A description of the first generation DAP may be found in ref. [9]. The features which are important for our purposes will now be described briefly.

The DAP differs from a conventional serial processor in that it can perform simultaneously the same operation on many items in DAP store. This parallel processing capability is provided by a 64×64 matrix of processors (processing elements or PEs) each of which may operate on its own local store which consists of 4096 bits (512 bytes). (There is a machine with four times as much store at Queen Mary College.) The DAP store is the aggregate of the local store of all 4096 PEs.

Hard wired connections exist between each PE and its nearest neighbours. A PE at an edge can be instructed to treat the corresponding PE on the opposite edge as a nearest neighbour, the connection once again being provided by the hardware, so that cyclic boundary conditions result.

The PEs are one-bit processors. Interpretation and processing of data as a specific type (integer, real, etc.) is entirely a function of the software and any length of representation, within wide constraints, can be chosen by the user.

3. The model and the Metropolis algorithm

The Ising model on a simple cubic lattice has a "spin" variable, $\sigma(i)$, associated with every point i of a simple cubic lattice. The spin variables can take on the values $+1$ or -1 (the spin is said to be, respectively, "up" or "down").

The energy associated with a given configuration of spins $\{\sigma\}$ is defined to be

$$E(\{\sigma\}) = -J\sum_{i}\sum_{\mu=1}^{3}\sigma(i)\sigma(i+e_\mu),$$

where e_μ is a unit vector pointing in the $+\mu$ direction and J is a coupling constant. The energy density is local in the sense that it depends only on the products of spins which are nearest neighbours. The probability of a given configuration of spins $\{\sigma\}$ occurring is

$$P(\{\sigma\}) = \frac{\exp[-E(\{\sigma\})/kT]}{\sum_{\{\sigma'\}}\exp[-E(\{\sigma'\})/kT]},$$

where the sum is over all possible spin configurations – that is over all possible assignments of the values plus or minus one to the spin variables. T is the temperature, and k is Boltzmann's constant.

The average value of a function f of the spins is defined by

$$\langle f \rangle = \sum_{\{\sigma\}} f(\{\sigma\})P(\{\sigma\}),$$

and it is such averages which are of interest. In a

Monte Carlo simulation a sequence of spin configurations is generated in such a fashion that, in the limit of an infinite sequence, the relative frequency of occurrence of a configuration $\{\sigma\}$ is $P(\{\sigma\})$. In practice only a finite number, N, of configurations is generated and $\langle f \rangle$ is approximated by

$$\frac{1}{N}\sum_{i=1}^{N} f(\{\sigma\}).$$

The method used in the code described here to generate the sequence is an adaptation of the scheme described by Metropolis et al. [10]. Local changes are made to one configuration to obtain a subsequent configuration. A spin is selected and one considers flipping it (i.e. if the spin is originally "up" it is set "down" and vice versa). The change in energy associated with the flip, ΔE, is calculated. If ΔE is negative (i.e. if the change reduces the energy of the configuration) the new value is accepted. If ΔE is positive (that is if flipping increases the energy) then the new value is sometimes accepted and sometimes rejected. The probability of acceptance is $\exp[-\Delta E/kT]$. The order in which the spins are updated before a new configuration is considered to have been generated is largely at the discretion of the programmer. A detailed description of the method is given in ref. [1].

4. Implementation

4.1. Introduction

This paper concentrates on the $128 \times 128 \times 144$ implementation.

In principle any set of spins which do not interact directly may be updated simultaneously. In our implementation the lattice is divided up into $2 \times 2 \times 2$ cubes which contain eight points each. A set of points which may be updated simultaneously is obtained by selecting one spin from each small cube; the points being in corresponding positions in their respective cubes. Eight sets are obtained in this way. Each type of point is stored and processed as a $64 \times 64 \times 72$ logical array; in DAP terms a set of 72 logical matrices (logical variables in the DAP occupy one bit of store each).

There are three code sections, each comprising an APAL DO loop:

(a) Random number generation (RNG). 254 planes of random bits are generated in one loop, from which 18 matrices are derived. Some low significance bits are used twice in order to create 24-bit random numbers. The generator is described in ref. [11], and is expected to produce high quality random numbers.

(b) Neighbour spin summation. One loop deals with a set of 72 matrices. Cyclic boundary conditions are provided 'free' by hardware in two dimensions, and by simple code in the third. There are slightly different pieces of code for each sort of point.

(c) Table look up of transition probabilities and comparison of them with random numbers. One loop deals with 18 spin matrices, consuming the random numbers generated in (a).

4.2. Neighbour summation

This code counts how many of the 6 neighbours have the same spin value as the point under consideration. The 3-bit result matrices take 21 machine cycles each, and 2 further cycles complete the "super-decode" needed for the advanced table look up (see section 4.3.2).

4.3. The (effective) matrix of transition probabilities

The transition probabilities are treated as 24-bit unsigned integers to which 24-bit unsigned random numbers are added.

In this section we shall consider only neighbour counts of 4, 5 and 6; we shall return to the other cases in section 4.4.

The correct transition probability must be used at each lattice point corresponding to neighbour counts of 4, 5 or 6 aligned spins. Straightforward methods can be used, such as conditional copying of the 3 scalars to matrices; however, we use a faster method.

Suppose that we had constructed a matrix of transition probabilities. Its bit planes would be fetched one at a time for the compare. The crucial

observation is that an examination of them as they were fetched would reveal only *eight* distinct bit patterns and *not* twenty-four, although what patterns we would see would depend on the count values. This is because there are only three transition probabilities so that at each bit significance there are only $8(=2^3)$ possibilities. Consequently, rather than constructing the 24 planes of the matrix of transition probabilities and then fetching them one by one, the amount of work that is done can be reduced by first of all constructing the eight possible bit patterns and then selecting the appropriate one for each bit-significance.

4.3.1. Construction of the bit patterns

Let the transition probabilities for the neighbour counts of 4, 5, and 6 be $P4$, $P5$ and $P6$ respectively.

If the nth bits of $P4$, $P5$ and $P6$ are all 0 a plane of zeros must be fetched to represent the nth bit-plane of the matrix of transition probabilities. Such a plane is set once and for all.

If the nth bits of $P4$, $P5$ and $P6$ are 0, 1 and 0, respectively, a plane must be fetched which is zero where the neighbour count is 4, one where the count is 5, and zero where the count is 6. But this is precisely the pattern in the least significant bit-plane of the neighbour count and so one may simply fetch that.

If the nth bits of $P4$, $P5$ and $P6$ are 0, 0 and 1, respectively, a plane must be fetched which has its zero where the count is 4 or 5, and is one where the count is 6. In this case one may fetch the middle bit of the neighbour count.

An exclusive OR of the two least significant bit-planes of the neighbour count gives a pattern corresponding to the bit-values 0, 1 and 1 for $P4$, $P5$ and $P6$, respectively.

Neighbour sum	Binary representation			XOR of 2 LS bits
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1

The other four patterns are obtained by inverting those above. The complete set is given in section 4.3.2.1.

4.3.2. Advanced table look up

What has yet to be explained is the way in which the correct pattern is selected. Two methods will be described. The first is suitable for use when the transition probabilities are specified at run time, and for it to work the patterns must be stored contiguously. The second may only be used when the transition probabilities are known at compile time, and then there is no restriction on where the patterns are stored. The latter method is faster at run time.

4.3.2.1. First method: table-as-data. The three table elements are put into an otherwise zero DAP vector as entries 57, 56 and 55 (labelling from 0 to 63). The bits of a single entry are stored in a row (East–West). Now if a "column" of bits (North–South) is extracted from the vector then it contains the three bits of a given bit-significance from the transition probabilities, all other bits being zero. The location of these three bits is such that they give a 3-bit number which can be directly used as a bit-plane offset address (the 64-bit number gives the offset in rows) to select the correct bit-plane for that bit-significance in the table look up. For example rows, 57, 56 and 55 in the table bit plane might look like:

```
← etc.  0  1  0 | 1 | 1  1  0  1  0
← etc.  0  0  0 | 1 | 0  1  1  1  0
← etc.  0  0  0 | 0 | 0  1  1  1  1
                  ↓
                011 ≡ 3
```

So at the marked level of bit-significance a plane offset of 3 would be used.
The correspondence between the offset and the contents of the bit-planes is as follows:

Plane (offset)	Plane entries for PEs with spin counts of: 4	5	6
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

Planes 0 and 7 are permanently false and true, respectively, and may be set once and for all. The other six are calculated just once each time a DAP plane of spins is updated. Only the two least significant bits of the neighbour sum are used in the calculation. The above conversion of a 2-bit (in a sense only "1.5-bit") index to 8 bits is an example of what has been termed a "super-decode". (A full 2-bit index super-decodes to 16 bits.) The above super-decode takes 7 cycles.

The table look up code itself consists of fetching a "column" from the table vector which is used to fetch the correct super-decode plane.

4.3.2.2. Second method: table-as-code. This method can only be used if the table values are known at compile time as the table data are built into the instructions that fetch the super-decode planes. Not only does this save the fetching of the table vector "columns" at every bit-significance, but the super-decode is much simpler. Four of the above eight super-decode planes are exact inverses of the other four. By having the option of inverting a fetched plane, only four planes are needed. One of these is fixed, and of the data-dependent planes two are just the index bits; calculation of the remaining plane constitutes the only super-decode work needed, and takes only 2 cycles.

This method has been made practical by a code generator which when supplied with a value for the coupling constant, J, produces optimised code for the table look up.

4.4. Table look up and random number compare loop

With the table-as-data method an APAL DO loop works along the bit-significance with code (costing 3.5 cycles/bit) such as:

```
DO 24 TIMES
RY M7 TABLE + 0.63(−)   !Fetch address of
                          next "super-
                          decode" plane
QS RANDS + 23(M4 −)     !Fetch next random
                          bit
CPCQSN 0(M7)            !Compare "super-
                          decode" plane
LOOP
```

As APAL DO loops cannot be nested the looping through spin planes must be explicitly coded, (see ref. [8] for description of APAL instructions).

With the table-as-code method, the equivalent of the above loop is written out, and the whole 24-bit compare included in an APAL DO loop which loops through spin planes. For each bit of the compare there are two instructions, a QS (or QSN) to fetch (and invert) the appropriate super-decode plane and a CPCQS to fetch and compare the random bit. However, in some cases an instruction may be omitted. For instance

```
QS 0            !Fetch the all zero "super-
                  decode" plane
CPCQS 7(M4)     !Compare a random number
                  plane to the Q and C planes
```

may be replaced by

```
CPCS 7(M4)
```

This and other simplifications can be used in about 25% of cases and the cost thus averages 1.75 cycles/bit.

The table-as-code loop also deals with the flipping of spins when the spin count is 0-3. For counts of 4, 5 and 6 an overflow from the compare causes transition, and the most significant bit of the spin count forces transition if the count is 0, 1, 2 or 3. The loop takes typically 45 cycles per plane of spins for 24-bit precision.

4.5. Overall performance

The cost per plane of spins including overheads is approximately:

Neighbour count + super-decode	23.1 cycles
Random number generation	24.4
Table look up etc. loop	45
Control overheads	1
Total	93.5 cycles

Each cycle is 200 ns, so this corresponds to about 219×10^6 spin updates/s. The measured time is 218×10^6 spin updates/s.

4.6. The 64^3 problem

The 64^3 problem has a more complex mapping into store. If the lattice is viewed as a 3D chessboard, the black points may be updated simultaneously followed by the white points. Rather than storing the spins from one lattice plane in a single DAP plane the black and white spins are stored in different planes. Each DAP plane contains either only black spins or only white spins, and each contains spins from two lattice planes. This avoids the necessity to merge planes for processing.

The 64^3 implementation using three calls of the GO5FASTINT2 RNG to produce two 24-bit random numbers and using the table-as-data method runs at 42.6×10^6 spin updates/s. Most of the time is spent in the RNG.

4.7. Further possibilities

One idea is to use the same set of random numbers to feed several independent simulations with different transition probabilities. The only computing limitation is storage space. Performance could be improved by up to 30%

Another idea is to change the Metropolis algorithm slightly such that a constant transition probability is not used, but rather two or more sets of probabilities are used at different times. It could then be arranged that the frequently used sets are approximations to the 'true' probabilities with several fewer bits precision. The 'errors' introduced could be balanced by sometimes selecting probabilities with balancing errors of opposite sign; this might be termed "probability equalisation" and would reduce the number of bits which have to be generated in the table look up and in random number generation. There is plenty of scope for theory and experiment on this idea, which might improve performance by up to a factor of two.

5. Conclusion and discussion

The performance obtained is nearly an order of magnitude better than the best results known to the authors.

Extrapolation to a possible 32×32 DAP product gives a performance of about 40×10^6 spin updates/s which is about two times faster than the CDC Cyber 205 [12] on a machine costing about two orders of magnitude less. This also indicates that the value of special purpose hardware for the Ising model is questionable.

At first sight the application may appear to require MFLOPS because probability is "obviously" a real quantity and RNGs such as the NAG algorithm use high precision multiplication. However, this paper shows that rather than choosing a floating point machine, it is much better to use a more general and flexible bit-organised array processor.

References

[1] L.D. Fosdick, in: Methods in Computational Physics, vol. 1, eds. B. Alder et al. (Academic Press, New York, London, 1963).
[2] R. Zorn, H.J. Hermann and C. Rebbi, Comput. Phys. Commun. 23 (1981) 337.
[3] C. Kalle and V. Winklemann, J. Stat. Phys. 28 (1982) 639.
[4] S. Wansleben and J.C. Zabolitzky, Institut für Theoretische Physik der Universität zu Köln, preprint.
[5] G.S. Pawley, D.J. Wallace, R.J. Swendson and K.G. Wilson, Phys. Rev. B29 (1984) 4030.
[6] R.B. Pearson, J.L. Richardson and D. Toussaint, J. Comput. Phys. 51 (1983) 241.
[7] A. Hoogland, J. Spaa, B. Selman and A. Compagner, J. Comput. Phys. 51 (1983) 250.
[8] ICL Technical Publication TP6919 (1979).
[9] P.M. Flanders, D.J. Hunt, S.F. Reddaway and D. Parkinson, in: High Speed Computer and Algorithm Organisation eds. D. Kuck et al. (Academic Press, New York, 1977) p. 113.
[10] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, J. Chem. Phys. 21 (1953) 1087.
[11] K.A. Smith, S.F. Reddaway and D.M. Scott, Comput. Phys. Commun. 37 (1985) 239.
[12] D. Stauffer, private communication.

Computer Physics Communications 37 (1985) 239–244
North-Holland, Amsterdam

VERY HIGH PERFORMANCE PSEUDO-RANDOM NUMBER GENERATION ON DAP

K.A. SMITH

DAP Support Unit, Queen Mary College, Mile End Road, London E1 4NS, UK

S.F. REDDAWAY

ICL, Stevenage, Hertfordshire, UK

and D.M. SCOTT

Dept. of Physics, Edinburgh University, Edinburgh, UK

Since the National DAP Service began at QMC in 1980, extensive use has been made of pseudo-random numbers in Monte Carlo simulation. Matrices of uniform numbers have been produced by various generators:

(a) multiplicative ($x_{n+1} = 13^{13} x_n \bmod 2^{59}$);

(b) very long period shift register ($x^{4423} + x^{271} + 1$);

(c) multiple shorter period ($x^{127} + x^7 + 1$) shift registers generating several matrices per iteration.

The above uniform generators can also feed a normal distribution generator that uses the Box–Muller transformation.

This paper describes briefly the generators, their implementation and speed. Generator (b) has been greatly speeded-up by re-implementation, and now produces more than 100×10^6 high quality 16-bit numbers/s. Generator (c) is under development and will achieve even higher performance, mainly due to producing data in greater bulk. High quality numbers are expected, and performance will range from 400 to 800×10^6 numbers/s, depending on how the generator is used.

1. Introduction

The DAP subroutine library on the Queen Mary College DAP has over 200 routines. However, one group of routines is used far more than any others – the pseudo-random number generators (RNGs). Moreover, for a significant number of applications (lattice gauge simulations, ferromagnetic simulations, Monte Carlo simulation of liquids, optimisation), the existence of a fast, high quality RNG is essential. In this paper, we describe the development of RNGs on the DAP in a basically chronological order. The main emphasis will be on uniform RNGs; however, there is a description of a normal RNG.

2. Multiplicative congruential RNG

The first generator implemented on the DAP was a multiplicative congruential generator (see ref. [1]). This is the standard type of generator on most computer systems. The sequence of pseudo-random numbers (x_n) is defined by

$$x_{n+1} = a x_n \bmod m.$$

For the DAP we choose a and m to be the same as those used in the NAG RNG (GO5CAF) on the ICL 2980 (see ref. [2]). Thus, we used

$$x_0 = 123456789(1 + 2^{32})$$

and

$$x_{n+1} = x_n 13^{13} \bmod 2^{59}. \tag{1}$$

The next task was to map this serial algorithm onto the DAP so that 4096 members of (x_n) could be produced simultaneously. This was done by rewriting (1) to generate a sequence of vectors of length 4096, (X_n). That is:

$$X_{n+1} = X_n (13^{13})^{4096} \bmod 2^{59}. \tag{2}$$

0010-4655/85/$03.30 © Elsevier Science Publishers B.V.
(North-Holland Physics Publishing Division)

Table 1

Routine	Function	Execution time
GO5_NAG_INT_8	INTEGER * 8 matrix from sequence (X_n)	670 μs
GO5_NAG_INT_4	INTEGER * 4 matrix from sequence $(X_n/2^{28})$	660 μs
GO5_NAG_REAL_8	REAL * 8 matrix from sequence $(X_n/2^{59})$	845 μs
GO5_NAG_REAL_4	REAL * 4 matrix from sequence $(X_n/2^{59})$	783 μs

Then, if we set

$$X_0 = (x_0, x_1, \dots, x_{4095})$$

and use (2), we get

$$X_1 = (x_{4096}, x_{4097}, \dots, x_{8191})$$
$$X_2 = (x_{8192}, x_{8193}, \dots, x_{12287})$$
$$\vdots$$

At first sight (2) contains a small computational difficulty – that of calculating $(13^{13})^{4096}$! However, all we need is $(13^{13})^{4096} \bmod 2^{59}$ to produce (x_n). The implementation on the DAP was easy as the essential computation in (2) is a multiplication of a scalar variable by a matrix variable. This RNG is available to DAP users and can be assessed via the routines given in table 1.

3. Very long period shift-register RNG

The second generator implemented on the DAP was a shift-register (or XOR) generator (see refs. [1,3,4]). Although shift-register generators seem very different to multiplicative ones, they both come from the same mathematical analysis. That is, they are both special cases of the general RNG

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m \qquad (3)$$

when $k = 1$, (3) reduces to (1). When m is prime and

$$f(x) = x^k - a_1 x^{k-1} - \cdots - a_k$$

is a primitive polynomial modulo m, then (3) produces a sequence of length $m^k - 1$. XOR generators refer to the case when $m = 2$; then (3) produces a sequence of pseudo-random bits. Random integers of any length are then constructed by selecting bits from this sequence to make up the binary representation of the integer. For our implementation we choose $m = 2$ and

$$f(x) = x^{4423} + x^{271} + 1,$$

since $2^{4423} - 1$ is a Mersenne prime and the values 4423 and 271 allowed efficient use of the DAP dimensions.

This meant that we wanted to generate a sequence of bits (b_n) according to

$$b_n = b_{n-4423} \oplus b_{n-271},$$

where $\oplus$ is exclusive OR. Also we needed to do this simultaneously for 16 independent subsequences of (b_n); that is, extract consecutive bits from (b_n) at 16 very widely differing points so that the bits could be considered to be unrelated. This was necessary because poor quality random numbers are obtained by taking 16 consecutive bits from (b_n) and calling them a 16 bit integer (see ref. [1]).

Due to the bit level nature of the XOR generator we have many ways of mapping the generator onto the DAP. To achieve each increase in performance required more detailed analysis of possible mappings with the aim of minimising data movement.

The first implementation consisted of taking 4423 16 bit integers (B_n) and storing them in horizontal mode on the DAP. Each B_n contains the bits

$$b_n^{(0)} b_n^{(1)} \dots b_n^{(15)}$$

from 16 independent subsequences of (b_n). Hence the process of generating further values of (B_n) can be represented by

$$B_n = B_{n-4423} \oplus B_{n-271}.$$

Table 2

Routine	Function	Execution time
GO5_XOR_INT_2	INTEGER * 2 matrix containing 4096 consecutive values of (B_n)	348 μs
GO5_XOR_REAL_3	REAL * 3 matrix containing $(B_n/2^{16})$	500 μs
GO5_XOR_REAL_4	REAL * 4 matrix containing $(B_n/2^{16})$	520 μs
GO5_XOR_PLANE	LOGICAL matrix where the probability of any component being true equals 1/2.	56 μs

Moreover, because of the horizontal storage mode, the next 4423 values of (B_n) can be generated 256 at a time by doing bitwise exclusive ORs on two DAP planes. In fact, it was possible to generate the next 4423 values in 216 μs. Unfortunately, the numbers could not be used efficiently in horizontal mode since they could only be accessed as scalars or vectors. Thus the last step was to apply the DAP Fortran routine CONVFM2 to 4096 values of (B_n) to give the user an INTEGER * 2 matrix. The resulting generator is available to DAP users via the routines given in table 2. An analysis of the APAL (see ref. [5]) used to generate the random number gives the following breakdown.

- 63 cycles/DAP plane for computation and output to the calling routines;
- 33 cycles/DAP plane for the mode conversion.

(By cycle we mean one DAP cycle; that is, 200 ns on the 64 × 64 DAP at QMC.) Thus the total cost is 96 cycles/DAP plane for the GO5 XOR generator.

The next implementation of the generator, GO5 FAST, considerably increased the efficiency of random number generation by using a data mapping which reflected the output required. The basic structure can be seen by looking at the mapping of 4096 values of (B_n) onto 16 DAP planes:

Store planes

15	$b_{4065}^{(15)} b_{4066}^{(0)} b_{4066}^{(1)} \ldots$	$b_{4066}^{(15)} \ldots$	$b_{4320}^{(15)} b_{4065}^{(0)} \ldots$	$b_{4065}^{(14)}$
14	$b_{3794}^{(14)} b_{3794}^{(15)} b_{3795}^{(0)}$	$b_{3795}^{(14)}$	$b_{4049}^{(14)} b_{4049}^{(15)}$	$b_{3794}^{(13)}$
⋮	⋮	⋮	⋮	⋮
1	$b_{271}^{(1)} b_{271}^{(2)} b_{271}^{(3)} \ldots$	$b_{272}^{(1)} \ldots$	$b_{526}^{(1)} b_{526}^{(2)} \ldots$	$b_{271}^{(0)}$
0	$b_{0}^{(0)} b_{0}^{(1)} b_{0}^{(2)} \ldots$	$b_{1}^{(0)} \ldots$	$b_{255}^{(0)} b_{255}^{(1)} \ldots$	$b_{255}^{(15)}$

4096 PEs

Clearly, each column of 16 bits satisfies the basic requirement that each bit comes from a different subsequence of the basic (b_n) generator.

The essential APAL instructions that are required to update this generator are:

```
DO      15 TIMES
QS      PL1(M2 + )   ! PL1: store plane 1
SQ         1(M1 + )  ! output
QQ      E C 1
:SIPQS PLO(M2 + )    ! PLO : store plane 0
```

A detailed analysis of the APAL used in GO5 FAST gives the following breakdown:

- 4.5 cycles/DAP plane for computation and output to the calling routine;
- 8.25 cycles/DAP plane for the "end effects" so that all the subsequences are correctly advanced.

Thus the total cost is 12.75 cycles/DAP plane for the GO5 FAST generator.

This generator is now available to DAP users via the routines given in table 3. A further implementation, GO5 NEW, of the (B_n) generator is under development. Again a speed-up is achieved by using a more efficient data mapping.

The essential APAL instructions that are required to update this generator are:

```
DO 15 TIMES
SIQPQS      PL1(M2 + )! PL1 : store plane 1
QQ   E C 1
:SQ      1(M1 + )           ! output to user space
```

It has been estimated that GO NEW will have the following breakdown:

- 3.5 cycles/DAP plane for computation and output to the calling routine;
- 3.5 cycles/DAP plane for end effects.

Table 3

Routine	Function	Execution time
GO5_FAST_INT_2	INTEGER * 2 matrix containing 4096 consecutive values of (B_n)	44 μs
GO5_FAST_REAL_3	REAL * 3 matrix containing $(B_n/2^{16})$	185 μs
GO5_FAST_REAL_4	REAL * 4 matrix produced by calling FAST INT 2 twice	267 μs
GO5_FAST_PLANE	LOGICAL matrix where the probability of any component being true equals 1/2.	37 μs

Hence it is estimated that GO5 NEW will cost 7 cycles/DAP plane.

4. A bulk RNG with many shift registers

The above basic loop takes 2.5 cycles/plane if the output (SQ) is omitted. It is possible to omit the QQ neighbour shift instruction as well, resulting in only 1.5 cycles/plane, but only by having several bits from the same subsequence in the same PE; to maintain number quality these must not form the bits of a single number. However, several matrices of numbers can be generated in the same update, with each number composed of bits from independent subsequences.

End effects can be very greatly reduced if one "shift register" has very few (preferable only one) bits on a plane. This allows the awkward polynomial positions to be dealt with economically. The length of the 4423 polynomial means it must be spread with many bits per plane, so another generator is being developed based on the irreducible trinomial:

$$x^{127} + x^7 + 1.$$

Each '127' generator has a cycle length of $2^{127} - 1$ which appears small in comparison with the '4423' generator; however, it is still very long in comparison with multiplicative RNGs (for example, the NAG generator). 4096 127-bit shift register generators occupy 127 planes, each generator having one bit to a plane. A row of PEs contains 64 '127' generators. A single '127' generator extends over 8 PEs; the mapping and updating sequence for a generator is illustrated (see fig. 1). Note that

- Updating order is not completely discretionary.
- any one PE contains data for 6 other generators interleaved; hence this generator can provide 18 sets of 7 bits that are from different generators.

This generator is simpler than the '4423', the complete APAL update code being:

```
QS      119 (M3)    !
QQ        E C 7     ! update bit 126
SIPQS   126 (M3)    !
DO        7 TIMES
QS      120 (M3 + )
QQ W C1
SIQPQS 0(M3 + )
SIQPQS 7(M3 + )
SIQPQS 14(M3 + )
  :
  :
:SIQPQS 119(M3 + )
```

This has been timed at 1.73 cycles/plane.

The above set of 4096 registers can provide 7 new high quality bits for each of 18 different numbers. This can be doubled to 14 bits per number by a second set of registers occupying a

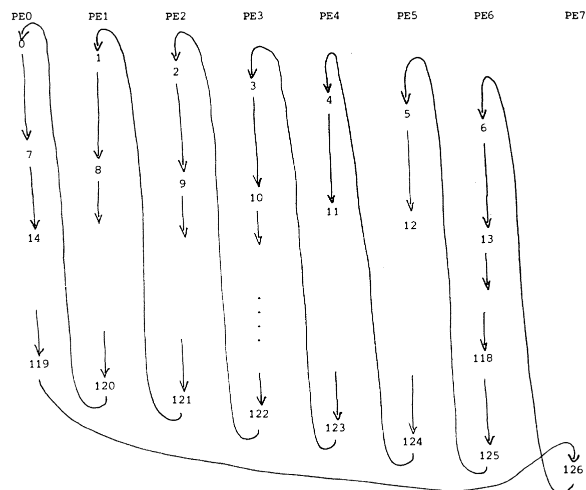


Fig. 1. Neighbouring PEs (W–E).

further 127 planes, either as another consecutive set, or interleaved with the first set in groups of 7 planes. This second set can be spread along a column instead of a row.

Four possible modes of use are:

(a) A library function (GO5 MANY) returning a single 16-bit integer matrix into user space. An extra copy stage (2 cycles/plane) selects 14 previously unused bits from different registers as the MS bits; the 2 LS bits are selected as repeat bits from other numbers. For 17 out of 18 calls only this copy stage is done; every 18th stage the whole generator is updated.

(b) A library function that provides the address of a single 16-bit integer. This would use an interleaved mapping to provide 14 consecutive previously unused bits, with the LS 2 bits being repeat bits from other numbers. Saving the copy doubles the performance.

(c) A library subroutine could produce 18 random 16-bit matrices. This would save calling overheads as well as copying. The 14-bit numbers would be spaced on 16-bit boundaries, with the extra 2 bits copied from other numbers.

(d) The generator produces 254 planes (in a specified mapping) that are used as desired by the application. It avoids copying, and good random numbers are formed from carefully selected planes that are not necessarily consecutive. The first application of the generator for a Monte Carlo simulation, (see ref. [6]) used it to produce 18 24-bit numbers by, for each number, picking off 14 bits from independent sequences plus a further 10 repeat bits from other numbers. The pattern used is given in the appendix. Performance is 24.4 cycles per matrix, or, from the users point of view, 1.02 cycles per plane.

In modes (b)–(d) the user accesses the generator space, but must not write to it. Also, it is important to note that the re-use of the bits for the 2 LS bits of a 16 bit number does not cause a problem with respect to the 'quality' of the RNs: this is demonstrated by the fact that all multiplicative RNGs have the property that the 2 LS bits are not random.

5. Summary for shift register generators

Table 4 summarises the performances of the different shift register generators.

6. Normal RNG

There are two normal RNGs on the DAP, both are based on the Box–Muller transformations; that is, if u and v are independent random variables both distributed uniformly on [0,1], then

$$x = \sqrt{-2 \log u}\ \cos(2\pi v)$$

and

$$y = \sqrt{-2 \log u}\ \sin(2\pi v)$$

are independent random numbers with a $N(0, 1)$ distribution (see refs. [1,7]).

On serial computers this technique has been avoided because of the computational cost of calculating square roots, logarithms and sinusoidal functions. However, on the DAP, the time to

Table 4

	Cycles/DAP plane			
	GO5_XOR	GO5_FAST	GO5_NEW	GO5_MANY
With separate output	96	12.75	7 [a)]	3.8 3.1 [b)]
Without separate output	95	11.75	6 [a)]	1.8 1.1 [b)]
	Random numbers/s			
Without separate output	13.1×10^6	106.6×10^6	210×10^6	813×10^6
(bits in a number)	(16)	(16)	(16)	(14–24)

[a)] Estimated; [b)] with re-use of bits.

Table 5
Pattern used in constructing 18 24-bit numbers. The entries in table 1 refer to both position in the register and to store plane; the two sets of 4096 registers are numbered independently

		1st generator set (M3)							2nd generator set (M4)														(M3)			
Bit significance	MS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	LS
1st number		0	18	36	54	72	90	108	0	18	36	54	72	90	108	125	107	89	71	53	35	17	125	107	89	
2nd number		1	19	37	55	73	91	109	1	19	37	55	73	91	109	124	106	88	70	52	34	16	124	106	88	
...																										
17th number		16	34	52	70	88	106	124	16	34	52	70	88	106	124	109	91	73	55	37	19	1	109	91	73	
18th number		17	35	53	71	89	107	125	17	35	53	71	89	107	125	108	90	72	54	36	18	0	108	90	72	

calculate any of these functions is comparable to the time it takes to multiply two REAL * 4 matrices (see ref. [8]).

The above transformations are converted directly into DAP Fortran. There are two versions available to users:

GO5 NAG NORMAL	2.27 ms.
GO5 XOR NORMAL	2.02 ms.

Appendix

Pattern used in constructing 18 24-bit numbers (see table 5).

A given generator supplies at most 2 bits to any number, and at carefully controlled positions.

The numbers are to be used in an outer APAL DO loop that iterates over the 18 numbers; to make this simpler, bits of a number are selected 18 planes apart, but this still results in independent sequences being used.

References

[1] D.E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, 2nd ed. (Addison–Wesley, Massachusetts, 1981).

[2] NAG Fortran Library Manual Mark 8, vol. 6.

[3] T.G. Lewis, Distribution Sampling for Computer Simulation (Lexington Books, 1975).

[4] H.S. Bright and R.L. Enison, ACM Computing Surveys 11 (1979) 357.

[5] DAP: APAL Language (ICL Technical Publication 6919, 1979).

[6] S.F. Reddaway, D.M. Scott and K.A. Smith, Comput. Phys. Commun. 37 (1985) 351.

[7] J.M. Hammersley and D.C. Hanscomb, Monte Carlo Methods (Methuen, London, 1964).

[8] R.W. Hockney and C.R. Jesshope, Parallel Computers (Adam Hilger, Bristol, 1981).

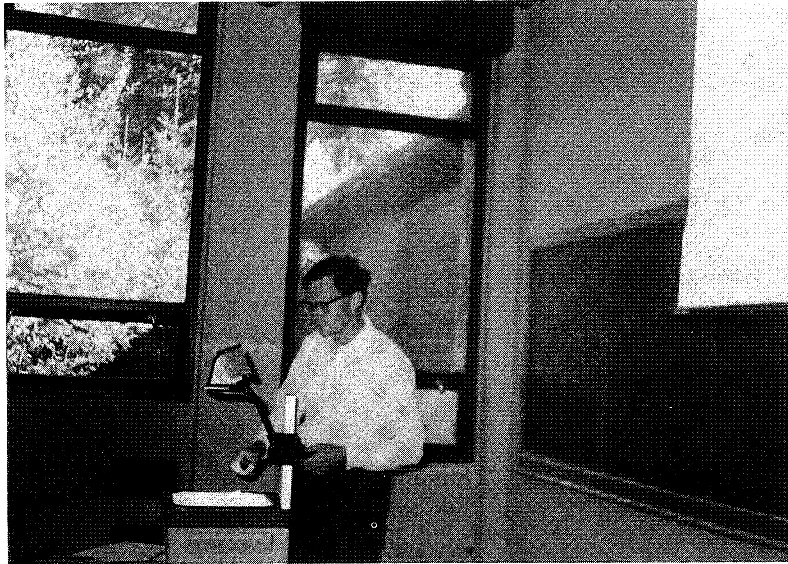
COURSE 14

SIGNAL PROCESSING ON A PROCESSOR ARRAY

Stewart F. REDDAWAY

ICL, Defense Technology Centre
Eskdale Road, Winnersh, Wokingham, Berk, RG11 5TT, UK

J.L. Lacoume, T.S. Durrani and R. Stora, eds.
Les Houches, Session XLV, 1985
Traitement du signal / Signal processing
© Elsevier Science Publishers B.V., 1987



Contents

1. Introduction . . . 834
2. Overview of Distributed Array Processors (DAP) . . . 834
   2.1. Digital Signal Processing (DSP) and programmable array processors . . . 834
   2.2. DAP background . . . 834
   2.3. DAP architecture . . . 835
   2.4. Programming MilDAP . . . 837
   2.5. Performance . . . 839
   2.6. Applications overview . . . 840
      2.6.1. Matrix algebra examples . . . 841
      2.6.2. Overview of a radar application . . . 842
   2.7. Some principles of application mapping . . . 843
   2.8. Some comparisons with systolic and wavefront arrays. . . . 844
3. Application case studies . . . 845
   3.1. MPRF radar . . . 845
      3.1.1. Some MPRF radar performance derivations . . . 847
   3.2. Large transforms . . . 849
   3.3. Convolutions . . . 851
   3.4. Image processing . . . 852
      3.4.1. Sobel edge detector . . . 853
      3.4.2. Histograms . . . 853
      3.4.3. Image understanding . . . 854
   3.5. Speech recognition . . . 854
   3.6. Graphics . . . 856
   3.7. Some other applications . . . 856
Note added in proof . . . 857
References . . . 857

1. Introduction

This work is based on experience of using, and planning to use, three generations of Distributed Array Processor (DAP) (see refs. [1–4]) from ICL. These are SIMD machines; other machines with some similarity are:

CLIP 4 (University College, London),
MPP (NASA and Goodyear),
GRID (GEC),
SCAPE (Brunel University).

There are also some similarities with systolic and wavefront arrays.

Section 2 is an overview of DAP, while section 3 is an examination of some application case studies.

2. Overview of Distributed Array Processors (DAP)

2.1. Digital Signal Processing (DSP) and programmable array processors

A wide range of digital signal processing can be implemented on a standardised architecture. Compared with custom hardware this allows:

(a) lower risk,
(b) more functional agility,
(c) functional evolution during in-service lifetime,
(d) keeping the same software through technology upgrades.

From another viewpoint, algorithms, languages and architecture all influence each other, and architecture is influenced by Very Large Scale Integration (VLSI).

2.2. DAP background

There is a considerable history of processor array machines, and this section gives a brief background on the machines the author has been

involved with. The connection with signal processing is more recent.

The pilot DAP system was completed in 1976 and had 1024 Processing Elements (PEs) arranged 32 × 32. This led to a similar first-generation DAP product with 4096 PEs organised 64 × 64 made from MSI technology, and connected as a backend to an ICL 2900 mainframe. These large DAPs (330 PCBs) were marketed with a number crunching emphasis and 6 of them were delivered during 1980–1983. One of these machines, supporting about 200 users, is at Queen Mary College, London, where there is a DAP Support Unit to help (mainly academic) users. These machines have lived up to their technical expectations with high performance on a wide range of mainly scientific applications.

The first second-generation, or MiniDAP machines were delivered at the end of 1985. The main focus of this course is a MiniDAP LSI machine known as MilDAP, which occupies 16 PCBs and has a 32 × 32 array of PEs. Its initial host machine is a PERQ, but it is less tightly coupled to it than with first generation, and the MilDAP will go into equipment without a host system. An important new feature is the (programmable) Fast I/O hardware which opens the way to high data rate applications such as graphics, signal and image processing.

Work has started on a military R & D contract, known as VDAP, to investigate machines, with several different array sizes, that exploit VLSI by having much more powerful PEs*.

2.3. DAP architecture

The MilDAP has a 32 × 32 array of one-bit processing elements (PEs) (see ref. [4]) which each have access to 8 or 16 Kbits of RAM. Communication can be two-dimensional (or with a little software help one-dimensional), or, as we shall see, global. The two-dimensional N–S and E–W features are nearest-neighbour connections and highways, both with bit-serial PE connection. The high data throughput and computing capacity is achieved by the parallel operation of the array: whether the operation is a data shift between PEs or an internal instruction, every PE participates, subject only to a veto imposed by an "activity" bit which can locally inhibit PEs. DAP processing is controlled by a Master Control Unit (MCU). This can form a matrix by broadcasting a 32-bit

*See note on p. 857.

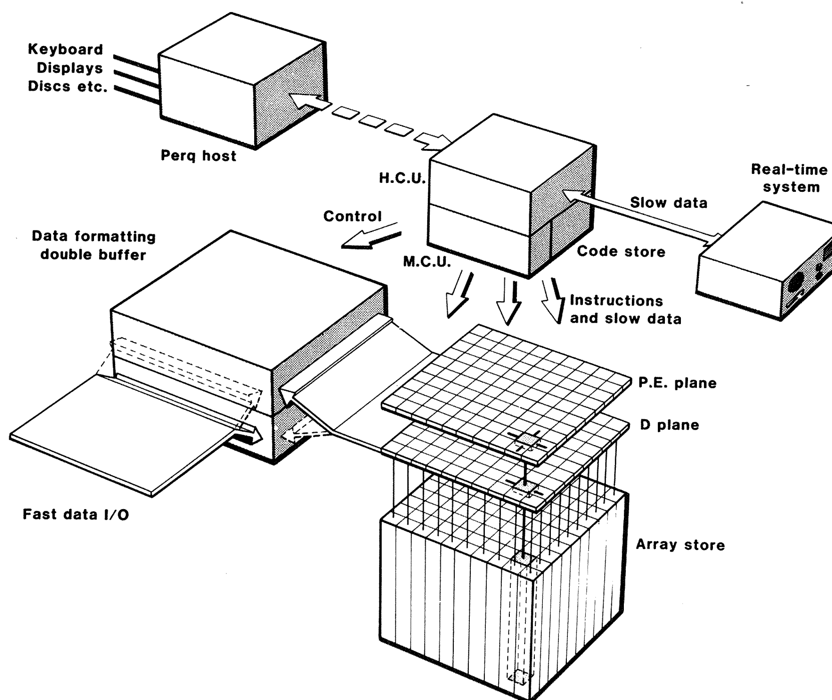


Fig. 1. Schematic representation of MilDAP showing the relationships between the Host Connection Unit (HCU), Master Control Unit (MCU) and the Processing Element (PE) plane; and the D input/output plane and the fast data I/O (FIO) unit.

vector along the highways to every row or every column of PEs, and the AND of a matrix can be performed in row or column directions to provide a 32-bit vector to the MCU. If the MCU then branches on that vector being all TRUE we have a global test in just 2 instructions. A scalar can be globally broadcast. The DAP can validly be regarded as a partitioned memory with processing power distributed across it, avoiding limitations to available memory bandwidth.

Figure 1 shows a simplified schematic of the MilDAP developed jointly by ICL and RSRE; it has a cycle time of 150 ns using gate array and standard RAM technology and occupies about 1 cu. ft. The Fast I/O unit includes two buffers, each of 16 K × 32 bits. The Fast I/O unit can be "programmed" to achieve a wide range of data reorganisations. At the corner-turning level, this is important since processing is fastest for data stored "vertically" below each processing element, whereas

external data is usually in a word parallel or "horizontal" format. Higher-level reorganisations are also very powerful in matching convenient I/O and data structures for efficient processing. During data input, words are written into one of the two buffers through a 32-bit wide bidirectional interface, and then read out (reorganised) into the D bit-plane (see fig. 1), along 32 "row" channels by fast data shift operations independent of other array operations. Once the D-plane is full, an array cycle is stolen to write the data plane of 1024 bits to a main memory store plane. Even at the maximum data transfer rate of 40 MBytes/s, less than 5% of array processing cycles are stolen. The procedure reverses for data output.

The bit-serial but highly word-parallel nature of MilDAP processing means that the machine is not committed to a particular data representation. This allows accuracy-speed tradeoffs right down to one-bit working for logical comparisons which can be made at speeds of several Gbits/s.

The program-development system is based on a separate PERQ host computer which provides the compiler, assembler, library of routines and other utilities including a DAP simulator*. Programs are down-loaded from PERQ during development and from a convenient storage medium such as a disc or ROM in stand-alone use.

2.4. Programming MilDAP

MilDAP has a very efficient assembly language (APAL) [5] and a high-level language DAP-Fortran [6] which is a parallel extension of Fortran, the main difference being the ability to process complete arrays as single items. As an example, consider the Fortran code:

```
    DO 9 M=1,32
    DO 9 N=1,32
      IF (A(M,N).GE.0) GOTO 9
      A(M,N)=A(M,N)+5
  9 CONTINUE
```

This is very inefficient on a serial machine partly due to the `IF`. In the corresponding DAP-Fortran statement,

```
A(A.LT.0)=A+5
```

*The program development system is now also available on VAX and SUN workstations.

the Boolean matrix "A.LT.0" is used as a mask so that only those values of A corresponding to a "TRUE" value are changed. The contrast with sequential machines on conditional operations is important: the sequential machine performs a conditional jump, whereas the DAP will typically use activity control to perform a masked assignment. In addition to the basic functions which have been extended to take vector and matrix arguments, a large number of other functions are provided as standard. These include (a) functions to create arrays from scalars, (b) row or column broadcasting, (c) maximum element, and (d) row, column and matrix sums and many others. DAP-Fortran inspired many of the parallel processing facilities planned for Fortran 8X.

Considerable improvements in speed can often be achieved by coding critical sections in APAL. APAL, for example, allows full advantage of variable wordlengths. An example of APAL code for the addition of two 5-bit integer matrices is

```
 AT
 CF
 DO 5 TIMES
 QS 4 (M3-)
:SICPCQS 4 (M4-)
```

where the MCU register M3 points to the sign bit of the first matrix, and M4 to that of the second matrix. An explanation of the above code is as follows:

(1) Set activity control register A to "true".
(2) Clear Carry register C.
(3) Loop over number of bits.
(4) Fetch first matrix, starting at low-order end.
(5) Do the add, overwriting the second matrix under activity control and preserving carry in C for the next pass of the loop. The effect of the negative sign is to automatically decrement the effective plane address.

Program development is performed on an ICL PERQ II workstation which is connected to the MilDAP via the Host Connection Unit (HCU). All the supporting software, DAP-Fortran compiler, APAL assembler and consolidator is run on the PERQ which accepts and displays diagnostic and trace information from the MilDAP. The PNX (UNIX) operating system provides file handling and screen editing

Table 1
MilDAP/MiniDAP and DAP-3 performance on matrix operations

Precision	Operation	MilDAP/MiniDAP (150 ns cycle)	DAP-3 (100 ns)
logical	logical	> 1 GOP	> 1 GOP
8-bit integer	add	280 MOP	420 MOP
	Mult.(M–M)	42 MOP	60 MOP
	(M–constant)	100–200 MOP	140–280 MOP
16-bit integer	add	140 MOP	210 MOP
	Mult.(M–M)	10 MOP	14 MOP
	(M–constant)	30–100 MOP	40–140 MOP
32-bit real	add	8.6 MFLOPs	12 MFLOPs
	mult.(M–M)	5.1 MFLOPs	7 MFLOPs
	square	10.9 MFLOPs	16 MFLOPs
	squareroot	7.4 MFLOPs	11 MFLOPs
	divide	4.1 MFLOPs	6 MFLOPs
	log (base e)	4.2 MFLOPs	6 MFLOPs
	max. of 1024 numbers	41 MFLOPs	60 MFLOPs
data	scan for match	6×10^9 bits/s	9×10^9 bits/s

facilities. In addition, a MilDAP simulator, written in "C" allows program development (including timing estimates) on workstations without a DAP. A comprehensive library of standard functions such as FFTs, matrix operations, random number generators, etc. is also available. The MilDAP can be used in field situations in a stand-alone mode without the PERQ.

2.5. *Performance*

The best serial algorithms are often not the best parallel algorithms. An example of an inefficient serial algorithm which is particularly suited to parallel computation is Batcher's "bitonic sort" [6], which has been applied to DAP in ref. [7]. To sort N items, this method requires $(\log N)/4 \times$ more comparisons than an efficient serial algorithm, but this is a small price to pay for a regular algorithm with $N/2$ parallelism.

The bit-serial MilDAP has some unusual relative speeds for functions; see table 1 for some matrix operations. Other performance figures for MilDAP and DAP-3 are given in table 2.

Table 2
MilDAP/MiniDAP and DAP-3 performance

Operation	MilDAP/MiniDAP (150 ns)	DAP-3 (100 ns)
Floating point–Peak MFLOPs		
24 bit	13	19
32 bit	9	13
64 bit	4	6
Fixed point–Peak MOPs		
logical	2000	3000
8 bit	260	400
16 bit	140	210
32 bit	70	110
FFT examples (APAL coding)		
(1) 1024 64-point FFTs		
complex 10-bit input		
time (ms)	8.5	5.6
overall (MOPs)	160	240
(2) 512 × 512 2D-FFTs		
real 8-bit input		
time (ms)	100	70
overall MOPs	80	120

Basic techniques of array processing on DAP are covered elsewhere; for example, ref. [3] describes maximum element, scalar-matrix multiplication, square root and "recursive doubling".

2.6. *Applications overview*

Most, but not all, applications where the emphasis is on performance and where the processing load is non-trivial have good DAP solutions. The following is a semi-random list of areas:
matrix arithmetic,
field problems,
sorting,
text processing (e.g. compilation),
speech,
image processing,
graphics,

pattern recognition,
CAD,
searching,
picture searching,
encryption,
AI/Prolog/Lisp?

Applications need significant work expressible as array operations. This can be not only arithmetic, but also many other forms, such as:

data movement,
table look-up,
searching,
sorting,
conditional operations.

Applications are best approached top down, with the whole application (or a large part of it) considered as to how it best maps on to a processor array. If this high-level mapping is done well, the details often fall nicely into place. If too small a part of an application is considered, it is often more difficult to get a good mapping.

2.6.1. Matrix algebra examples

Before coming on to more specifically signal processing we look at some matrix algebra.

A DAP-Fortran code for matrix multiplication when the matrices match the DAP size is:

```
      C=0
      DO 10 K=1, 32
   10 C=C+MATC(A(, K))*MATR (B(K,))
```

Figure 2 shows the data movements involved with, for example, a column of matrix **A** being expanded to a temporary matrix. The time for such a code on MilDAP is about 15 ms. Larger matrices involve handling indexed sets of 32×32 sub-matrices, with various possibilities for the mapping into sub-matrices. (See section 3.4 on image processing for a discussion on sheet and crinkled mapping.)

Matrix inversion is discussed in refs. [2] and [8]. For both inversion and multiplication, data movement overheads can be reduced for larger matrices.

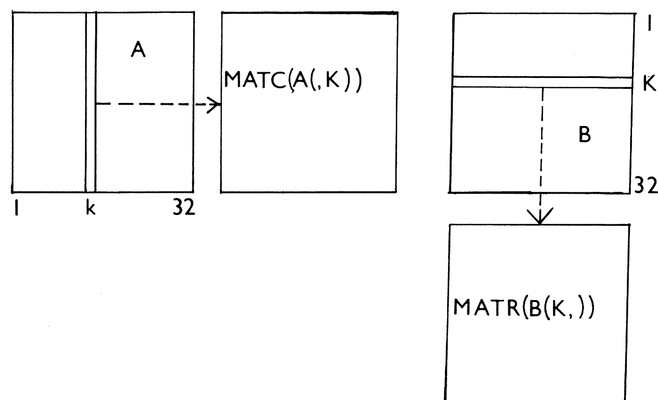


Fig. 2. Data movements for matrix multiplication.

2.6.2. Overview of a radar application

This application has been implemented on MilDAP and is considered in more detail below. Here we use it to illustrate some points.

The implementation is an air-to-air search mode of a multi-mode airborne radar; the same DAP hardware can be programmed to process many different modes. This mode has a medium pulse repetition frequency, and the characteristics are shown in table 3. The performance on the stages is shown in table 4. Note that no stage dominates the time, and so a special-purpose piece of hardware suitable for just one type of processing (for example FFT) could not have a dramatic impact, quite

Table 3
Characteristics of MPRF air-to-air search mode

Algorithm	Types of processing
Spectral decomposition	
Window and FFT	unconditional +, −, *
Log modulus	special
CFAR	unconditional +, −
	conditional +, −
	logical
	:
Ambiguity resolution	table look-up
	data movement
	:

Table 4
MPRF radar performance; 60 000 radar complex (2×10 bits) points at $0.5\,\mu s$ sampling

Stage	MilDAP	
	Time (ms)	MOPs
"Input"	1.4	
Window 64-Point FFT (10–14 bit) fixed point	9.0	150
Log modulus	3.8	
CFAR	8.2	200
Ambiguity resolution (depends on # targets)	< 2.6	
Total	< 25.0	
Time available	35	

apart from the cost of interfacing. The wide range of different types of processing required is typical of the increasing sophistication of signal processing applications; this means that special hardware solutions have to be increasingly complex, in contrast to solutions with programmable hardware.

It should also be noted that the implementation batched up 60 000 points for processing. This achieves greater throughput by allowing the exploitation of high-level parallelism and at the same time using efficient serial algorithmic techniques at the detailed level; this also minimises communication between PEs, and in this example each 64-point FFT is done serially entirely in a single PE, with 1000 FFTs being done in parallel across the array of PEs.

Signal processing typically involves several very different stages, some with data-dependent (conditional) processing and often with some Boolean processing. Data precision is generally fairly low. With specialised hardware a sequence of special boxes or chips are needed, and much emphasis must be given to transfers between these; performance is governed by the speed of the slowest unit, and flexibility is lacking. With a programmable SIMD processor array such as DAP we have a "one box" solution which offers good flexibility and overall performance.

2.7. *Some principles of application mapping*

It is important to minimise communication and I/O. The following (related) techniques can be useful.

Go for high-level parallelism. Low-level algorithms are often best done serially, with many of them being done in parallel. This may involve (a) using more than the minimum memory, or (b) increasing the latency (delay between input and output); but throughput is increased.

Localise the processing. For example, in applying neighbourhood operations over whole images "crinkling" the image so that one PE deals with a local area is superior to mapping a "sheet" of the image across the array of PEs. The latter involves work to join the sheets, more data movement, and often extra arithmetic.

If data to be combined is held in a sequentially accessed memory, then "communication" is achieved in the sequence of addresses given to the memory. For example, the patterns of "movements" needed for FFT or for Batcher-bitonic sorting are "free" when built into the addresses used in the code, provided the combining data is all in the same PE, and (from high-level parallelism) similar operations are replicated across the array.

"Oversize" problems are advantageous. For example dealing with 256×256 matrices on a 32×32 DAP. This allows the above techniques to be used. Algorithms are a combination of serial and parallel.

In moving from one processing stage to another it is sometimes worthwhile reorganising the data in the array. (A uniform spread of data across the PEs is retained.) An example is if one stage is one-dimensional processing on a multi-dimensional array, and another stage is two or more dimensional neighbourhood processing. The former stage should have data from one line in the dimension being processed in as few PEs as possible, while the latter should have a local section along all directions being processed in as few PEs as possible so as to minimise routing. The MPRF radar is an illustration.

I/O can be minimised by doing as much of an application as possible in one array, with no intermediate I/O. Having memory inside the array is very important, both for data and constants. In MilDAP, the use of processing cycles for data movement can be minimised by using the Fast I/O to re-order the data on input or output.

2.8. Some comparisons with systolic and wavefront arrays

There are some similarities between the processor arrays described in this course and systolic and wavefront arrays, but there are important differences.

With DAP a large memory mixed with processing power helps the use of the techniques in section 2.7 so as to minimise communication and

maximise throughput. Most systolic/wavefront arrays do not have large memories distributed in the array.

Systolic/wavefront arrays tend to be tailored to a particular algorithm or algorithm class, and hence the emphasis is on hardware. This contrasts with the fixed design of the DAP with the emphasis on programming the applications. Hence DAP tends to tackle complete applications and systolic/wavefront arrays tackle parts of applications.

Systolic/wavefront arrays deal with individual or local parts of algorithms very fast, with much of the work being communication. With the use of high-level parallelism on DAP the individual or local parts of algorithms may be serial and slow; but they are efficient, with minimum communication and high overall throughput.

3. Application case studies

System requirements have been programmed for MilDAP to explore the best ways of mapping algorithms onto the machine and to quantify its performance on total problems. The performance rating of any machine is rarely achieved in practice, and with DAP it is particularly necessary to overcome the suspicions: (a) that algorithms with branching instructions cannot exercise all the PEs effectively, and (b) that the fixed size of the array of PEs limits us to problems with special array sizes. These suspicions are shown to be unfounded by the wide diversity of algorithms described in this course.

3.1. MPRF radar

Modern airborne radars require complex algorithms applied to data of MHz bandwidths. Moreover radars are now multi-functional with variable parameters such as Pulse Repetition Frequency (PRF) and pulse width, and have distinct operational modes like ground attack, air interception, terrain following, synthetic aperture mapping, etc. This necessitates a programmable processor.

One of the most demanding tasks is the "medium PRF" aircraft interception mode. This comprises several algorithms which exercise the processor by: (a) input and output of data; (b) arithmetically intensive processing on multiple FFTs; (c) data dependent, arithmetic and logical, local area operations for Constant False Alarm Rate (CFAR) processing in strongly varying clutter environments; (d) data reorganisation and logical comparisons to resolve ambiguities in both range and velocity

(Doppler effect). This application was programmed for MilDAP by Reddaway et al. [9] in order to expose any deficiencies, and to demonstrate versatility combined with high throughput.

The radar input to the fast I/O buffer is 10 + 10 bit complex samples at maintained rates of 2 MHz (5 MBytes/s). In addition to this, housekeeping instrumentation data at a much slower rate will be input via a separate channel. The radar PRF varies from pulse burst to pulse burst through a cycle of 5 different values, and blocks of data are combined together in the fast I/O hardware. This organises the data into the most efficient distribution in the DAP store, and in this case inputs half a bit-plane of 32 × 32 bits to the array memory for each "stolen" clock cycle. Data is processed in batches of 10 pulse bursts; while processing is proceeding, the data for the next batch is being input to the DAP array memory.

Bursts of 64 pulses sampled in up to 128 range gates are windowed using fixed 10-bit real weight vectors, Fourier-transformed to 14 + 14 bit complex spectra and converted to 10-bit log-modulus power estimates (70 dB dynamic range) in 64 Doppler cells. The total number of range gates over the five bursts totals rather less than 512, including an allowance for duplication of range gates to cater for cyclic wrap-around in range. Thus 2 sets of 5 bursts are laid out across the 1024 PEs with all 64 points for each range gate in the same PE; the fast I/O subsystem has also introduced index reversed ordering for the 64 points so that after the FFT the ordering is natural. The FFTs are entirely contained within single PEs, and are done with full serial optimisation for radix 8; parallelism is obtained from doing 1000 independent FFTs in parallel. Extensive use is made of efficient scalar–matrix multiplications in the FFT computations, because each PE is using the same multiplier coefficients concurrently; a code generator automatically minimises the add/subtract work by analysing the bit-patterns of the constant multipliers. During the FFT, there is precision growth at the MS end of the fixed-point numbers, and noisy bits are dropped at the LS end. The log-modulus function is specially coded; it has some similarity to conversion from fixed to floating point. The spectrum analysis phase is done at a rate equivalent to 220 ns per sample, or 45% of the corresponding data collection interval. Performance could be improved somewhat by doing 60 or 72 point FFTs by Winograd or prime factor techniques.

CFAR detection includes (i) smoothing in the two-dimensional range–Doppler domain using a simple two-dimensional filter with cyclic boundaries, (ii) time averaging where clutter discontinuities are found, (iii) comparing the signal with the local clutter level, (iv) removing

detections where the known angle of look and aircraft velocity result in high-clutter levels, and (v) eliminating returns which behave in range or velocity in a way inconsistent with expected target behaviour. The three-dimensional filters are continually updated on the basis of recent history. The data is rearranged between the spectrum analysis and CFAR processing, so that a single PE holds 16 Doppler points from 4 range gates instead of 64 Doppler points from 1 range gate; this reduces both communication and arithmetic in the two-dimensional neighbourhood operations. (A square 8×8 section of points would be ideal, but the cost of the extra data rearrangement means the optimum is 16×4.) The DAP activity control is efficient in dealing with the data dependent processing. The CFAR computation takes 140 ns per sample (28%).

The medium PRF mode is doubly ambiguous. The apparent detections are first collapsed in Doppler, keeping the Doppler location of the detection with the peak power value, and are then "unfolded" in range to form a binary vector showing the possible target ranges corresponding to the observed detections out to 1280 range cells. Five of these range patterns, each from a different PRF pulse burst, are examined, and a 3 out of 5 detection criterion applied to ranges where a toleranced coincidence is found. Finally, these potential target detections are checked for consistency in Doppler by referring back to their original frequency values (before collapsing) and generating by table look-up the possible real frequency bin. These table entries are logical maps of the real velocity (frequency) span divided into 1024 bins, and stored as DAP logical matrices with the possible bins marked (up to 50 per ambiguous frequency bin, when all allowance for tolerance is included). These Doppler frequencies are then checked for coincidences, again with a 3/5 criterion, to finally confirm the targets. This comparing of (Boolean) detections is particularly efficient with 1-bit PEs. Resolving the radar ambiguities takes less than 40 ns per sample (8%).

The above items total 400 ns per sample, whilst the sample separation is 500 ns. Extra dead time exists between pulse bursts so there is 30% rather than 20% spare processing capacity. This shows that MilDAP provides a flyable processor adequate for this extremely demanding role.

3.1.1. Some MPRF radar performance derivations

More details of some of the algorithms and performance derivations follow here.

The operation counts for a 64-point FFT on complex data are given in table 5. The general rotations use 3 multiplies and 3 adds. The multiplies use the scalar-matrix code generator. The average precision is about 12

Table 5
64-Point FFTs–Operation counts and performance for 1024 off

Operation	Number of complex operations	Number of simple arithmetic operations	
		multiply	add
Complex adds	384	–	768
45° Rotations	36	72	72
General rotations	44	132	132
Total		204	972
Approximate average cycles/ operation		100	35
Totals		20400	34000
			54400 cycles total

bits. (At each stage the precision grows 1 bit at the MS end, and loses 0.5 bit at the LS end). The cycle count is for 1024 independent FFTs; overheads are about 10% and the time is 9 ms with a 150 ns cycle time.

The log modulus was specially coded and takes 390 cycles per result matrix. For 64 result matrices and 150 ns cycle time, this gives 3.74 ms.

The data rearrangement, using a methodology known as "musical bits" [10] takes place in two stages, one involving a route distance of 1 PE and one a distance of 2 PEs. The costs are 4 and 5 cycles/BIP (BIt-Plane). At this point there are 640 BIPs, so the time is 0.87 ms.

In the CFAR, 5 × 5 cell averaging is done with an algorithm that for one dimension adds the points in pairs resulting in half as many results (half an add per point), and then adds the results so as to get all sums of 4 starting at 2-point intervals (half an add per point). Each of these sums is used to form 2 sums of five by adding the appropriate original point (1 add per point). Thus, there are 2 adds per point for one dimension; applying the same procedure in the other dimension gives all 25-point sums at a cost of 4 adds/point. The central point is subtracted and the resulting 24-point sum is divided by 3 (effectively = 24), equivalent in cost to 2 adds.

The total arithmetic is thus about 7 adds per point. The precision growth during the adds is limited by dropping at intervals some LS bits; the average precision is about 11 bits, with a matrix add costing about 32 cycles. The arithmetic time is thus about 2.2 ms. However, the routing and end effects due to each PE holding only a 16 × 4 section increase this to about 4 ms. This represents about half of the total CFAR cost.

Finding the maximum Doppler cell is a local operation and therefore is not like the DAP-Fortran maximum functions that apply to one DAP matrix. The heart of the work is within one PE where each point is compared with the maximum so far, and (conditionally) copied in if it is bigger. The comparison costs 2 cycles/BIP and the copy costs 2.5 cycles/BIP. In total there is about 20% extra work because one range gate is spread over 4 PEs. This work thus takes about 0.52 ms, to which must be added a small amount for finding the position of the maximum Doppler cell in each range gate.

3.2. Large transforms

Doing one 64-point FFT in every PE was dealt with in the MPRF radar. Here we consider some big transforms.

First, consider the following transforms with 64 K complex data points:

$64 \times 32 \times 32$	3D
64×1024	2D
32×2048	2D
64 K	1D

MilDAP would map these across its 32×32 array and down the memory of each PE to a depth of 64 points. The best approach is to perform a 64-point FFT within the PEs, as in the radar case; this is followed by using "musical bits" to transpose the cuboid of points so that lines of 32 points in one of the array directions end up "vertically" in single PEs. (The data can be considered as 2 cubes $32 \times 32 \times 32$ stacked above each other. Each cube is transposed or rotated about 1 axis.) Using the data now in each PE, two 32-point FFTs are performed (in every PE). The procedure is now repeated by transposing about the other array direction and doing two more 32-point FFTs in every PE.

As described, a $64 \times 32 \times 32$ FFT has been performed, with the data left in index reversed order. (If necessary this can be unscrambled with more "musical bits".) The other FFTs can be performed by inserting appropriate twiddle multiplications.

The cost for a careful APAL coding, in the style of the MPRF radar, and for an average precision of 16 bits (11 bits growing to 20 bits) is as follows:

- 64-Point FFTs with 16-bits average precision costs about 80 000 cycles.

- Transposition costs about 30 cycles/BIP, so with 2 K BIPs this is 60 K cycles.
- Two 32-point FFTs with 16-bit average precision costs about 64 000 cycles.

The cost of a twiddle multiplication is quite high, as these multiplies must be matrix–matrix; at 16-bit precision the cost for a complete twiddle multiply is about 80 K cycles. Thus the cost of the three-dimensional transform is about 328 000 cycles (49 ms), for the two-dimensional transforms about 408 000 cycles (61 ms) and for the one-dimensional transform about 488 000 cycles (73 ms).

Other mappings are possible. For example the two- and three-dimensional transforms can save some multiplications by having some of the multiplies within the 32- or 64-point transforms combined across the dimensions of the FFT.

With a 512 × 512 FFT that is mapped with the two-dimensional structure matching the DAP two-dimensional array, one line of 512 points maps to a row of 32 PEs 16 points deep in memory; the two dimensions give a total of 256 points per PE. There are two equally good ways of proceeding.

Both methods start by performing 16 × 16 two-dimensional FFTs on the data in each PE; advantage can be taken of combining the multiplies across the dimensions. The first method then rotates in new 16 × 16 sections into each PE and repeats the 16 × 16 FFTs, and then completes the transform by rotating in the final 2 × 2 sections. The second method rotates lines of 32 points into PEs and performs 32 point FFTs, and then repeats this in the other array direction. In image processing the data will be real, and adapting the algorithms is left as an exercise for the reader; the time for 16-bit average precision is about 100 ms.

Another image-processing example is the generation of cosine transforms on all 8 × 8 segments of a 512 × 512 image (for the purpose of compression). If the data is 8-bit there are 2048 BIPs, and if the mapping is "crinkled" no data routing is needed. On either MilDAP or a third-generation 8 × 8 DAP the time should be about 10 ms.

Multiple independent transforms where it is not possible to have one transform per PE should have transforms confined to as few PEs as possible, usually as square a section of PEs as possible. For example, 128 independent 512-point transforms should have a 512-point transform in a 4 × 2 section of PEs. (Sometimes earlier or later processing may dictate a linear section, 8 × 1 in this case.)

Prime factor (or Winograd) can give a modest performance improvement. Individual transforms confined within a PE are no problem. In

other cases it is even more advantageous than for power of 2 FFTs to have as many points per PE as possible. For factors that cannot initially be mapped in the memory of one PE, the points are "rotated" into position to do that prime factor transform.

The first prime factor example is for 64 independent 1680-point ($3 \times 5 \times 7 \times 16$) transforms. Each transform is put in a 4×4 section of PEs, with 105 points per PE. When the 16-point transforms are needed, "musical bits" techniques are used to rotate the 16 points into "vertical" lines. This is repeated 7 times until there are 112 matrices of points; some PEs in the top 16 matrices are empty. The "inefficiency" in the processing is confined to the 16-point transforms and is only $112 - 105 = 7$ out of 112; overall this is a negligible 2% in this case.

The second prime factor example is a single two-dimensional 16×5040-point transform mapped onto a third-generation 16×16 DAP, giving 315 points deep in each PE. To do the 16-point transforms, they are rotated into position; this is done 20 times to generate a total of 320 points deep, an "inefficiency" of 5 in 320.

3.3. Convolutions

The use of transforms to perform (large) convolutions is well known, as it reduces computational complexity from order N^2 to order $N \log N$. The FFTs described above can be used for this purpose.

Number theory can be used in two ways. One is in the prime-factor and Winograd transforms mentioned above. The other is in the number representations used; these Number Theoretic Transforms (NTT) do arithmetic modulo special numbers, such as Fermat numbers ($2^{2^n} + 1$). The advantages are that the multiplications in the transforms can be reduced to special forms of add/subtract (efficient to implement on DAP) and the results are exact (no rounding error). The main disadvantage is that higher precision must be used, both to have unambiguous results and to cater for long transforms without having to use multi-dimensional methods.

One application implemented a few years ago was the checking of large Mersenne numbers (of form $2^p - 1$) to see if they are prime. The standard test involves repeated squaring of numbers of p bits precision; with p about 100 000 this is a major task. The number can be divided into appropriate precision "digits" and the square done by convolution methods. With Fermat Number Transforms (FNT) good results can be achieved, and it is just worth going to a recursive, or multi-dimensional, approach similar to the Schonhage–Strassen algorithm for very high

Table 6
Checking Mersenne numbers for primes, with p around 86 000

Computer	Time (min.)
CRAY-1 direct	140
CRAY-1 divide + conquer	70
CYBER 205 transform	60
DAP(4096PE) transform	38

precision multiply. A one-dimensional approach was implemented on DAP, and in 1983 the present author compiled table 6 showing the speed of checking Mersenne numbers for different implementations on different machines. The first-generation (64 × 64) DAP checked 16 numbers at a time.

Both FFTs and FNTs have been studied for an area correlation application in image processing (see section 3.4.3); the performance is similar.

3.4. *Image processing*

Since most images have many more pixels than the DAP has PEs, we have a choice in the way we section the image and pack it into the array store. Two extreme mappings are "crinkled" and "sheet" [11]. With the crinkled mapping (for 32 × 32 MilDAP) the image is divided into 32 × 32 local areas (each of 8 × 8 pixels for a 256 × 256 image), with each area put into a single PE. A DAP matrix is a regular sample of the whole image. With the sheet mapping, the image is divided into 32 × 32 sheets of pixels which are then mapped onto DAP matrices. For local operations on whole images the "crinkle" mapping is normally best in that it minimises the number of data shifts required and avoids sub-image boundaries. For processing where it may be desirable to do some work exclusively on a limited area, then the sheet mapping allows this; for example, some approaches to blob extraction favour a sheet mapping. Sheet and crinkled mappings can be advantageously combined. For example, an image can be divided into 64 × 64 sheets which are then crinkled into the 32 × 32 array; this limited crinkling avoids the worst performance effects of the sheet mapping on whole image local operations. The fast I/O can input most mappings "free" to the processing.

Table 7
Processing 256 × 256 images of 8 bits

Operation	Time (ms)	
	MilDAP/MiniDAP	DAP-3
3 × 3 Sobel edge detection	0.8	0.54
3 × 3 Median filtering	6	4
Histogram	16	11
Grey-scale normalisation	2	1.4
Summing or differencing	0.22	0.15
Shrink or expand (1-bit data)	0.6 per stage	0.4
2D FFT (8-bit growing to 17-bit)	22	15

Some times for APAL coding on 256 × 256 8-bit images are given in table 7. These benchmarks for MilDAP show that the machine is able to achieve on-line processing for requirements relating to images in TV format. Furthermore, the capacity for inputting data at 40 MBytes/s is more than adequate.

3.4.1. Sobel edge detector
The Sobel operator is a very special form of a 3 × 3 convolution operator:

$$\begin{array}{ccc} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{array}$$

The number of adds per point with careful use of intermediate results is 3. For a 512 × 512 image and a fully crinkled mapping the neighbour routing is only about a 10% overhead. Thus for 8-bit pixels and 150 ns cycle time, the total time for the application of the Sobel operator in one direction is $16 \times 16 \times 3 \times 8 \times 3 \times 150 \times 10^{-9} = 2.8$ ms. [There are 16 × 16 pixels/PE, 3 adds/pixel, 8 bits/pixel and (2.5 or) 3 cycles per BIP for the adds.]

For a combined mapping with 64 × 64 sheets, the time nearly doubles to about 5.5 ms due to work stitching the sheets together and extra routing. For a fully sheet mapping, the time nearly doubles again to about 10 ms.

3.4.2. Histograms
It is not obvious how to get a good histogram algorithm on DAP, and three very different algorithms are considered in ref. [11]. The pixel-

decoding method is best for short pixels and surprisingly good results can be achieved. It works by decoding each pixel matrix into each of its logical matrix grey levels, and then summing the logicals for each grey level.

It is important that the summation is done within the PEs first, as this produces a large reduction in the amount of data, before summing across the PEs. For the latter, the parallelism can be maintained in the later stages of summation by performing in parallel the summation of an increasing number of different grey levels. For large images the total cost can be kept down to around 6 or 7 cycles per grey level logical matrix. For 8-bit pixels the time is about 60 ms for a 512 × 512 image and 16 ms for a 256 × 256 image. For 6-bit pixels, these times are, of course, 4 times faster, and techniques can often be used to limit the number of grey-scale values needed.

VDAP is planned to have some local index capability which will greatly speed up histograms.

3.4.3. Image understanding

Area correlation finds the best fit of a two-dimensional image against two-dimensional reference data. If no rotation or scale change is involved, this is a two-dimensional convolution which can be done by transforms. For example, fitting a 300 × 150 image to a 512 × 256 reference (to give a 213 × 107 result) could be done in 100 ms; finding the maximum, and hence the best-fit position, takes only another 0.3 ms. Similar methods can also be used for Difference Of Gaussians (DOG) convolution operators.

Another image-understanding algorithm being studied with encouraging results is a statistical classifier based on features extracted from blobs, which in turn were formed by multi-channel DOG analysis and area filling (see section 3.6). More syntactical methods are a matter for future research.

3.5. Speech recognition

The widely-used dynamic programming method known as Dynamic Time Warping (DTW) has been implemented in DAP Fortran plus a little APAL [12]. Performance using 16-bit data and a Euclidean distance measure (sum of the squares of the differences at each frequency) is over 5 words per second with a vocabulary of 102 words, assuming 19

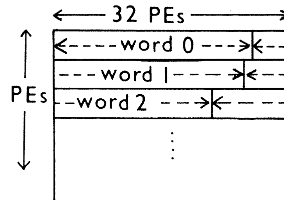


Fig. 3. Mapping of word templates for speech recognition.

frequency channels and an average of 30 spectra per word. Data as short as 4 bits is often used, and this offers nearly an order of magnitude performance improvement on DAP, because time is dominated by the square operation. With 8-bit data and APAL coding about 15 words per second is achieved (1.5 words per second with a 1000-word vocabulary). The DAP mapping is illustrated in fig. 3. With an average of 30 spectra per word, 34 words are evaluated simultaneously. The computation of the minimum-cost function is illustrated in fig. 4. The cost is accumulated by moving from corner to corner using a local decision function.

Recently an advanced technique [4] has been studied that further improves speed. The template word data is stored in a way that uses extra storage and lends itself to a table look-up technique for the

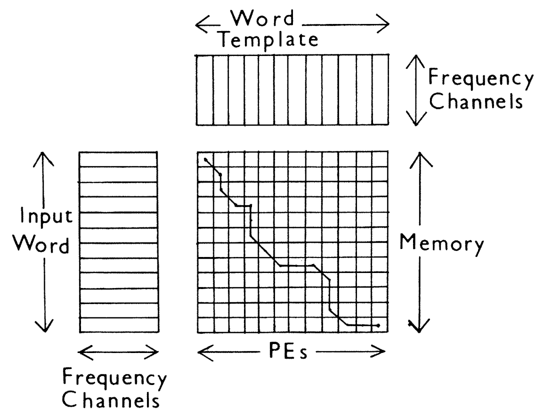


Fig. 4. Dynamic time warping.

Table 8
Some basic graphics operations

Operation	Speed for typical cases
Line drawing	8 Mpixels/s
Picture composition (Raster-Op)	500 Mbits/s (up to 3000 M)
Area filling	50 Mbits/s
3D Coordinate rotation (16 bit)	1 Mpoints/s (10% load for 2000 points @ 50 Hz)

differencing and squaring operations; a single input value is used as a global index to perform a matrix of table interpolations. For 6-bit data, this pushes performance up to about 50 words/s with a 100-word vocabulary (5 words/s for a 1000-word vocabulary, although the current MilDAP will be short of space).

3.6. Graphics

As an illustration of the potential of a programmable array, brief consideration is given to graphics, which can sometimes be an important function to combine with signal processing. MilDAP output can be via the fast I/O, which can be interfaced to a raster display, either via a frame store or else more directly. Performance varies with the parameters of particular cases, but (theoretical) figures for some basic graphics operations on MilDAP are given in table 8.

3.7. Some other applications

Some other applications are best dealt with mainly by way of references. Sorting has been mentioned already.

Very high-speed high-quality random number generation is used extensively in Monte Carlo calculations and is also sometimes useful in signal processing. Generation at speeds up to 800 million per second on a 64×64 DAP is described in ref. [13]; this speed is an order of magnitude faster than any other machine. The use of these random numbers in a very high-speed Monte Carlo simulation (the Ising model) is described in ref. [14], where speeds on a 64×64 DAP an order of magnitude faster than the best achieved on a CYBER 205 are reported. The use of DAP in Electronic Support Measure (ESM) is reported in

refs. [1], [3] and [15]. A promising application area is radar plot extraction, track formation and track combination; for some of this work refer to ref. [16].

Note added in proof

A new company, AMT, is producing a product, DAP-3, hosted on a microVAX or SUN workstation. DAP-3 has more memory and a faster cycle time (100 ns) than MilDAP/MiniDAP, and performance figures for this machine are included in some tables.

References

[1] S.F. Reddaway, DAP–a Distributed Array Processor, in: Proc. 1st Ann. Symp. on Computer Architecture, eds. G.J. Lipovski and S.A. Szygerda, IEEE Catalog No. 73CH0824-3C, Comput. Architect. News 2-4 (1973) 61–65.

[2] P.M. Flanders, D.J. Hunt, S.F. Reddaway and D. Parkinson, Efficient high speed computing with the distributed array processor, in: High Speed Computer and Algorithm Organisation, eds. D.J. Kuck, D.H. Lawrie and A.H. Sameh (Academic Press, New York, 1977) pp. 113–128.

[3] P. Simpson, J.B.G. Roberts and B.C. Merrifield, MilDAP: Its architecture and role as a real time airborne digital signal processor, in: Proc. AGARD Conf. on The Impact of VHPIC on Radar, Guidance and Avionics Systems, Lisbon, May 1985, pp. 23-1–23-17.

[4] S.F. Reddaway, J.B.G. Roberts, P. Simpson and B.C. Merrifield, Distributed array processors for military applications, in: MILCOMP85, London 1985 (Microwave Exhibitions and Publishers Ltd, Tunbridge Wells, 1985) pp. 74–82.

[5] MiniDAP: APAL Language, ICL Techn. Publ. R30014/02 (Literature and Software Operations, ICL, Reading, UK, 1986).

[6] MiniDAP: DAP FORTRAN Language, ICL Techn. Publ. R30011/02 (Literature and Software Operations, ICL, Reading, UK, 1986).

[7] P.M. Flanders and S.F. Reddaway, Sorting on DAP, in: Parallel Computing 83, eds. M. Feilmeir, G. Joubert and U. Schendel (North-Holland, Amsterdam, 1984) pp. 247–252.

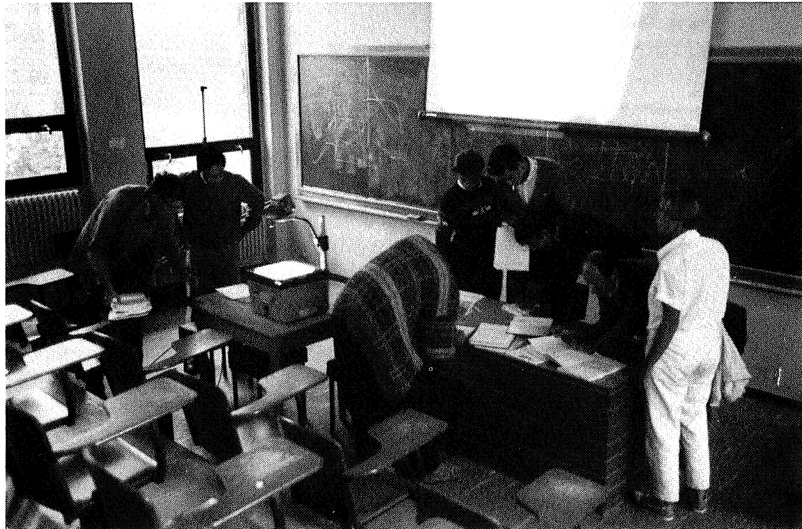
[8] S.F. Reddaway, Distributed array processor, architecture and performance, in: Nato ASI Series, Vol. F7, High-Speed Computation, ed. J.S. Kowalik (Springer, Berlin, 1984) pp. 89–98.

[9] S.F. Reddaway, A.L.G. Flanagan, D.J. Hunt, and J. Morris et al., unpublished work (1983).

[10] P.M. Flanders, A Unified Approach to a Class of Data Movements on an Array Processor, IEEE Trans. Comput. C 31-9 (1982) 809–819.

[11] S.F. Reddaway, DAP and its application to image processing tasks, paper 6.3 at the Image Processing Symposium RSRE Malvern, July 1983.

[12] P. Simpson and J.B.G. Roberts, Speech recognition on a distributed array processor, Electron. Lett. 19 (1983) 1018–1020.
[13] K.A. Smith, S.F. Reddaway and D.M. Scott, A very high performance pseudo-random number generation on DAP, Comput. Phys. Commun. 37 (1985) 239–244.
[14] S.F. Reddaway, D.M. Scott and K.A. Smith, A very high speed Monte Carlo simulation on DAP, VAPP II Conference, Oxford, August 1984, Comput. Phys. Commun. 37 (1985) 351–356.
[15] J.B.G. Roberts, P. Simpson, B.C. Merrifield and J.F. Cross, Signal processing applications of a distributed array processor, IEE. Proc. F 131 (1984) 603–609.
[16] R.W. Gostick and K.S. MacQueen, Radar tracking on the DAP, to be published.



PARALLEL COMPUTING 83
M. Feilmeier, J. Joubert and U. Schendel (eds.)
Elsevier Science Publishers B.V. (North-Holland)
© Int. Conf. "Parallel Computing 83", 1984

SORTING ON DAP

P.M.Flanders & S.F.Reddaway

INTERNATIONAL COMPUTERS LIMITED
Cavendish Road
Stevenage
UK

Sorting data is an important and time-consuming activity in many computer applications. This paper describes how a highly parallel processor, the ICL DAP, can be used effectively for both internal and external sorts.

INTRODUCTION

In many computer applications, both scientific and commercial, there is a requirement to sort data. Also sorting may be used to implement other operations on a parallel processor such as scatter-gather, permutations, conformal mappings etc [1]. In commercial applications the volume of data to be sorted frequently exceeds the size of main memory.

A variety of sorting algorithms have been developed, for the most part with sequential computers in mind. We consider here the application of a parallel computer, the ICL DAP [2], using an algorithm suitable for parallel processing, Batcher's bitonic sort [3]. Sorts for which all data can be accommodated in main memory (internal sorts) and those which require backing store (external sorts) are considered. Consideration is also given to the use of tag sorts, to improve performance and to handle variable length records. Examples of internal non-tag sorts of fixed length records have been implemented but the remainder of the work is theoretical. More details of topics covered in this paper are given in [4].

OUTLINE OF DAP

The following description summarises the main features of the DAP relevant to this paper; a general description is given in [2]. The DAP is an SIMD (Single Instruction stream, Multiple Data stream) computer the first model having 4096 processing elements (PEs) arranged 64*64. Each PE has a local store of a few thousand bits and obeys the same instruction simultaneously under the control of a master control unit (MCU). Instructions which access the store use the same address in each PE; PEs can however ignore certain instructions depending on the value of a locally held "activity control bit". In the principal mode of operation successive bits of a data item are held in the store of a single PE so that each PE produces one result. This is called "vertical mode processing" and has a parallelism of 4096. The PEs are bit organised with operations on multi-bit data implemented as sequences of bit level instructions. An alternative is "horizontal mode processing" where a data item is mapped so that a row of PEs is used to produce each result; results are therefore produced more quickly but the parallelism is reduced to 64. Data is routed between PEs either by shifting "bit-planes" (64*64 bits, one per PE) in any of the four directions or by highways which traverse the rows and columns of the PE matrix. For the former, the time taken increases with the distance shifted.

BATCHER'S BITONIC SORT

Figure 1 shows the comparisons and exchanges of data in a sample sort of 8 records using Batcher's bitonic sort; a brief general description is given below for the more interested reader. In figure 1 data is moved as indicated at each step and compared with the data already there; the larger or smaller element, indicated by "+" or "-", is then selected.

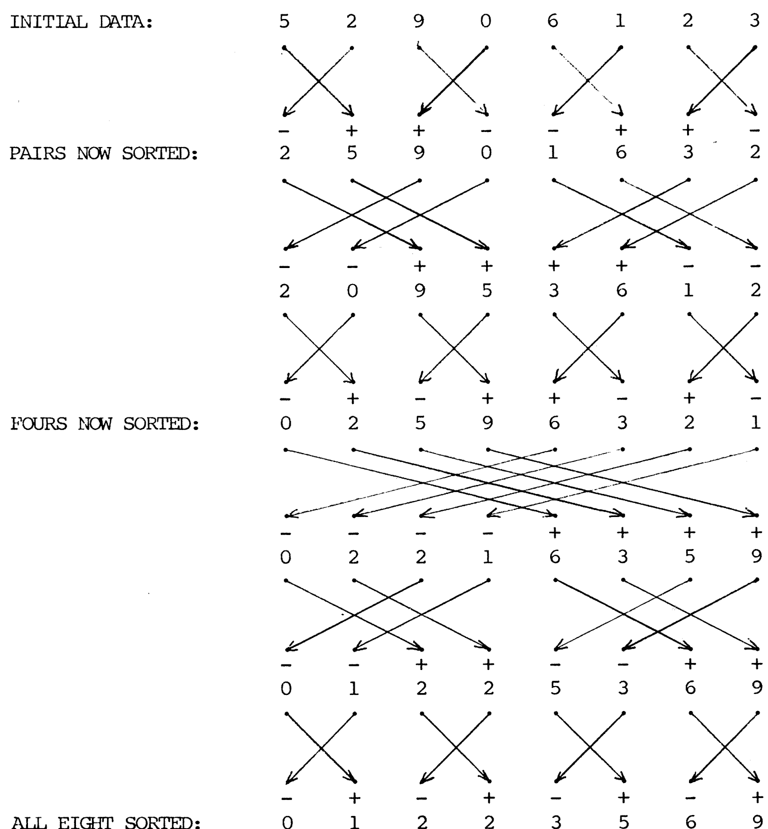


Figure 1
Sample Bitonic Sort of 8 Records

Batcher's bitonic sort is based on the idea of a bitonic sequence; a sequence is bitonic if, when considered cyclicly, it has one ascending and one descending portion. Suppose a bitonic sequence of records, A_i where i = 1,2, ... ,N is split into the two sequences A1 and A2 such that:

$$A1_i = \min(A_i, A_{i+N/2})$$

and

$$A2_i = \max(A_i, A_{i+N/2})$$

where $1 \leq i \leq N/2$ and records are compared using the record keys, then it can be shown that A1 and A2 are also bitonic sequences and all records of A2 are greater than or equal to all records of A1. Thus if a bitonic sequence of N records is split into two sequences in this way and the resulting sequences are again split and so on, the result is ordered after log N steps (All logarithms here are base 2). This enables a bitonic sequence to be ordered by successive splitting; the ordering may be increasing or decreasing.

To sort an arbitrary set of N records the algorithm to order a bitonic sequence is used as follows. First consider the N records as N/2 sequences each containing two records. Each sequence is trivially bitonic and is ordered as above; half the N/2 sequences are put into ascending order and half into descending order. The N/2 sequences are then taken in pairs, one ascending and one descending sequence in each pair, so that each pair is a bitonic sequence of four records. This gives N/4 bitonic sequences each having 4 records; the number of sequences has been halved and the number of records in each sequence has been doubled. Repeating this procedure log N times leaves the data sorted.

USE OF DAP FOR INTERNAL SORTS

At each step of the algorithm to sort N records there are N/2 "comparison-exchanges" all of which can be done in parallel. Each of these involves comparing two records and conditionally exchanging them depending on the result of the comparison. In a complete sort there are $\frac{1}{4}(N \log N (1 + \log N))$ comparison-exchanges, that is about $\frac{1}{4}\log N$ more basic operations than in a conventional serial sort. However the Batcher algorithm has the advantage that it is readily implemented in parallel on the DAP with each PE performing one comparison-exchange, for vertical mode processing. For sorts where the number of records is at least twice the number of PEs (or twice the number of rows for horizontal mode processing) the parallelism of DAP can be utilized to the full.

On DAP the conditional exchanges are executed in parallel using activity control; this may be contrasted with implementation on a serial machine using conditional jumps. The time taken on DAP is independent of the initial ordering of data.

Before each stage of comparison-exchanges can be executed, records must be paired up in the same PEs. In general this requires data to be routed amongst the PEs and this is done using parallel techniques which shift and merge bit-planes. Unless care is taken this routing can dominate the execution time. Much attention has been given to this in the literature; however most published algorithms concentrate on the special case where the number of records equals the number of PEs [5,6]. The more general problem where there are more records than PEs has been studied by Baudet and Stevenson [7] but they use an algorithm involving more comparisons than Batcher's bitonic sort in order to reduce routing; they also assume independent addressing in each PE is possible.

The algorithm for DAP for the general case where there are more records than PEs is described more fully in [4]; it achieves high performance by:

(a) ensuring that all PEs are active and perform distinct comparison-exchanges;
(b) taking into account, in the mapping of data on to the store, the relative frequency of comparison-exchanges at different separations;
(c) allowing the mapping of data to vary during the sort (This almost halves the routing required for an optimal static mapping. It leaves the data in an unnatural order but the overhead to restore natural ordering is small).

With reference to (b), the data is sorted first within PEs, then over the PE plane starting with nearest neighbours in both directions and moving on to the

longer routes. The resulting complexity in data organisation and movement is best handled using the "mapping vector" techniques described in [8,9]. These provide a systematic and readily implemented means of data organisation.

The time taken to sort N records has a contribution from the comparison-exchanges which is proportional to N(log N) and a contribution from routing proportional to N. For practical values of N on DAP the contribution from these is of the same order. An estimate of performance is given below for a vertical mode sort of 128,000 records each 8 bytes long; all 8 bytes are used in comparing records. For the sorts implemented on DAP, actual times were typically 10% greater than estimated. Rates of processing data are given in megabytes/sec to allow easy comparison with disc transfer rates. Figures are given for a DAP having 32*32 PEs; performance on the smaller DAP does not fall in line with the reduction in the number of PEs since routing is less dominant.

	64*64 DAP	32*32 DAP
Comparison-exchanges	536 cycles/bit-plane	536 cycles/bit-plane
Routing	743 cycles/bit-plane	424 cycles/bit-plane
Number of bit-planes	2048	8192
Total machine cycles	$2.6x10^6$	$7.2x10^6$
Total time at 200nsec/cycle	0.53secs	1.44secs
Rate of processing data	1.9Mbyte/sec	0.7Mbyte/sec

TAG SORTS AND VARIABLE LENGTH RECORDS

If the sort key is a small part of the record, better performance is achieved by using a "tag sort". Tags comprising the record key and record address are formed for each record and these are sorted instead of the records themselves. The sorted tags are then used to order the complete records, thereby moving each record once only. This improves performance of the sorting process by a factor roughly equal to the ratio of the record length to the tag length; it does however incur the additional costs of tag formation and record movement.

Moving records according to the sorted tags makes effective use of the abilitity of the DAP to move whole bit-planes of data at a time. The shifting required to produce a new alignment for the moved record is noticeably more with larger DAPs; also the larger number of bits in a bit-plane is only useful for records larger than 128 bytes and so the smaller DAP may actually be faster. Performance for tag sorts is in many cases dominated by record movement and to a lesser extent tag formation.

Tag sorts give an effective means of sorting variable length records. It is assumed that each record contains a byte count for the record and that the key fields are fixed length and at fixed positions relative to the start of record. Despite extra work in forming tags and moving records, DAP is still effective.

Estimates are given below for a horizontal mode tag sort of 32,000 100-byte records having 4-byte keys; figures are given for fixed and variable length records, with 100 bytes being the average length in the latter case. Unless stated otherwise figures are in cycles per bit-plane of record data; the cycle time is 200ns.

Fixed Length Records:

	64*64 DAP	32*32 DAP
Tag formation:	70	21
Sort tags:	120	78
Move records:	260	29
Total:	450	128
Rate of processing data:	5.7 Mbytes/sec	5.0 Mbytes/sec

Variable Length Records:

	64*64 DAP	32*32 DAP
Tag formation:	200	43
Sort tags:	120	78
Move records:	410	74
Total:	730	195
Rate of processing data:	3.5 Mbytes/sec	3.3 Mbytes/sec

EXTERNAL SORTS

Sorting an amount of data too large to be held in the DAP store is done in two parts, a prestring stage followed by one or more merge passes. The prestring stage forms sorted strings, each of as many records as can be sorted in memory, and is implemented as a sequence of internal sorts. The merge passes combine these strings to form progressively fewer sorted strings until just one string is obtained. The algorithm for a two-way merge pass is similar to that for a conventional two-way merge except it merges blocks using a bitonic merge rather than comparing single records. At each step the next block of records is taken from one of the strings (for ascending order, the one whose next block contains the smallest key) and merged with the "current block"; the half containing the smallest keys is then output and the remainder left in store as the next "current block".

To prove that this algorithm works it is sufficient to show that after each merge the records output from the merge area all have keys less than or equal to those of all records in the input strings (for sorting in ascending order). Clearly this is true for the string from which the last block was selected. We denote this string by S1, the block selected from it by B1 and the smallest key in that block by K1; we denote the the other string by S2, its next block by B2 and the smallest key in B2 by K2. Before block B1 was selected for merging, the records left in the merge area from the previous merge must have all been less than or equal to the larger of K1 and K2; however since B1 was chosen in preference to B2 we know that $K1 \leqslant K2$ and consequently all these records must have keys less than or equal to K2. The result follows immediately.

The merging of two blocks, sorted into ascending order, can be achieved by reversing the order of one of the blocks so that together they form a bitonic sequence. In practice it is not necessary to explicitly reverse the order since the bitonic merge can easily be modified to give the first half of its result in ascending order and the second half in descending order; when the first half is output it is replaced with another block in ascending order so that the two halves to be merged are in opposite order.

The above describes a two-way merge; a number of strings can be merged by forming a binary tree of two-way merges within the DAP memory. The output of each merge at a given level of the tree is input to the next level. Merging a number of strings not equal to a power of two can be done by placing the input from some strings higher up the tree.

For an m-way merge a block of N records is output after each stage of log m two-way merges. The work per two-way merge is approximately proportional to N(1+log N) which implies that to achieve high performance the block size must be as small as possible whilst allowing effective use of DAP parallelism. A smaller block size also helps reduce storage requirements. We therefore assume that on the 64*64 DAP,horizontal mode processing is used with a block size, N, equal to 64. Merge performance varies somewhat with the record parameters, but a typical performance for a 32-way non-tag merge is as follows:

Example of Merge Performance (32-way merge):

	64*64 DAP	32*32 DAP
Block size(records):	64	32
Number of steps in block merge:	7	6
Cycles per merge per bit-plane output:	320	220
Total cycles per bit-plane output:	1600	1100
Rate of processing data(Mbyte/sec):	1.6	0.58

Tags can be used to handle variable length records and, where appropriate, to achieve performance improvements similar to internal sorts. A tag is formed for each record on input and the tags are merged rather than the complete records, which are left in input buffers. As tags emerge from the merge tree the corresponding records are moved to the output buffers.

With the high speeds possible for both prestring and merge stages, disc transfer rates may be the limiting factor in overall performance. The availability of a large DAP memory helps in allowing large disc block sizes which maximise transfer rates.

CONCLUSIONS

The DAP can be used for high performance sorting on both internal and external sorts. Tag sorts can be used to improve performance where relevant and to handle variable length records. Ancillary operations of tag formation and record movement can be effectively implemented on DAP hardware. The performance of smaller DAPs is much better than would be expected from comparing array sizes.

REFERENCES

[1] Schwartz, J.T., Ultracomputers, ACM Transactions on Programming Languages and Systems, Vol. 2 No. 4 (Oct 1980).

[2] Flanders P.M., Hunt D.J., Parkinson D., Reddaway S.F., Efficient High Speed Computing with the Distributed Array Processor, Symposium on High Speed Computer and Algorithm Organisation, Univ. Illinois (Academic Press, 1977).

[3] Batcher K.E., Sorting Networks and their Applications, Proc. AFIPS 1968 SJCC, Vol. 32, Montvale NJ (AFIPS press) 307-314.

[4] Flanders P.M., Reddaway S.F., Sorting on DAP, SSC Report CM72, International Computers Ltd., Stevenage UK (Oct. 1982).

[5] Nassimi D., Sahni S., Bitonic Sort on a Mesh-connected Parallel Computer, IEEE Trans. Comput., Vol. C-28 No. 1, (Jan. 1979).

[6] Thompson C.D., Kung H.T., Sorting on a Mesh-connected Parallel Computer, CACM, Vol. 20 (April 1977).

[7] Baudet G., Stevenson D., Optimal Sorting Algorithms for Parallel Computers, IEEE Trans. Comput., Vol. C-27 No. 1 (Jan. 1978).

[8] Flanders P.M., A Unified Approach to a Class of Data Movements on an Array Processor, IEEE Trans. Comput., Vol. C-31 No. 9 (Sept. 1982).

[9] Flanders P.M., Languages and Techniques for Parallel Array Processing, Ph.D. Thesis, Dept. of Comp. Sc., Queen Mary College, Univ. of London (July 1982).

The Solution of Linear Complementarity Problems on an Array Processor

C. W. CRYER

Department of Applied Mnthematics and Theoretical Physics, University of Cambridge, Cambridge CB3 9EW, England

P. M. FLANDERS, D. J. HUNT, AND S. F. REDDAWAY

Research and Advanced Development Centre, International Computers Limited, Stevenage, Hertfordshire SGI 2BD, England

AND

J. STANSBURY

Computer Sciences Department University of Wisconsin-Madison, Madison, Wisconsin 53706

Reprinted from JOURNAL OF COMPUTATIONAL PHYSICS
Vol. 47, No. 2, August 1982
Printed in Belgium
All Rights Reserved by Academic Press, New York and London

Reprinted from JOURNAL OF COMPUTATIONAL PHYSICS
Vol. 47, No. 2, August 1982
Printed in Belgium
All Rights Reserved by Academic Press, New York and London

The Solution of Linear Complementarity Problems on an Array Processor

C. W. CRYER*[†]

Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge CB3 9EW, England

P. M. FLANDERS, D. J. HUNT, AND S. F. REDDAWAY

Research and Advanced Development Centre, International Computers Limited, Stevenage, Hertfordshire SG1 2BD, England

AND

J. STANSBURY*

Computer Sciences Department, University of Wisconsin–Madison, Madison, Wisconsin 53706

Received January 29, 1980; revised April 13, 1982

The Distributed Array Processor (DAP) manufactured by International Computers Limited is an array of 1-bit 200-nanosecond processors. The Pilot DAP on which the present work was done is a 32×32 array; the commercially available machine is a 64×64 array. We show how the projected SOR algorithm for the linear complementarity problem $Aw \geqslant b$, $w \geqslant 0$, $w^{\mathrm{T}}(Aw - b) = 0$, can be adapted for use on the DAP when A is the *finite-difference* matrix corresponding to the difference approximation to the Laplace operator. Application is made to two linear complementarity problems arising, respectively, from two- and three-dimensional porous flow free boundary problems.

1. INTRODUCTION

An LCP (*linear complementarity problem*) is a problem of the form: Find an n-vector $w = (w_i)$ satisfying

$$Aw \geqslant b, \tag{1.1a}$$

$$w \geqslant 0, \tag{1.1b}$$

$$w^{\mathrm{T}}(Aw - b) = 0, \tag{1.1c}$$

where $b = (b_i)$ is a known real n-vector and $A = (a_{ij})$ is a known real $n \times n$ matrix.

* Sponsored by the National Science Foundation under Grant No. MCS77-26732.
[†] Sponsored by the United States Army under Contract No. DAAG29-80-C-0041.

0021-9991/82/080258–23$02.00/0
Copyright © 1982 by Academic Press, Inc.
All rights of reproduction in any form reserved.

Linear complementarity problems arise in many contexts (Balinski and Cottle [2]). In particular, there is a connection between linear complementarity problems and variational inequalities (Cottle, Giannessi, and Lions [5], Cryer and Dempster [9]).

Many problems in continuum mechanics can be reformulated as variational inequalities (Duvaut and Lions [10], Kinderlehrer and Stampacchia [20]), which, when discretized, reduce to linear complementarity problems of the form (1.1) with special features.

(1) A is large matrix, perhaps of order 25,000.

(2) A is a *finite-difference* or *finite-element* matrix; in particular, A is sparse with a great deal of structure. (1.2)

(3) A large percentage of the elements of the solution w are nonzero.

Because of these special features, the standard methods of solving linear complementarity problems are not very efficient, and methods of solution have been developed which take advantage of the structure of A: projected SOR (Cryer [7], Glowinski [14]); modified block SOR (Cottle, Golub, and Sacher [6]); multigrid projection (Brandt and Cryer [3]); and generalizations of projected SOR (Mangasarian [21]). Cryer [8] briefly surveys much of this work.

In the present paper we consider the use of the parallel computer DAP to solve linear complementarity problems with the features (1.2). The DAP (Distributed Array Processor, manufactured by International Computers Limited), which is an SIMD array of typically 64×64 processors, is described in Section 2. In Section 3 we describe the implementation on the DAP of projected SOR to solve a linear complementarity problem derived from a two-dimensional porous flow free boundary problem, and in Section 4 we extend this work and solve a linear complementarity problem derived from a three-dimensional porous flow free boundary problem. In Section 5 we comment on possible future developments, and the overall conclusions are in Section 6.

2. The Pilot DAP (Distributed Array Processor)

The present work was carried out on the Pilot 32×32 DAP at Stevenage, England, and we shall describe this machine first. A 64×64 version is available, and the minor differences between the two machines are indicated at the end of this section.

DAP Hardware

The essential features of the Pilot DAP hardware are as follows (Flanders *et al.* [12], Reddaway [23]):

(1) A 32×32 array of identical processing elements (PEs) with a cycle time of 200 nanoseconds.

(2) Each PE has a one-bit adder, 2K bits of storage, and three one-bit registers (a general purpose register for accessing data and performing arithmetic; a carry register; and an activity control register).

(3) Each PE is connected to its four neighboring PEs (North, South, East, and West). In a given cycle all PEs access their neighbor in the same direction (determined by the program). In addition, the PEs are linked by row and column highways which connect together all the PEs in each row and column.

(4) There is a master control unit (MCU) which broadcasts instructions to all the PEs. All PEs can perform the same instruction simultaneously, but certain instructions are only effective if the activity control register is *true*.

DAP Software

A program to run on a DAP system normally comprises a standard FORTRAN program and a number of subroutines and functions written in an array processing extension of FORTRAN known as DAP-FORTRAN (Flanders [11], Gostick [15], ICL [19]). The standard FORTRAN is executed by the host computer and provides mainly input-output and overall control. The DAP-FORTRAN is executed by the DAP and provides high speed computation. Data is shared between them using common blocks held in DAP store. Some features of DAP-FORTRAN are described below.

In addition to the data types of FORTRAN, DAP-FORTRAN has two new data types: *vector* and *matrix*. With a 32×32 DAP, a vector has 32 components and a matrix has 32×32 components; the components can be real, integer, or logical.

For example, the data statements

$$\begin{aligned} &\text{REAL } U(\ ), V(\ ,\ ), W(\ ,5), X(\ ,\ ,3) \\ &\text{INTEGER } A(\ ,1), B(\ ), C(\ ,\ ,4) \\ &\text{LOGICAL } FLAGS(\ ,2), MASK(\ ,\ ) \end{aligned} \tag{2.1}$$

declare U (a real vector), V (a real matrix), W (an array of five real vectors), X (an array of three real matrices), A (an array of one integer vector), B (an integer vector), C (an array of four integer matrices), $FLAGS$ (an array of two logical vectors), and $MASK$ (a logical matrix).

Expressions in DAP-FORTRAN consist of scalars, vectors, and matrices with the usual unary and binary operations. Operations on vectors and matrices are performed in parallel using all 32×32 PEs.

Operations between a scalar and a vector or a matrix cause implicit expansion of the scalar to the necessary dimensions. For example, if M is a matrix of size 32×32 and S is a scalar, then $M = M + S$ causes S to be implicitly expanded to size 32×32 with each element being equal to S; then the corresponding elements of "matrix" S and matrix M are added in parallel and assigned to M in parallel.

Arrays of vectors and matrices may be used to construct more complex structures . To process a vector or matrix array requires performing calculations on the individual vectors or matrices in the array.

Selection and updating of parts of vectors and matrices can be performed using the powerful indexing capabilities of DAP-FORTRAN. Matrix sections can be specified by omitting subscripts along which all elements are to be taken. Using this, whole rows or columns can be selected from matrices. For example, $M(I,\ )$ specifies the Ith row of matrix M.

Shift indexing is a very useful feature of DAP-FORTRAN. For example, in a simple solution of Laplace's equation on a 32×32 grid we wish to replace each element with the average of its four neighbors. This could be coded in FORTRAN as:

```
DO 10 I = 2, 31
DO 10 J = 2, 31
Y(I, J) = (X(I + 1, J) + X(I - 1, J) + X(I, J + 1) + X(I, J - 1))/4.0
10 CONTINUE
```

Further code would be needed to handle elements on the edges of the matrix.

The DAP-FORTRAN code is much simpler

$$X = (X(+,\ ) + X(-,\ ) + X(\ ,+) + X(\ ,-))/4.0. \tag{2.1}$$

The term $X(+,\ )$ uses shift indexing. In particular, $X(+,\ )$ specifies a matrix where the (I, J) element is the $(I + 1, J)$ element of X, for $1 \leqslant I \leqslant 32$ and $1 \leqslant J \leqslant 32$. Thus, $X(+,\ )$ contains all the *south* neighbors of X. Edge values (corresponding to subscripts 0 or 33) are defined to be zero. As an alternative, cyclic geometry may be specified by using a GEOMETRY statement.

Longer shifts can be performed by explicit system functions; for example, $\mathrm{SHS}(X, I)$ shifts the matrix X to the south I positions. Note that since all the updating is performed simultaneously, it is not necessary to write the results to another matrix.

Logical matrices and vectors can be used to select elements from an array. For example, if we wished to update only certain elements of X in statement (2.1), we could set the corresponding elements of LM, a logical matrix, to true and all other elements of LM to false. That is, if $X(I, J)$ is to contain the average of its four neighbors, then $LM(I, J)$ is set to true. Otherwise, $LM(I, J)$ is false. Then the following statement performs the required task:

$$X(LM) = (X(+,\ ) + X(-,\ ) + X(\ ,+) + X(\ ,-))/4.0.$$

DAP-FORTRAN has a number of useful system functions whose arguments and results may be scalars, vectors, or matrices. The ALTC, ALTR, MERGE, MAX, and ABS functions will be briefly described since these are used in the programs in this paper.

The functions ALTC and ALTR return logical matrices. If C is the argument to ALTC, then the first C columns of the result matrix are set to false, the next C columns to true, the next C columns to false, etc. ALTR performs similarly for rows.

The function MAX (now named MAXV) returns a scalar equal to the largest number in its vector or matrix argument. The function ABS returns a vector or matrix containing the absolute value of every element in its argument.

The function MERGE takes three arguments and returns a matrix. The first two arguments are matrices (or implicitly expanded scalars) and the third argument is a logical matrix. If the (I, J) element of the logical matrix is true then the (I, J) element of the result matrix is set equal to the (I, J) element of the first matrix; otherwise, it is set equal to the (I, J) element of the second matrix.

Examples of DAP-FORTRAN programs are given in Sections 3 and 4.

DAP Arithmetic

When a DAP-FORTRAN program is executed by the DAP, expressions involving only scalars are executed sequentially, but operations on vectors and matrices are performed in parallel by the PEs.

The DAP memory can be visualized as a cuboid, with 2 K horizontal planes, each plane being a 32×32 square of bits. The 32×32 array of PEs lies on top of the cube, and each column of 2 K bits belongs to the PE above it.

Two storage modes are used in DAP-FORTRAN, vertical and horizontal. Scalars and vectors are stored in horizontal mode while matrices are held in vertical mode.

In vertical mode, each number is held entirely within the store of one PE with successive bits in successive store locations. Thus, for an integer matrix, the sign bit of every element in the matrix would be held in the same store address of each PE.

In horizontal mode, a number is spread along a row of PEs. Thus, a scalar occupies one row while a vector occupies 32 rows. DAP instructions are also stored in this format.

All arithmetic is carried out using subroutines. Some operation times for 32 bit numbers are given in Table I.

TABLE I

Average DAP-FORTRAN Arithmetic Times for the Pilot DAP

Operation	Matrix	Vector	Scalar
Floating point addition	140–180 μs	54 μs	27 μs
Floating point multiplication	315 μs	50 μs	34 μs
Floating point multiplication by a scalar	60–200 μs	40 μs	–
One shift of a real matrix, e.g., $X(+,)$	15 μs	2 μs	–
Move a floating point matrix	15 μs	2 μs	2 μs
Logical AND	2 μs	2 μs	2 μs
Logical mask	1 μs	2 μs	–

Note. Times are slightly different on production DAPs.

It will be noted that vector arithmetic is faster than matrix arithmetic. This is because a row of PEs is available for each vector component, while only one PE is available for each matrix component.

Some of the quoted computation times are data dependent. In particular, matrix multiplication by a scalar typically varies from 170 μs to 200 μs depending upon the distribution of zeros in the binary representation of the constant; for special scalars such as 0.5 or 3 the multiplication time can be as low as 60 μs.

Host-DAP Interface

The sequence of operations for compiling and running DAP programs is as follows:

(a) The host computer compiles the host FORTRAN program and the DAP-FORTRAN subroutines into host and DAP machine codes respectively.

(b) DAP machine code, incorporating all necessary low level subroutines, is loaded into DAP memory in horizontal mode where it occupies a few bits of each PE's memory. Host machine code is loaded into the host memory.

(c) Execution begins in the host and control is transferred to the DAP as required by subroutine calls. On completion of DAP processing, the host resumes execution at the point following the call.

Detailed information on the Pilot DAP relevant to understanding the programs in this paper is given in the Appendix.

The Production DAP

The current production DAP is generally similar to the Pilot but differs as follows:

(a) there are 4096 PEs arranged in a 64×64 array;

(b) each PE has 4 K bits of memory;

(c) arithmetic operations differ somewhat in timing but are overall a little faster;

(d) coupling between host and DAP is more direct so the interface is simpler than indicated in the Appendix.

3. Numerical Solution of a Two-Dimensional Free Boundary Problem

The flow of water through a porous dam is a well-known model problem. Water seeps from a reservoir of height H through a rectangular dam of width L to a reservoir of height h. Part of the dam is saturated and the remainder of the dam is dry. The wet and dry regions are separated by an unknown free boundary Γ which must be found as part of the solution (Fig. 3.1).

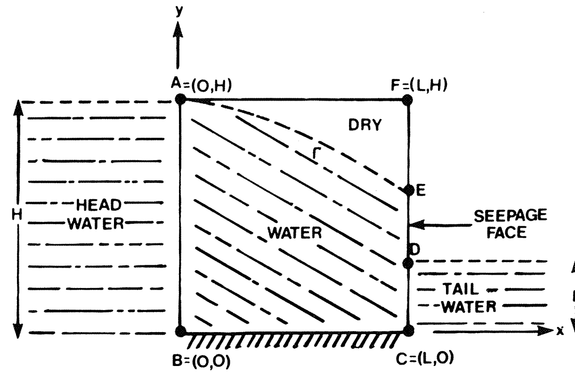


FIG. 3.1. Flow through a porous rectangular dam R.

As shown by Baiocchi [1], the problem can be formulated as follows: Find u on the rectangle $R = ABCF$ such that

$$-\nabla^2 u \geqslant -1, \quad \text{on} \quad R, \tag{3.1a}$$

$$u \geqslant 0 \quad \text{on} \quad R, \tag{3.1b}$$

$$u(-\nabla^2 u + 1) = 0 \quad \text{on} \quad R; \tag{3.1c}$$

and

$$\begin{aligned} u = g &= (H-y)^2/2, && \text{on} \quad AB, \\ &= (h-y)^2/2, && \text{on} \quad CD, \\ &= [H^2(L-x) + h^2 x]/2L, && \text{on} \quad BC, \\ &= 0, && \text{on} \quad DFA. \end{aligned} \tag{3.2}$$

The wet region of the dam consists of the points where $u > 0$ and the dry region consists of the points where $u = 0$.

When the problem (3.1), (3.2) is approximated using the classical five point difference approximation for the Laplace operator, one obtains an LCP of the form (1.1), where the matrix A and right hand side b are the same as those that would be obtained if the Dirichlet problem

$$\begin{aligned} -\nabla^2 u &= -1 && \text{on} \quad R, \\ u &= g && \text{on} \quad \partial R \end{aligned} \tag{3.3}$$

were approximated by the finite difference equation $Aw = b$. More precisely, let an $M \times N$ grid with gridlength Δx be superimposed upon R, and let the values of u and g at the point $([j-1]\,\Delta x, [i-1]\,\Delta x)$ be denoted by u_{ij} and g_{ij}, respectively, for $1 \leqslant i \leqslant M$ and $1 \leqslant j \leqslant N$. Then (1.1) takes the form

$$4w_{ij} - w_{i+1,j} - w_{i-1,j} - w_{i,j+1} - w_{i,j-1} \geqslant -(\Delta x)^2 \quad \text{for} \quad 1 < i < M, \quad 1 < j < N, \tag{3.4a}$$

$$w_{ij} \geqslant 0 \quad \text{for} \quad 1 < i < M, \quad 1 < j < N, \tag{3.4b}$$

$$w_{ij}(4w_{ij} - w_{i+1,j} - w_{i-1,j} - w_{i,j+1} - w_{i,j-1} + (\Delta x)^2) = 0 \quad \text{for} \quad 1 < i < M, \quad 1 < j < N, \tag{3.4c}$$

$$w_{ij} = g_{ij} \quad \text{for} \quad ((j-1)\Delta x, (i-1)\Delta x) \in \partial R. \tag{3.4d}$$

We discuss below two iterative methods for solving (3.4), the projected Jacobi method and the projected SOR method. The projected Jacobi method is much slower than the projected SOR method, but is trivial to implement on the DAP and serves as a useful introduction to DAP programming.

TABLE II
The DAP Subroutine JACOBI

Code	Comment
SUBROUTINE JACOBI	
LOGICAL *MASK*(,), *WSIGN*(,)	Declare logical 32×32 matrices, *MASK* and *WSIGN*
REAL *W*(,), *Z*(,)	Declare real floating point 32×32 matrices *W* and *Z*
REAL *INDEX*( )	Declare a real floating point 32-vector *INDEX*.
EQUIVALENCE(*W*, *WSIGN*)	Declare the logical matrix *WSIGN* equivalent to the first bit, the sign bit, of the matrix *W*.
HEIGHT = 31.0	
WIDTH = 31.0	
DO 10 *I* = 1, 32	Initialize *INDEX* vector.
INDEX(*I*) = (32 − *I*)/31.0	
10 **CONTINUE**	
W = 0	Clear matrix *W*
TEMP = *HEIGHT* * *HEIGHT* * .5	
W(1,) = *TEMP* * *INDEX*	Set values of the matrix *W* equal to *g* on bottom (*BC*).
W(, 1) = *TEMP* * *INDEX* * *INDEX*	Set values of the matrix *W* equal to *g* on left (*AB*).
MASK = .TRUE.	
MASK(1,) = .FALSE.	
MASK(32,) = .FALSE.	Set the matrix *MASK* to be true at interior points and false at boundary points.
MASK(, 1) = .FALSE.	
MASK(, 32) = .FALSE.	
DO 50 *I* = 1, 100	Start of main loop
1 *Z* = *W*(+,) + *W*(−,) + *W*(, +) + *W*(, −) − 1.0	Sum neighbors and store in *Z* matrix.
2 *W*(*MASK*) = .25 * *Z*	Transfer average to *W* at interior points.
3 *W*(*MASK* .AND. *WSIGN*) = 0.0	Project by setting $W = 0$ at points where *MASK* is true and the sign of *W* is negative.
50 **CONTINUE**	
END	

TABLE III

Statement	Operations	Time (μs)
$Z = W(+,\ ) + W(-,\ ) + W(\ ,+)$ $+ W(\ ,-) - 1.0$	4 floating point matrix additions/subtractions	640
	4 index shifts	60
	1 scalar-matrix assignment	15
$W(MASK) = .25 * Z$	1 floating point matrix multiplication by a special constant	70
	1 logical mask	1
$W(MASK$ .AND. $WSIGN) = 0.0$	1 logical AND	2
	1 logical mask	1
	1 scalar-matrix assignment	15
DO 50 $I = 1, 100$		7
		811

The Projected Jacobi Method

Let $w^{(0)} = (w_{ij}^{(0)})$ be an initial guess for the solution $w = (w_{ij})$ of (3.4). One generates a sequence of approximations $w^{(k)} = (w_{ij}^{(k)})$, $k = 1, 2, \ldots,$

$$z_{ij}^{(k)} = w_{i-1,j}^{(k)} + w_{i+1,j}^{(k)} + w_{i,j-1}^{(k)} + w_{i,j+1}^{(k)} - (\Delta x)^2, \tag{3.5a}$$

$$w_{ij}^{(k+1/2)} = \tfrac{1}{4} z_{ij}^{(k)}, \tag{3.5b}$$

$$w_{ij}^{(k+1)} = \max(0, w_{ij}^{(k+1/2)}), \qquad \text{for} \quad 1 < i < M \text{ and } 1 < j < N; \tag{3.5c}$$

$$w_{ij}^{(k+1)} = g_{ij}, \qquad \text{for} \quad ((j-1)\Delta x, (i-1)\Delta x) \in \partial R. \tag{3.5d}$$

It is known that the projected Jacobi method will converge (Mangasarian [21]).

If $M \leqslant 32$ and $N \leqslant 32$, then the gridpoints can be regarded as a subset of a 32×32 array, and one PE can be associated with each gridpoint. Defining $w^{(k)}$, $w^{(k+1)}$, and $z^{(k)}$ as real DAP-FORTRAN matrices, the computation (3.5) is trivial to implement on the DAP.

In Table II we list a DAP subroutine JACOBI which solves the dam problem for the case $h = 0$, $H = 31$, $L = 31$, $M = N = 32$, and $\Delta x = 1$. This subroutine could be called by a host program, which could then print the answers in the matrix W.

Using the operation times given in Table I, we can readily estimate the time required per iteration in the main loop of the JACOBI subroutine (see Table III).

From Table III we see that one projected Jacobi iteration over the whole 32×32 grid requires 811 μs.

The Projected SOR Method

Let $w^{(0)} = (w_{ij}^{(0)})$ be an initial guess for the solution $w = (w_{ij})$ of (3.4). In the usual implementation of projected SOR, one generates a sequence of approximations $w^{(k)} = w_{ij}^{(k)}$ as follows:

$$z_{ij}^{(k)} = w_{i-1,j}^{(k+1)} + w_{i+1,j}^{(k)} + w_{i,j-1}^{(k+1)} + w_{i,j+1}^{(k)} - (\Delta x)^2, \tag{3.6a}$$

$$\begin{aligned} w_{ij}^{(k+1/2)} &= w_{ij}^{(k)} + \omega(z_{ij}^{(k)} - 4w_{ij}^{(k)})/4 \\ &= (\omega/4)\, z_{ij}^{(k)} + (1-\omega)\, w_{ij}^{(k)}, \end{aligned} \tag{3.6b}$$

$$w_{ij}^{(k+1)} = \max\{0, w_{ij}^{(k+1/2)}\}, \qquad \text{for} \quad 1 < i < M \text{ and } 1 < j < N, \tag{3.6c}$$

where ω is a constant, the over-relaxation parameter.

It is known that the iteration (3.6) converges for all initial guesses $w^{(0)}$ iff $0 < \omega < 2$ (Cryer [7], Glowinski [14]). The implementation (3.6) is not suitable for parallel computation because the new values $w^{(k+1)}$ cannot be computed simultaneously; $w_{i-1,j}^{(k+1)}$ and $w_{i,j-1}^{(k+1)}$ must be known before $w_{ij}^{(k+1)}$ can be computed.

There is, however, a simple but ingenious way of making SOR suitable for parallel computation. In the implementation (3.6), we order the gridpoints by rows and columns (Fig. 3.2a). Instead, let us visualize the gridpoints as forming a red-black chess board and number first the red points and then the black points (Fig. 3.2b).

Applying projected SOR to the points numbered as in Fig. 3.2b we find that each projected SOR iteration can be broken down into two stages: in the red (first) stage projected SOR is applied to the red points; and in the black (second) stage projected SOR is applied to the black points.

Red stage.

$$z_{ij}^{(k,\text{red})} = w_{i+1,j}^{(k,\text{black})} + w_{i-1,j}^{(k,\text{black})} + w_{i,j+1}^{(k,\text{black})} + w_{i,j-1}^{(k,\text{black})} - (\Delta x)^2, \tag{3.7a}$$

$$w_{i,j}^{(k+1/2,\text{red})} = (\omega/4)\, z_{ij}^{(k,\text{red})} + (1-\omega)\, w_{ij}^{(k,\text{red})}, \tag{3.7b}$$

$$w_{i,j}^{(k+1,\text{red})} = \max\{0, w_{i,j}^{(k+1/2,\text{red})}\}. \tag{3.7c}$$

Black stage.

$$z_{ij}^{(k,\text{black})} = w_{i+1,j}^{(k+1,\text{red})} + w_{i-1,j}^{(k+1,\text{red})} + w_{i,j+1}^{(k+1,\text{red})} + w_{i,j-1}^{(k+1,\text{red})} - (\Delta x)^2, \tag{3.8a}$$

$$w_{ij}^{(k+1/2,\text{black})} = (\omega/4)\, z_{ij}^{(k,\text{black})} + (1-\omega)\, w_{ij}^{(k,\text{black})}, \tag{3.8b}$$

$$w_{ij}^{(k+1,\text{black})} = \max\{0, w_{ij}^{(k+1/2,\text{black})}\}. \tag{3.8c}$$

Each stage can be carried out in parallel, with the red (black) processors working and the black (red) processors idle.

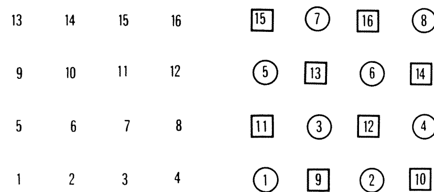


FIG. 3.2. Ordering of gridpoints (for a 4 × 4 grid) (a) usual, (b) red (○) and black (□).

This idea of using the red-black ordering for parallel processors has appeared several times in the literature (Heller [16]). Its use on DAP was first suggested by Hunt [17]. (In Europe, white-black chessboards are more usual than red-black ones).

In Table IV we list a DAP-FORTRAN subroutine PROJSOR for implementing the heart of the algorithm (3.7), (3.8). The subroutine is provided with several input parameters with obvious meanings. In addition, two logical matrices are provided as input: the logical matrix *MASKMASK* is true at gridpoints in the interior of the dam, and false elsewhere; the logical matrix *MASK* is true at black gridpoints and false at red gridpoints. Finally, the values of the real matrix W at the boundary points ∂R must be computed using (3.4d) before PROJSOR is called.

The computation time for one pass through the main loop of the subroutine

TABLE IV

The DAP Subroutine PROJSOR

```
      COMMON/RMAT/W( , )
      COMMON/RSCA/MAX DIFF, OMEGA, EPSILON, DAM WIDTH, DAM HEIGHT
      COMMON/ISCA/NUMB ITERATIONS, NUMB ROWS, NUMB COLS
      COMMON/SUBLMAT/MASK( , ), MASK MASK( , )
      REAL W, MAX DIFF, OMEGA, EPSILON, DAM WIDTH, DAM HEIGHT
      LOGICAL MASK, MASK MASK
      INTEGER NUMB ITERATIONS, NUMB ROWS, NUMB COLS
      REAL Z( , ), GRID2, W( , ), SAVEW( , )        Local variables.
      REAL ALPHA, BETA
      INTEGER NUMB TIMES
      LOGICAL DONE, WSIGN( , )
      EQUIVALENCE (WSIGN, W)
      W(WSIGN) = 0.0                                 Ensure that W is nonnegative everywhere.
      ALPHA = OMEGA * .25                            Calculate the constants that are needed
      BETA = 1.0 - OMEGA                               later on.
      GRID2 = (DAM HEIGHT/NUMB ROWS) ** 2
40    SAVE W = W                                     Start main loop.
      NUMB ITERATIONS = NUMB ITERATIONS + 1          Save the old value of W.
      DO 45 NUMB TIMES = 1, 2
 1    MASK(MASK MASK) = .NOT. MASK                   Reverse state of MASK.
 2    Z = W + W(-,-)                                 Calculate Z on only the red (or black)
 3    Z = Z(+, ) + Z( ,+) - GRID2                      points as determined by the MASK.
 4    W(MASK) = ALPHA * Z + BETA * W
 5    W(WSIGN .AND. MASK MASK) = 0.0                 Project
45    CONTINUE
      MAX DIFF = MAX(ABS(SAVEW - W))                 Find maximum difference between old
                                                       and new.
      DONE = (MAX DIFF .LE. EPSILON)                 Check if desired accuracy is attained.
      IF (.NOT. DONE) GO TO 40
      RETURN
```

TABLE V

Estimated Computation Time for the Inner Loop of PROJSOR

Statement	Operations	Time (μs)
1 $MASK(MASKMASK) =$ .NOT. $MASK$	1 logical mask	1
	1 logical negation	1
	1 logical store	1
2 $Z = W + W(-,-)$	1 index shift two places	21
	1 floating point matrix addition	160
3 $Z = Z(+,\ ) + Z(\ ,+) - GRID2$	2 index shifts	30
	1 floating point matrix addition	160
	1 floating point matrix subtraction	160
	1 scalar-matrix assignment	15
4 $W(MASK) = ALPHA * Z + BETA * W$	1 floating point matrix addition	160
	2 floating point matrix multiplications by a constant	400
	1 logical mask	1
5 $W(WSIGN$ .AND. $MASK\ MASK) = 0.0$	1 logical AND	2
	1 logical mask	1
	1 scalar-matrix assignment	15
DO 45 $NUMB\ TIMES = 1, 2$		7
		1135

PROJSOR is estimated in Table V, from which it follows that each PROJSOR iteration, which requires two passes through the loop, takes about $2 \times 1135\,\mu s = 2.27$ ms. To check this estimate, the average execution time per iteration in the subroutine PROJSOR was obtained by measuring (on a real external physical clock) the time required for a large number of iterations for the dam problem with $H = 24$, $h = 0$, $L = 16$, and $\Delta x = 1$. (This particular problem was chosen because it is a test problem which has been solved by many authors). The measured time per iteration on the Pilot DAP was 2.2 ms, as compared to the estimated time of 2.27 ms.

We conclude this section with some comments.

(1) For comparison, the dam problem with $H = 24$, $h = 0$, $L = 16$, and $\Delta x = 1$ was also solved on the UNIVAC 1180 at the University of Wisconsin, using the conventional ordering of gridpoints and an optimizing compiler with single precision arithmetic (36 bits), and the time per iteration was found to be 5.29 ms. For this problem the Pilot DAP was therefore 2.4 times faster than the UNIVAC 1180.

It should be noted that for this problem only $25 \times 17 = 425$ of the 1024 DAP PEs were used. On a 31×31 region the Pilot DAP would be six times faster than the UNIVAC 1180.

(2) In general, one expects to be able to predict DAP execution times to within

about 5 %, because DAP programs have little overhead and spend almost all their time in computation.

(3) Since DAP floating point operations are relatively expensive, it is worthwhile optimizing the code. (Readers who used early computers which also had relatively slow arithmetic operations may feel nostalgic). An example of such optimization occurs in the subroutine PROJSOR (see Table IV). The computation (3.7a) could have been implemented as:

$$Z = W(+,\ ) + W(-,\ ) + W(\ ,+) + W(\ ,-) - GRID2$$

which requires three additions and one subtraction, and takes

$$\underset{\text{(shifts)}}{4(15)} + \underset{\text{(additions)}}{4(160)} + \underset{\text{(scalar-matrix assignment)}}{15} = 715\,\mu\text{s}.$$

By sharing intermediate results between PEs, however, the amount of arithmetic can be reduced; the implementation in PROJSOR is

$$Z = W + W(-,-)$$
$$Z = Z(+,\ ) + Z(\ ,+) - GRID2,$$

which is estimated at only 546 μs. It should be noted that both implementations use only half the PEs for arithmetic at any one time. Larger grids or three-dimensional problems (see Section 4) can use all the PEs simultaneously.

(4) The UNIVAC 1180 was used for comparison, because this was readily available. It would be of interest to have timings on a computer such as the Cray 1.

4. Numerical solution of a Three-Dimensional Free Boundary Problem

A three-dimensional extension of the dam problem of Fig. 3.1 was introduced by Stampacchia [24] (see also France [13]). Water seeps through a porous dam in a rectangular channel of width a and height H. The walls of the dam are vertical but the thickness of the dam is variable, so that the dam occupies the region

$$\Omega_3 = \Omega_2 \times (0, H), \tag{4.1}$$

where the horizontal cross section Ω_2 is of the form

$$\Omega_2 = \{(x, y): 0 < x < a, \varphi_1(x) < y < \varphi_2(x)\}. \tag{4.2}$$

In the specific problem considered here, Ω_2 is the L-shaped region

$$\Omega_2 = (0, ED) \times (0, FE) \cup [ED, AF) \times (0, AB), \tag{4.3}$$

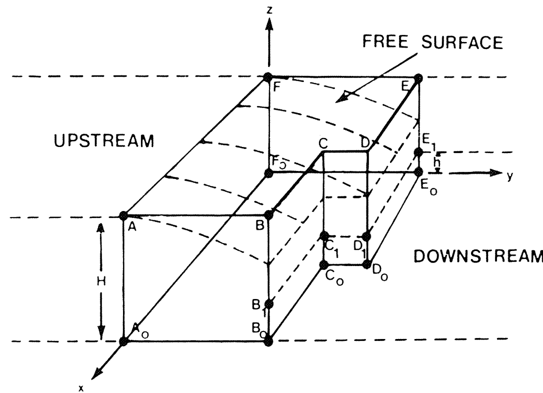


FIG. 4.1. Flow through a three-dimensional porous dam with L-shaped horizontal cross section.

where the points A, B, C, D, E, and F are as shown in Figure 4.1. The upstream water height is H and the downstream water height is h.

As shown by Stampacchia [24], the problem can be formulated as follows: Find u on the region Ω_3 such that:

$$-\nabla^2 u = -[u_{xx} + u_{yy} + u_{zz}] \geqslant -1 \quad \text{in } \Omega_3, \tag{4.4a}$$

$$u \geqslant 0 \quad \text{in } \Omega_3, \tag{4.4b}$$

$$u(-\nabla^2 u + 1) = 0 \quad \text{in } \Omega_3; \tag{4.4c}$$

and

$$\begin{aligned} u = g = \tfrac{1}{2}(H-z)^2, &\quad \text{on the upstream face } AA_0F_0F, \\ = \tfrac{1}{2}(h-z)^2, &\quad \text{on the downstream face below water level } B_0C_0D_0E_0E_1D_1C_1B_1, \\ = 0, &\quad \text{on the downstream face above water level } B_1C_1D_1E_1EDCB, \\ = 0, &\quad \text{on the top } ABCDEF, \\ = \alpha(x, y), &\quad \text{on the bottom } A_0B_0C_0D_0E_0F_0; \end{aligned} \tag{4.5}$$

and

$$u_x \equiv u_n = 0, \quad \text{on the sides } ABB_0A_0 \text{ and } EFF_0E_0. \tag{4.6}$$

Here $\alpha(x, y)$ is the solution of the two-dimensional mixed boundary value problem

$$\alpha_{xx} + \alpha_{yy} = 0, \quad \text{on } A_0B_0C_0D_0E_0F_0, \tag{4.7a}$$

$$\begin{aligned} \alpha = \tfrac{1}{2}H^2, &\quad \text{on } A_0F_0, \\ = \tfrac{1}{2}h^2, &\quad \text{on } B_0C_0D_0E_0, \end{aligned} \tag{4.7b}$$

$$\alpha_x \equiv \alpha_n = 0, \quad \text{on } A_0B_0 \cup E_0F_0. \tag{4.7c}$$

To solve problem (4.4)–(4.7) numericaly, we introduce a grid with $\Delta x = \Delta y = \Delta z$ and denote the approximation to $u([i-2]\Delta x, [j-1]\Delta y, [k-1]\Delta z)$ by w_{ijk}, and the approximation to $\alpha([i-2]\Delta x, [j-1]\Delta y)$ by $w_{ij} \equiv w_{ij1}$, for $2 \leqslant i \leqslant M-1$ and $1 \leqslant j \leqslant N$. As in Bruch [4], the computation proceeds in two stages.

Stage I. The two-dimensional problem (4.7) is approximated by replacing differential equation (4.7a) by the difference equations

$$4w_{ij} - w_{i+1,j} - w_{i-1,j} - w_{i,j+1} - w_{i,j-1} = 0. \tag{4.8}$$

The Dirichlet boundary conditions (4.7b) are satisfied by computing and storing the values of $w_{ij1} = \alpha_{ij}$ on A_0F_0 and $B_0C_0D_0E_0$. The Neumann conditions (4.7c) are satisfied by introducing two fictitious rows of gridpoints, adjacent to A_0B_0 and E_0F_0 respectively, and requiring that the values of w on a fictitious row should be equal to the values of w on the corresponding interior row; that is, $w_{1j} = w_{3j}$ and $w_{M,j} = w_{M-2,j}$, for $1 \leqslant j \leqslant N$.

The resulting system of equations is solved using a simple modification of the subroutine PROJSOR (see Table IV): the term $-GRID2$ is dropped from statement number 3; statement number 5 is deleted; and the statements

$$\begin{aligned} W(1,\ ) &= W(3,\ ), \\ W(M,\ ) &= W(M-2,\ ), \end{aligned} \tag{4.9}$$

are inserted between statements number 1 and 2, so as to make the values at the fictitious points equal to the corresponding interior values;

Stage II. The three-dimensional problem (4.4) is approximated by the LCP

$$\begin{aligned} 6w_{i,j,k} \geqslant{} & w_{i+1,j,k} + w_{i-1,j,k} + w_{i,j+1,k} + w_{i,j-1,k} \\ & + w_{i,j,k-1} + w_{i,j,k+1} - (\Delta x)^2, \end{aligned} \tag{4.10a}$$

$$w_{i,j,k} \geqslant 0, \tag{4.10b}$$

$$\begin{aligned} w_{i,j,k}[6w_{i,j,k} & - w_{i+1,j,k} - w_{i-1,j,k} - w_{i,j+1,k} - w_{i,j-1,k} \\ & - w_{i,j,k+1} - w_{i,j,k-1} + (\Delta x)^2] = 0. \end{aligned} \tag{4.10c}$$

The Dirichlet boundary conditions (4.5) are readily imposed, while the Neumann conditions (4.6) are treated by introducing fictitious sides parallel to the sides ABB_0A_0 and EFF_0E_0, and requiring that the values of w on the fictitious sides be equal to the values of w at the corresponding interior points.

To solve the LCP (4.10) we introduce a three-dimensional red-black partitioning of the gridpoints, so that each red (black) gridpoint has six black (red) neighbors. (It should be noted that the red/black ordering on any horizontal plane is the negation of the red/black orderings on the adjacent horizontal planes.) As in the two-dimensional problem treated in Section 3, each projected SOR iteration can be broken down into two stages: a red stage in which projected SOR is applied to all the

red points in the three-dimensional w array, followed by a similar black stage. In detail,

Red stage.

$$z_{ijk}^{(k,\mathrm{red})} = w_{i+1,j,k}^{(k,\mathrm{black})} + w_{i-1,j,k}^{(k,\mathrm{black})} + w_{i,j+1,k}^{(k,\mathrm{black})} + w_{i,j-1,k}^{(k,\mathrm{black})} + w_{i,j,k+1}^{(k,\mathrm{black})} + w_{i,j,k-1}^{(k,\mathrm{black})} - (\Delta x)^2, \tag{4.11a}$$

$$w_{ijk}^{(k+1/2,\mathrm{red})} = (\omega/6)\, z_{ijk}^{(k,\mathrm{red})} + (1-\omega)\, w_{ijk}^{(k,\mathrm{red})}, \tag{4.11b}$$

$$w_{ijk}^{(k+1,\mathrm{red})} = \max\{0, w_{ijk}^{(k+1/2,\mathrm{red})}\}. \tag{4.11c}$$

Black stage.

$$z_{ijk}^{(k,\mathrm{black})} = w_{i+1,j,k}^{(k+1,\mathrm{red})} + w_{i-1,j,k}^{(k+1,\mathrm{red})} + w_{i,j+1,k}^{(k+1,\mathrm{red})} + w_{i,j-1,k}^{(k+1,\mathrm{red})} + w_{i,j,k+1}^{(k+1,\mathrm{red})} + w_{i,j,k-1}^{(k+1,\mathrm{red})} - (\Delta x)^2, \tag{4.12a}$$

$$w_{ijk}^{(k+1/2,\mathrm{black})} = (\omega/6)\, z_{ijk}^{(k,\mathrm{black})} + (1-\omega)\, w_{ijk}^{(k,\mathrm{black})}, \tag{4.12b}$$

$$w_{ijk}^{(k+1,\mathrm{black})} = \max\{0, w_{ijk}^{(k+1/2,\mathrm{black})}\}. \tag{4.12c}$$

To implement the algorithm (4.11), (4.12) it was assumed that the dimensions of Ω_2 were such that the gridpoints on any horizontal cross section of the dam could be regarded as a subset of a 32×32 array. The solution w was stored as an array of matrices, the matrix $W(\ ,\ ,k)$ containing the values of w on the horizontal plane at a height $(k-1)\Delta z$. To control the parallel computation, two logical matrices were used: *MASKRB* which is true at interior red gridpoints in the current horizontal cross section and false otherwise; and *MASKMASK* which is true at interior points of Ω_2 and false otherwise.

The algorithm (4.11), (4.12) was implemented in two ways.

Implementation 1. During each red (black) stage the horizontal planes were updated in turn, and on each plane the red (black) points were updated in parallel.

The computation of $z^{(k)}$ requires five additions and one subtraction. Given an unlimited number of processors, n additions/subtractions require $\log_2 n$ steps, so that six additions/subtractions require at least three steps. By taking advantage of idle PEs, and remembering that, on the DAP, shift operations are much faster than arithmetic operations, the DAP-FORTRAN subroutine in Table VI is an efficient implementation of (4.11), (4.12) (compare Table IV). A full listing of the program is available upon request from the authors.

The subroutine in Table VI uses the functions SHS(outh) and SHN(orth) to shift W instead of the equivalent, but slower, statements (4.9).

Implementation 2. As in the three-dimensional magnetohydrodynamic code of Reddaway [22] we rearrange the values of w. Horizontal planes are considered in pairs, and the red points on each even-numbered plane are exchanged with the corresponding black points on the next odd-numbered plane. As a result, instead of

TABLE VI

First Implementation of (4.11) and (4.12)

```
C THE MAIN LOOP - PROCESS ALL THE Z PLANES
C
      SUBROUTINE MAIN LOOP
      COMMON/ISCA/TOPPLANE, M
      COMMON/ISCA/DAMMAXITERS, BOTTOMMAXITERS, NUMBITERS, NUMBOT
      INTEGER TOPPLANE, M
      INTEGER DAMMAXITERS, BOTTOMMAXITERS, NUMBITERS
      REAL DAMEPSILON, BOTTOMEPSILIN, OMEGA, MAXDIFF
      COMMON/RSCA/DAMEPSILON, BOTTOMEPSILON, OMEGA, MAXDIFF
      COMMON/RMAT/W( , ,25)
      COMMON/SUBLMAT/MASKRB( , ), MASKMASK( , )
      LOGICAL MASKRB, MASKMASK
      REAL SAVEW( , ), Z( , ), Z1( , )
      REAL MAXSOFAR, ALPHA, BETA, WIDGRID2, WIDTHGRID
      INTEGER NUMBTIMES
      LOGICAL TEMPMASK( , ), DONE, WSIGN( , )
      EQUIVALENCE(WSIGN, Z)
      ALPHA = OMEGA * 1.0/6.0
      BETA = 1.0 - OMEGA
C
C WIDTH OF GRID (I.E. ONE UNIT SQUARE) IS SET TO 1.0
C
      NUMBITERS = 0
      WIDTHGRID = 1.0
      WIDGRID2 = WIDTHGRID * WIDTHGRID
C
C SAVE THE MASKRB FOR LATER RESTORATION
C
      TEMPMASK = MASKRB
C
C MAXDIFF IS THE MAXIMUM DIFFERENCE BETWEEN SAVEW
C AND W( , ,K) AFTER W( , ,K)
C HAS ITS RED (OR BLACK) VALUES CHANGE (FOR ALL K)
C
1     MAXDIFF = 0.0
      NUMBITERS = NUMBITERS + 1
      MASKRB = TEMPMASK
      DO 30 NUMBTIMES = 1, 2
C
C ITERATE FROM THE 2ND PLANE TO THE TOP PLANE
C
      DO 20 K = 2, TOPPLANE
      SAVEW = W( , ,K)
C
C REVERSE RED/BLACK FOR SUCCESSIVE PLANES
C
      MASKRB(MASKMASK) = .NOT. MASKRB
```

Table continued

TABLE VI (*continued*)

```
C
C SUM THE SIX NEIGHBORS
      SAVEW(1, ) = SHN(SAVEW, 2)
      SAVEW(M, ) = SHS(SAVEW, 2)
      Z = SAVEW(-, -)
      Z1 = SAVEW
      Z1(MASKRB) = W( , , K + 1)
      Z(MASKRB) = W( , , K - 1)
      Z = Z + Z1
      Z1 = Z( , +)
      Z1(.NOT. MASKRB) = -WIDGRID2
      Z = Z + Z1
      Z = Z + Z(+, )
C
C STORE THE AVERAGE OF THE SIX NEIGHBORS IN W
C ONLY IN THE RED (OR BLACK) CELLS
C
      Z = ALPHA * Z + BETA * SAVEW
      Z(WSIGN) = 0.0
      W(MASKRB, K) = Z
C
C FIND THE MAXIMUM DIFFERENCE ON THIS PLANE
C
      MAXSOFAR = MAX(ABS(SAVEW - Z), MASKRB)
      IF (MAXSOFAR .GT. MAXDIFF) MAXDIFF = MAXSOFAR
20    CONTINUE
C
C REVERSE STATE OF ORIGINAL MASKRB FOR THE 2ND PASS
C THROUGH THE PLANES
C
      MASKRB(MASKMASK) = .NOT. TEMPMASK
30    CONTINUE
      DONE = (NUMBITERS .GT. DAMMAXITERS) .OR. (MAXDIFF .LE. DAMEPSILON)
      IF (.NOT. DONE) GOTO 1
      MASKRB = TEMPMASK
      RETURN
      END
C
C
```

having n planes, each containing red and black points in a checkerboard pattern, we have $n/2$ planes of red points interleaved with $n/2$ planes of black points. This makes it possible to use simultaneously all interior PEs for arithmetic.

The corresponding subroutine is given in Table VII. In the full program, to save time, the test for convergence was executed only every *TIMES* iterations, where *TIMES* is an input parameter.

The subroutine in Table VII assumes that there is an even number of planes. To

TABLE VII

Second Implementation of (4.11) and (4.12)

```
  THE MAIN LOOP - PROCESS ALL THE Z PLANES
      SUBROUTINE MAIN LOOP
      COMMON/ISCA/TOPPLANE, M
      COMMON/ISCA/DAMMAXITERS, BOTTOMMAXITERS, NUMBITERS, NUMBOT
      INTEGER TOPPLANE, M
      INTEGER DAMMAXITERS, BOTTOMMAXITERS, NUMBITERS, TIMES
      REAL DAMEPSILON, BOTTOMEPSILON, OMEGA, MAXDIFF
      COMMON/RSCA/DAMEPSILON, BOTTOMEPSILON, OMEGA, MAXDIFF, TIMES
      COMMON/RMAT/W( , ,25)
      COMMON/SUBLMAT/MASKRB( , ), MASKMASK( , )
      COMMON/WORK/Z, WK
      LOGICAL MASKRB, MASKMASK
      REAL SAVEW( , ), Z( , ), Z1( , ), WKP1( , ), WK( , ), MAXD( , )
      REAL MAXSOFAR, ALPHA, BETA, WIDGRID2, WIDTHGRID
      INTEGER NUMBTIMES
      LOGICAL TEMPMASK( , ), DONE, WSIGN( , ), TEST, NOTTEST
      EQUIVALENCE(WSIGN, Z), (Z, Z1), (WK, WKP1)
      ALPHA = OMEGA * 1.0/6.0
      BETA = 1.0 - OMEGA
C
C WIDTH OF GRID (I.E. ONE UNIT SQUARE) IS SET TO 1.0
C
      NUMBITERS = 0
      WIDTHGRID = 1.0
      WIDGRID2 = WIDTHGRID * WIDTHGRID
C
C
1     TEST = TIMES .EQ. 1
      NOTTEST = .NOT. TEST
      ITIMES = TIMES
      MAXD = 0.0
C
C
C ALTER ALL THE ODD NUMBERED PLANES:
2     KM2 = 1
      DO 20 K = 2, TOPPLANE, 2
      SAVEW = W( , ,K + 1)
      WK = W( , ,K)
      WK(1, ) = SHN(WK, 2)
      WK(M, ) = SHS(WK, 2)
      Z = WK + WK(-,-)
      Z = (Z(+, ) + Z( ,+) + WK + MERGE(W( , ,KM2), W( , ,K + 2), MASKRB)
          - WIDGRID2) * ALPHA + SAVEW * BETA
      Z(WSIGN) = 0.0
      W(MASKMASK, K + 1) = Z
      IF (NOTTEST) GOTO 20
      Z1 = ABS(SAVEW - Z)
      MAXD(Z1 .GT. MAXD) = Z1
```

Table continued

TABLE VII (*continued*)

```
20      KM2 = K
C
C
C ALTER ALL THE EVEN NUMBERED PLANES:
        DO 21 K = 2, TOPPLANE, 2
        SAVEW = W( , ,K)
        WKP1 = W( , ,K + 1)
        WKP1(1, ) = SHN(WKP1, 2)
        WKP1(M, ) = SHS(WKP1, 2)
        Z = WKP1 + WKP1(-, -)
        Z = (Z(+, ) + Z( ,+) + WKP1 + MERGE(W( , ,K + 3), W( , ,K - 1), MASKRB)
           - WIDGRID2) * ALPHA + SAVEW * BETA
        Z(WSIGN) = 0.0
        W(MASKMASK, K) = Z
        IF (NOTTEST) GOTO 21
        Z1 = ABS(SAVEW - Z)
        MAXD(Z1 .GT. MAXD) = Z1
21      CONTINUE
C
C
        ITIMES = ITIMES - 1
        IF (ITIMES .GT. 1) GOTO 2
        IF (ITIMES .EQ. 0) GOTO 3
        TEST = .TRUE.
        NOTTEST = .FALSE.
        GOTO 2
C
3       NUMBITERS = NUMBITERS + TIMES
        MAXDIFF = MAX(MAXD, MASKMASK)
        IF (MAXDIFF .LT. DAMEPSILON) GOTO 4
C
        IF (NUMBITERS .LT. DAMMAXITERS) GOTO 1
4       RETURN
        END
```

avoid additional testing, it is assumed that a copy of the top plane is stored above the top plane.

The two implementations were run on the problem with $H = 10$, $h = 0$, $AF = FE = 20$, $CD = BC = 10$, which was chosen because it had previously been solved by Bruch [4]. For comparison, the problem was also solved on the UNIVAC 1180 using single precision arithmetic and optimized FORTRAN code. The measured computation times per projected SOR iteration (including both red and black stages) were:

Implementation 1: 32 ms,
Implementation 2: 16.0–18.2 ms (dependent on frequency of convergence tests),
UNIVAC 1180: 34 ms,

so that Implementation 2 on the Pilot DAP is about two times faster than the UNIVAC 1180. The estimated time per SOR iteration (Implementation 2) was found as in Table V, and was found to lie between 15.4 and 17.4 ms, depending upon the frequency of convergence tests.

For this problem, only 383 (i.e., $21 \times 23 - 10 \times 10$) of the 1024 PEs were used.

5. Future Possibilities

(a) For purposes of comparison we have used previously published problems but they have dimensions which do not match the DAP array closely. In many practical problems, the resolution would be tailored to the DAP dimensions to achieve higher performance.

(b) The programs presented are readily extensible to larger problems on correspondingly larger DAPs such as the production 64×64; it is only necessary to change the boundaries. The time to process one plane would be unchanged.

(c) Performance on small three-dimensional problems can be improved by mapping several problem planes onto one DAP matrix.

(d) Problems with large *horizontal* dimensions can be mapped with each PE holding a small neighborhood group of points. Performance improves because each PE holds both black and red points (Hunt [18]).

(e) Very large problems cannot be held entirely within DAP store. For example, with four times as many points in a horizontal plane as there are PEs, the limit is about 26 planes with 4 K bits per PE or about 122 planes with 16 K bits per PE. With backing store, the transfers rates with N active problem planes in the DAP can be minimized by advancing each plane $(N-2)/2$ iterations per backing store fetch. Hence it should be possible to achieve a balance between input-output and processing times (Reddaway [22]).

(f) Problems of this type offer possibilities for using fixed point arithmetic (with suitable scaling) and using low precision for computing the iterative corrections. This is much faster than floating point work and performance improvements as large as a factor of ten are predicted without loss of accuracy in the final solution.

6. Conclusions

We have demonstrated that two- and three-dimensional linear complementarity problems can be solved on DAP with high performance and easy programming using a version of projected SOR. There is scope for even higher performance and for tackling a wide range of problem sizes.

APPENDIX: THE PILOT HOST-DAP INTERFACE

In the Pilot DAP system, the store of the DAP is not an integral part of the host's store as with the production DAPs. It is therefore necessary to explicitly move data between the host and DAP, and this is achieved by using standard host FORTRAN subroutines. The subroutine names begin with DAPTO or DAPFROM depending on whether they move data into or out of the DAP. The remaining letters of the name indicate the type (integer or real denoted by I or E) and rank (scalar, vector or matrix denoted by S, V, or M) of the variable transferred. Parameters of DAPTO and DAPFROM give the name of the host program variable and the location within the DAP in terms of the name of the common area and the offset from the start of this area.

Initiation of DAP processing is also less direct on the Pilot system with DAP-FORTRAN subroutines being called via the standard host FORTRAN subroutine DAPGO. A statement of the form:

CALL DAPGO('DAPSUB', N)

will suspend execution of the host FORTRAN and transfer control to the DAP-FORTRAN subroutine DAPSUB. Execution of the host FORTRAN is resumed after DAPSUB and any further levels of DAP-FORTRAN subroutines have been executed. The parameter N gives the maximum number of seconds allowed for DAP processing.

ACKNOWLEDGMENTS

C. W. Cryer and J. Stansbury gratefully acknowledge the provision by International Computers Ltd. of facilities for using the Pilot DAP at Stevenage, England.

REFERENCES

1. C. BAIOCCHI, *Ann. Mat. Pura Appl.* **92**(4) (1972), 107.
2. M. L. BALINSKI AND R. W. COTTLE, Ed., "Complementarity and Fixed Point Problems," North-Holland, Amsterdam, 1978.
3. A. BRANDT AND C. W. CRYER, "Multigrid algorithms for the solution of linear complementarity problems arising from free boundary problems," Technical Summary Report No. 2131, Mathematics Research Center, Univ. of Wisconsin–Madison, 1980.
4. J. C. BRUCH, JR., *Adv. Water Resour.* **3** (1980), 115.
5. R. W. COTTLE, F. GIANNESSI, AND J. L. LIONS, Ed., "Variational Inequalities and Complementarity Problems," Wiley, New York, 1980.
6. R. W. COTTLE, G. H. GOLUB, AND R. S. SACHER, *Appl. Math. Optim.* **4** (1978), 347.
7. C. W. CRYER, *SIAM J. Control Optim.* **9** (1971), 385.
8. C. W. CRYER, *in* "Proceedings, Seminar on Free Boundary Problems, Pavia, 1979" (E. Magenes, Ed.), Vol. 2, pp. 109–131, Instituto Nazionale di Alta Matematica Francesco Severi, Rome, 1980.
9. C. W. CRYER AND M. A. H. DEMPSTER, *SIAM J. Control Optim.* **18** (1980), 76.

10. G. DUVAUT AND J. L. LIONS, "Inequalities in Mechanics and Physics," Dunod, Paris, 1976.
11. P. M. FLANDERS, *in* "Supercomputers," Infotech International, London, 1979, 117.
12. P. M. FLANDERS, D. J. HUNT, S. F. REDDAWAY, AND D. PARKINSON, *in* "High Speed Computer and Algorithm Organization" (D. J. Kuck, Ed.), Academic Press, New York, 1977.
13. P. W. FRANCE, *J. of Hydrology* **21** (1974), 381.
14. R. GLOWINSKI, *Rendiconti di Mathematica* **14** (1971), Universita di Roma.
15. R. W. GOSTICK, ICL *Tech. J.* **1** (1979), 116.
16. D. HELLER, *SIAM Rev.* **20** (1978), 740.
17. D. J. HUNT, "Numerical Solution of Poisson's Equation on an Array Processor Using Iterative Techniques," Report No. CM21, International Computers Limited, Research and Advanced Development Centre, Stevenage, 1974.
18. D. J. HUNT, *in* "Supercomputers," Infotech International, London, 1979, 205.
19. ICL Technical Publication No. 6918, "DAP Fortran Language," 1979.
20. D. KINDERLEHRER AND G. STAMPACCHIA, "An Introduction to Variational Inequalities and their Applications," Academic Press, New York, 1980.
21. O. M. MANGASARIAN, *J. Optim. Theory Appl.* **22** (1977), 465.
22. S. F. REDDAWAY, "A 3D Magnetohydrodynamics Code (3DMHD) on DAP," Report No. CM59, International Computers Limited, Research and Advanced Development Centre, Stevenage, 1976.
23. S. F. REDDAWAY, *in* "Supercomputers," Infotech International, London, 1979, 309.
24. G. STAMPACCHIA, *Russian Math. Surveys* **29** (1974), 89.

Printed by the St. Catherine Press Ltd., Tempelhof 41, Bruges, Belgium

DISTRIBUTED ARRAY PROCESSOR, ARCHITECTURE AND PERFORMANCE

S.F. Reddaway
International Computers Limited
Stevenage
England

1. INTRODUCTION

The origin of the DAP (Distributed Array Processor) architecture lay in matching the capabilities of rapidly advancing semi-conductor technology with some real world problems. A major feature that suited both the technology and the applications was parallelism, and the design adopted was a large array of bit-organised processing elements (PEs), each having both storage and processing. The emphasis was on simplicity, generality and flexibility.

The approach is consistently parallel throughout and by-passes many of the problems associated with grafting some parallelism onto heavily ingrained sequential practices. In particular, a parallel language (DAP-Fortran) is a key component providing both natural parallel application programming and efficient hardware execution. The approach is a more radical departure than that adopted by vector machines, which leads to both advantages and disadvantages.

Vector machines emphasise arithmetic performance, especially floating point multiply and add/subtract on one (or at most two) precisions. DAP is very flexible with respect to number representation and precision, with full trade-offs against speed and storage space. In a wider sense the bit organisation gives generality of function as we shall see. This means that DAP performance on a complete algorithm or application is often much more impressive than the raw MFLOPS processing power.

The simple nature of the bit-organisation means that hardware development is relatively cheap and quick and the result relatively cheap to manufacture and maintain. An emphasis has been cost-effective power, rather than power-at-any-price.

NATO ASI Series, Vol. F7
High-Speed Computation. Edited by J. S. Kowalik
© Springer-Verlag Berlin Heidelberg 1984

An important aspect is the balance between storage and raw processing power. In many applications, a large store is of very great value. Considerable emphasis has therefore been given to providing a large store, even at the expense of processing power, to enable effective problem solving.

2. DAP ARCHITECTURE

This has been described elsewhere (references 1,2,3) so this account is brief.

DAP is an SIMD machine with a large array of simple bit-organised PEs under the control of a Master Control Unit (MCU). Connectivity is of 3 kinds, the first 2 giving the array a 2D (square) organisation.

a) 2D nearest neighbour connections, including wrap-around cyclic connections.

b) Row and column highways terminating on one of a set of MCU registers. As well as bit-vector transfers between an MCU register and any chosen row or column, a bit-vector can be replicated to make a bit-matrix and a bit-matrix can be condensed to a bit-vector by ANDing along rows or columns. If the latter is followed by a jump dependent on the MCU register being all True, a branch depending on a condition being True throughout the array has taken only two machine instructions.

c) A data bit can be broadcast to all PEs.

The store is 3D, formed by the 2D array of PEs each having a few thousand bits of store. Data can be stored and processed in many layouts, but only 2 basic modes are supported in DAP-Fortran. In vertical mode, each number is held entirely within one PE store, and the numbers stored in corresponding sets of addresses in each PE form a matrix of numbers; processing is parallel across all the numbers, but serial through the bits. In horizontal mode, a number is stored along a row of PEs at the same store address. A vector consists of a number in each row at the same address; vectors are processed essentially in parallel, with the one-bit adders in each PE being able to form a word adder along the rows. However, the processing rate for vectors is considerably slower than for matrices despite one vector operation taking considerably less time than one matrix operation. Scalars are held in horizontal mode, and processed in either the MCU or the array. Any precisions or data type (e.g. integer, floating point) can be used.

3. FIRST GENERATION DAPs

3.1 Hardware

This machine had 4096 PEs (64x64) each with 4K bits of store, later enhanceable to 16K bits. (i.e. 2MBytes enhancable to 8 MBytes). The store forms part of an ICL 2900 mainframe, which acts as a host. The DAP cycle time is 200 nsecs.

3.2 Software

This is discussed more fully in reference 4. In DAP-Fortran (reference 5), operations may be performed on three kinds of atom: Matrix, Vector and Scalar. The Matrix and Vector atoms match the DAP size: 64x64 elements and 64 elements respectively. Indexed sets of these atoms are allowed in a similar way to Fortran's indexed sets of scalars. Many precisions are supported, in 8-bit increments. As well as element by element parallel operations on arrays, there are several intrinsic functions for data reorganisation and for mixed rank operations, such as MAX or SUM of an array.

A powerful language feature is the use of Boolean arrays to control the assignment of results.

Programs in DAP-Fortran tend to be clear and concise leading to high programmer productivity. The elegance of some algorithmic constructs was a major inspiration to the forthcoming array extensions in ANSI Fortran 8X.

3.3 Basic Performance

As part of the system software, operations such as multiply or MAX have been programmed in the DAP assembly language APAL (reference 6). For example, store to store performance on 40-bit floating point matrix operations is about 20 MFLOPS for addition, and 10 MFLOPS for multiplication. Staying with 40 bits, better performance is shown by other operations: 20 MFLOPS for squaring, 15 MFLOPS for square root (this is the rate at which square roots are produced - equivalent to about 100 MFLOPS) and 60 MFLOPS for MAX (equivalent to over 200 MIPS). Also extremely fast are exponential, logarithm, trigonometrical functions and random number generation, which, along with square root and MAX, take good advantage of the bit-organisation.

Fixed point add is many times faster than floating point, and multiplication of a matrix by a scalar can be several times faster than matrix-matrix. Lower precision operations can be much faster, with multiply and related operations varying as roughly the square of the precision. Trick operations, such as changing the sign of floating point arrays, can achieve 10,000 MFLOPS. All the above performance rates include the option of local activity control.

Other less arithmetically oriented operations can be extremely fast, such as moving arrays, sorting and scanning data, compressing and expanding data, processing of Boolean and symbol arrays.

4. PERFORMANCE OF FIRST GENERATION DAPS

The first customer installation was at QMC (Queen Mary College of London University) in 1980 with the aim of providing a national service to the academic community. The machine has recently been enlarged from 2 MBytes to 8 MBytes. There are some 400 users, doing a wide range of mainly scientific applications. In all there is now an installed base of 5 machines. As expected, performance relative to other machines varies widely and, because the programs are different, is often not easy to measure.

Many of the reported results fall in the range 0.1 to 6x CRAY-1 (or the rough equivalent on other machines). As expected, performance on arithmetic applications is better than the raw MFLOPS figures would suggest. The best performances are for physical simulations that involve Boolean arrays; the array processing here is very fast even in DAP-Fortran, and APAL could offer a further order of magnitude improvement in these cases.

It is interesting that in the early days there was considerable theoretical criticism of DAP concerning lack of high performance on "scalar processing", even on generally favourable problems, and that this would be a dominating bottleneck. (For example, if 20% of work is "scalar", and array processing is, say, 100 times faster than scalar, then only 4% of the theoretical array performance would be achieved). In general, where complete applications have been written for the DAP, this has not happened. There are wide variations, but fairly typical is an equal number of matrix, vector and scalar operations. (The situation is complicated by scalar-matrix, scalar-vector and Boolean operations as well as data movement and control overheads). If one further assumes that the matrix operation effectively replaces only 2000 operations (instead of 4096) in the best scalar code, then the scalar operation is only 0.05% of the work (rather than the critics' 20%) and the

vector operation, if it replaces 50 operations, is only 2.5% of the work. Put another way, 99.95% parallelism is indicated. In terms of DAP time, assuming matrix/vector/scalar operation times are in the ratio 10:2:1, then 15% of time is vector and 8% is scalar (rather than the critics' 96%). The effective "MFLOPS rates" would be 40% of theoretical.

An actual example is inverting a matrix (64x64 for simplicity) with full pivoting. The DAP code has been given elsewhere (reference 1), and the required operations per step can be summarised as:

1 Matrix $\pm$
1 Scalar-vector divide
1 extract the pivot row and expand to a matrix
1 extract the pivot column and expand (after the above division) to a matrix
1 extract the pivot element from the pivot column
1 maximum element (matrix-scalar)
Some Boolean work

At the end, the rows and columns are re-ordered to take account of pivoting. The truly scalar work is almost negligible - extracting the pivot element from the pivot column, and expanding the scalar into a vector. This takes only about 1% of the DAP time. Extracting and expanding the pivot rows and pivot column and the scalar-vector divide are "vector operations" and take about 20% of the time. The Boolean work (mostly), maximum element and reordering are to do with pivoting and take about 20% of the time; this fraction is much less than for a serial machine. This example can be viewed as virtually 100% parallel, and the effective "MFLOPS rate" is about 79% of theoretical without pivoting and about 59% with pivoting. (The best serial codes need nearly as many arithmetic operations as this parallel code).

The above matrix inversion with full pivoting runs about an order of magnitude faster on the DAP than an equivalent code on a CDC 7600, and is one of many examples of performance being much better than indicated by peak arithmetic rates. Far from the supposed scalar content usually causing DAP applications performance to be worse than indicated by raw arithmetic there have often been reverse effects due to various causes.

Cases vary widely and not all algorithms map well onto DAP. However, the early criticism in terms of scalar performance appears to have been misconceived. It seems to have largely derived from experience of Fortran codes on early vector machines, where everything not performed in vector instructions was called scalar. Much of this was things like loop control, indexing, end effects and boundary conditions rather than essential scalar floating point arithmetic. (The latter is what DAP is weakest at; some of the scalar control operations are fast). The amount of loop control and indexing would be quite large on vector machines due to DO loops iterating most naturally over only one dimension of the problem at a time. Once a problem is structured to use DAP matrix processing, the time for residual loop control or matrix indexing is small compared to the matrix arithmetic; there is not much of it, and the pace of matrix operations is comparatively leisurely.

Boundary conditions usually have only minor effects. Either certain things are not done at boundaries, which can be controlled very easily, or something extra is required which can usually be dealt with effectively by conditional operations. The latter also arise in the very important case of data dependent conditional operations. One example is pivoting. A very different example is the highly conditional operations in meteorological "physics" codes which are highly dependent on the local presence of cloud, rain or ice. A Fortran code will vectorise poorly due to many tests and branches. With DAP a superset code is written which deals with all cases by using the Boolean matrices obtained from the tests to turn PE activities on and off. This is extremely fast, and although at times there may be very few PEs active, there is no code branching and execution speed is data independent. This is an example of the associative processing of arbitrary subsets of data, which can be very powerful. The effect of this kind of highly conditional processing is usually to improve DAP performance compared with other machines. A meteorological physics benchmark coded for DAP had several times better performance than the more regular "dynamics" part of the benchmark; as well as the "conditional" effect, logarithms and Table Look Up contributed to this good performance. (Similar superset conditional array processing can be applied in commercial data processing such as data validation or payroll).

Other key factors are choice of algorithm and good top-down mapping onto DAP and DAP-Fortran; good choices here result in clear and easy coding as well as good performance. It is often constructive to consider problems that are larger than the minimum size that spans the DAP array. ("Oversize problems"). There is a spectrum from the serial emphasis of grossly oversize problems to a parallel emphasis for problems of minimum size. In the four examples that follow it is advantageous to squash an oversize problem into the array such that several

neighbouring points are in the same PE. (This is known as "crinkled" mapping). This minimises both data movement and the effects of boundaries between sub-matrices. (In other contexts an oversize matrix can advantageously be divided into sub-matrix "sheets" that match the hardware array).

4.1 ADI

Here the solution of tridiagonal sets of equations has a parallel formulation in which log n more arithmetic is done than for the serial formulation (reference 7). However, increasingly oversize mappings look progressively more like the serial formulation and the excess work decreases.

4.2 Multi-Grid

Here an "exact fit" problem causes all except the finest grid to have wasted PEs. A problem oversize by a factor of only 2 or 4, spends most of its time in the finest two or three grids, which have no wasted PEs.

4.3 FFT

FFTs with one point per PE have been programmed and run on DAP. Having two points per PE is better because each PE performs a complete "butterfly" which enables considerable reduction in arithmetic and routing. Further gains can be achieved by further increasing the number of points/PE, with fewer multiplications (and more of those remaining being the faster scalar-matrix type) and less routing. More points per PE can either be achieved with larger transforms, or, more often, by doing many transforms in parallel; the individual transforms are becoming more serial. The extreme case is a complete transform in every PE, for which performance is several times better than with one point per PE. Because additions predominate over multiplications, fixed point arithmetic further substantially improves performance, and this suits most signal and image processing. As an example, 4096 64-point complex FFTs with 16-bit fixed point output precision, programmed in APAL, take only 20 msec; this corresponds to about 300 million arithmetic operations per second. With bulk FFTs some further advantage could be obtained with Winograd transforms.

4.4 Convolution by Fermat Number Transform

The solution of many problems can be formulated in terms of a convolution algorithm. Poisson's equation in some circumstances is one example, and there are several in signal processing. Another example is very high precision multiplication. Convolution in turn has efficient solutions in terms of transforms like the FFT and FNT (Fermat Number Transforms, reference 8). FNTs have the advantage that multiplications are eliminated (except for the one essential for the convolution) and they produce an exact result; this makes them attractive for multiplication. One use of very high precision multiplication is in testing Mersenne numbers for primality, using the Lucas-Lehmer test. In searching for such new primes large numbers of multiplications of the order of 100,000 bits are required.

On conventional number crunchers, most testing to date has used direct multiplication methods with the fast arithmetic facilities. A "divide and conquer" algorithm has also recently been used on CRAY-1 with a slight improvement, and a transform algorithm has very recently been used on CYBER 205 with some further improvement. From complexity theory the latter two algorithms should be much faster, but because they put more emphasis on fixed point addition and less on multiplication, the multiplication hardware favoured the basically slower direct method.

On DAP, the absence of multiplication hardware means performance reflects complexity theory much more closely. The FNT method has been implemented, and is about 100 times faster than the direct method. It is nearly twice as fast as a transform method on Cyber 205 and about four times faster than the direct method on CRAY-1. The FNT replaces ordinary arithmetic by fixed point add/subtracts which have some rather unusual carry and shift requirements. These can be handled better on the bit-organised DAP than on conventional word-organised hardware.

5. SUPERCOMPUTER MARKET

The market for machines with the emphasis on high performance 64-bit floating point multiply and add is well established, notwithstanding the fact that they are often not well matched to algorithms. Such a market might be attacked by a multi-bit DAP development. Unpublished work on PEs with a small locally addressable fast store and dealing with 4 or 8 bits at a time have shown the possibility of 100-200 times performance improvement per PE on high precision floating point, with 10-20 times

more hardware per PE. Some simplicity, flexibility and generality is lost, and the control side needs to move fast to keep up, but there is still more flexibility than in word organised machines.

An interesting question is how many applications really require very large machines. Mostly the requirement is for throughput rather than very fast absolute execution speed. In any event, the latter is often not achieved on big machines because of time sharing. Smaller machines under the control of smaller groups can serve many needs for large-scale cost-effective processing.

6. SECOND GENERATION DAPS

Development work is proceeding on a much smaller and cheaper DAP with fewer PEs and more integration, attachable to the PERQ single user workstation. An interesting area is signal processing and other high data rate applications. Good graphics will allow close interaction with programs which will often increase productivity.

Signal and image processing usually involve lower precision than number crunching and a variety of algorithmic techniques including highly data dependent Boolean processing. In this area, the competition is often from special hardware, with several different pieces of hardware for different stages of processing. With DAP, programmable power that can give powerful performance on many different stages can lead to a powerful and flexible "one-box" approach. The special hardware may be better at particular parts, but the DAP is often better overall, and alterable into the bargain.

As part of a very powerful single user workstation the array processor can contribute to a wide range of powerful functions to give high performance man-machine interaction, as well as providing powerful numerical processing.

7. REFERENCES

1. Flanders P.M., Hunt D.J., Parkinson D., Reddaway S.F.
"Efficient High Speed Computing With The Distributed Array Processor"
Proc. Conf. on High Speed Computer and Algorithm Organisation.
Academic Press Inc, 1977

2. Hunt D.J., Reddaway S.F.
"Distributing Processing Power In memory" in "The Fifth Generation Computer Project" Pergamon Infotech Ltd. 1983

3. Parkinson D.
"The Distributed Array Processor (DAP)"
to be published in "Computer Physics Communications", 1983

4. Flanders P.M.
"Fortran Extensions For A Highly Parallel Processor" in "Supercomputers"
Infotech International. 1979

5. International Computers Limited
"DAP: Fortran Language" Technical Publication No. 6918. 1979

6. International Computers Limited
"DAP: APAL Language" Technical Publication No. 6919. 1979

7. Hunt D.J., Webb S.J., Wilson A.
"Application of a Parallel Processor to the Solution of Finite Difference Problems" In "Elliptic Problem Solvers"
M.H. Schultz (ed). Academic Press 1981

8. McClellan J.H., Rader C.M.
"Number Theory in Digital Signal Processing"
Prentice Hall Inc. 1979

REPRINTED FROM
HIGH SPEED COMPUTER AND ALGORITHM ORGANIZATION
© 1977
ACADEMIC PRESS, INC
NEW YORK SAN FRANCISCO LONDON

EFFICIENT HIGH SPEED COMPUTING WITH THE DISTRIBUTED ARRAY PROCESSOR

P.M. Flanders, D.J. Hunt, S.F. Reddaway, D. Parkinson
International Computers Limited

The Distributed Array Processor (DAP) is a SIMD (Single Instruction - Multiple Data) machine which distributes a few thousand very simple bit-organised processing elements throughout a store module of a conventional computing system. The design is presented, together with details of its arithmetic and logical capability and of the software system being implemented. Some algorithms have been run on a pilot 32 x 32 DAP and their performance is discussed. The size and performance of probable future models are also considered. The results of some application studies are presented illustrating the high performance achievable with this flexible and cost effective processor.

I. PRINCIPLES

The Distributed Array Processor (DAP) is a computer architecture capable of achieving high performance on a variety of large computing jobs. It comprises a few thousand Processing Elements (PEs) arranged in a two-dimensional array. The PEs are very simple but high processing power is achieved by having many of them. They execute a common instruction stream broadcast by a Master Control Unit (MCU); the DAP is thus classified as a SIMD (Single Instruction - Multiple Data) machine.

Various sizes of PE array can be made, offering a range of processing powers. It is most natural for the array to be a square whose side is simply related to standard store highway widths. The design aims for cost-effectiveness rather than speed at any price.

The PEs are bit organised giving great flexibility. Hence parallelism can be exploited in operations such as table lookup, scanning data, symbol processing and sorting as well as arithmetic. Each PE has associated with it a few

thousand bits of fast random access storage. The fast parallel transfer between stores and PEs balances the high processing speed.

The totality of PE stores form a standard store module of a conventional computer. Hence there is no need for separate transfer of data between host and DAP. DAP processing is under the control of the host via a simple interface. Being part of a general purpose system has two related advantages. It gives access to the facilities of that system, in particular the operating system, languages and input-output, and it allows users progressively to take advantage of DAP processing.

A pilot model having 1024 PEs arranged 32 x 32 each with 1K (expandable to 2K) bits of bipolar store has been working since Spring 1976. It has a total of 89 boards, most of which are "array" boards each having 79 16-pin off-the-shelf TTL integrated circuits forming the logic and storage for 16 PEs. The total power dissipation excluding fans and power supply losses is under 1.5K Watts

The next model will achieve about five times more processing power with a 64 x 64 array. Each PE will have 4K bits giving a 2 MByte store module. The DAP architecture is ideal for custom LSI, so a 128 x 128 model could be built using today's semiconductor technology.

A. Processing Element

Figure 1 shows the essential features of one PE with some of the control and data paths omitted for clarity (the PE has been simplified since (1)). All data paths are one bit wide.

The top multiplexor selects the input to the Arithmetic and Logic Unit (ALU), which may be either the PEs own output or that from its North, East, South, or West neighbour. The bottom multiplexor selects the PE output, which may be an ALU output, or store output, and may go to the store, another PE or to the MCU. The two multiplexors also allow input of data broadcast from the MCU along row or column highways.

The ALU has three one-bit registers (A, Q and C) and a one-bit full adder. The low level software allows total flexibility in the use of ALU facilities but most common usage is as follows.

Register A provides 'activity' control; certain store write operations are only effective if this register is true. This allows application of a function to selected elements of an array and is also used for bit level implementation within functions. The A register incorporates a logical AND facility which allows conditions to be combined rapidly and

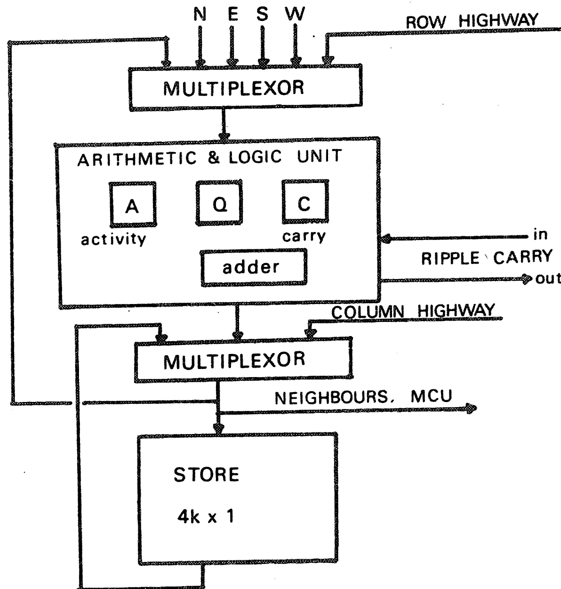


Fig.1. Processing Element.

may also be used in its own right for implementing general logic functions.

The Q register acts as a one bit accumulator and the C register as a carry store. The adder adds Q, C and the ALU input and its sum output may be written back to Q and its carry output to C.

B. Master Control Unit

Figure 2 is a schematic of the MCU, which may be likened to the instruction sequencing and control sections of a small computer. Implementation of an array instruction involves a Fetch phase, during which the instruction is fetched from the array store, and an Execute phase, during which appropriate control signals are broadcast to the array. Each phase uses one basic machine cycle.

Each instruction occupies 32 bits along one row of PEs and successive instructions are held in successive rows (or part rows) so that code is spread evenly among the PE stores. It must be emphasised that instructions can only be

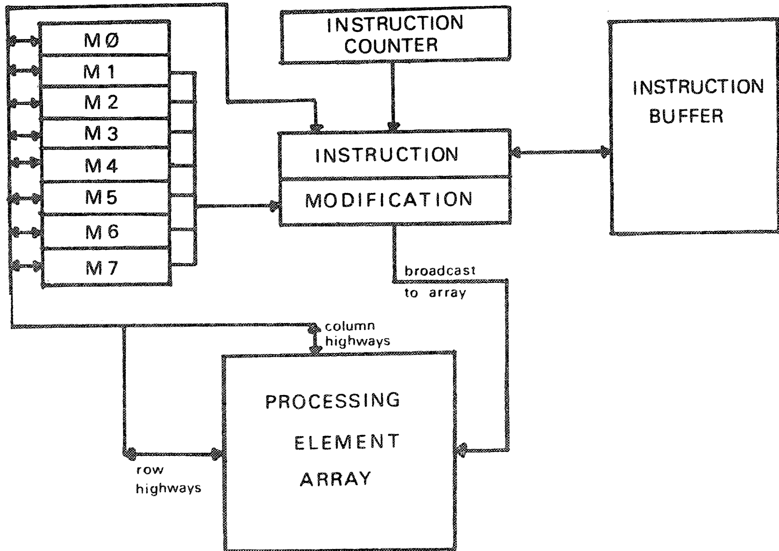


Fig.2. Master Control Unit

interpreted by the MCU; the PEs cannot recognise instructions themselves.

An important feature is the instruction buffer which stores loops (of up to 15 instructions on the pilot) explicitly specified by low level coding. Hence the instructions need to be fetched only once prior to the first time found the loop. Automatic stepping of addresses is possible for operating on successive bits of a word.

There are eight general purpose registers, MØ to M7 with length equal to the side of the PE array. These may hold data or addresses used in instruction fetching or execution. The register contents may be transferred to or from the array along the row or column highways.

The normal control transfer instructions GOTO, LINK and EXIT and facilities for address arithmetic are provided.

The MCU also contains the store and control interfaces (not shown in Figure 2) to the host. When the DAP is processing, the host can still access the store by cycle stealing.

C. Modes of Storage and Processing

There are two formats in which data are held in the DAP store. In the 'vertical' format each number is held entirely within one PE with successive bits in successive store locations. In the 'horizontal' format each number is spread along a row of PEs, in a manner similar to the storage of DAP instructions. Words as seen by the host are also in this format.

The most powerful processing mode, known as matrix mode, operates on data in vertical format.

Vector mode processing is used where problem parallelism is lower but makes less effective use of the PEs. This operates on data in horizontal format and makes use of instructions which allow carrys to ripple along each row.

Small amounts of scalar processing may be done in the array to avoid the overhead of returning to the host. This uses the horizontal format and is faster than vector processing since more use is made of data dependent jumps and advantage is taken of any parallelism within the arithmetic operations.

Transformations between horizontal and vertical format are done by DAP processing and take less time than a multiply operation. Thus the overhead is normally negligible.

II. ARITHMETIC AND BASIC OPERATIONS

Since the PEs are bit organised, arithmetic is built up by low level software. This software is continually being improved and is now substantially faster than a year ago. Table 1 gives the times for some operations in standard ICL 2900/IBM 360 floating point format, based on a 200 nSec cycle time as in the pilot 32 x 32 DAP. Unbiased rounding is used for greater accuracy than truncation towards zero. The operations are on matrices or vectors which match the DAP size.

For the operations in Table 1 both operands and the result have the same rank. In cases where the ranks are different the operation may be significantly faster, as indicated by the examples in Table 2.

The time for multiplication of every element of a matrix by a scalar is strongly dependent on the value of the scalar and on whether it is known at compile time. It is always significantly less than the time to multiply corresponding elements of two matrices.

Surprisingly, the sum of all elements of a matrix can be computed in a time only slightly longer than to add two matrices. The method involves converting the array to block

TABLE 1

Some Operation Times for the Pilot DAP in μSec.
Estimates for a 64 x 64 DAP are in Brackets.

Operation	Matrix	Vector	Scalar
Floating Point (32 bit):			
Z←X+Y	148(135)	54(32)	27(21)
Z←X*Y	305(250)	50(45)	34(27)
Z←X/Y	390(330)	100(90)	
Z←X**2	155(125)	40(35)	
Z←SQRT(X)	215(180)		
Z←X	14(13)	1	
Z←MAX(X,Y)	34(33)		
Fixed Point (32 bit):			
Z←X+Y	23(22)	4	

TABLE 2

Some Times in μSec for 32-bit Floating Point Mixed
Rank Operations. X and Y are Matrices; S is a Scalar

Operation	Time	
Y←X*S	40–150	(35–125)
S←ΣX	170	(160)
S←MAX(X)	46	(45)

floating point, enabling the ten stages of addition to be done essentially as fixed point operations. At each stage the number of partial results is halved. During each of the first five stages, each partial result field is split into two fields such that twice as many PEs are devoted to each partial sum. The last five stages, with the partial results in horizontal format, are somewhat similar to vector mode fixed point additions except that a carry save technique is used. Finally the scalar result is normalised. Sufficient guard bits are used that both the worst case and average rounding errors are less than if floating point were used throughout.

A bit level algorithm is used to find the maximum element of an array.

The DAP offers complete flexibility of function so the time taken by a function depends only on its complexity. Some important consequences of the flexibility are:

(a) More complex functions such as square root and logarithm are faster in comparison with basic arithmetic operations

than on a conventional computer.

(b) The DAP can take advantage of symmetries within a function so that, for example, the time for the squaring operation is about half that for general multiplication.

(c) Operations that are inherently simple may be several orders of magnitude faster than on conventional computers. For example, to replace every element of a matrix by its modulus takes less than 1 μSec for the whole matrix. Also, data dependent jumps in conventional programs are usually implemented on the DAP as conditional assignments of the result of an arithmetic operation. Thus the jumps are essentially replaced by setting the activity; this takes negligible time and the routines in the tables all incorporate activity control. Global tests, resulting in a branch if all elements of a boolean matrix are true, take less than 1 μSec.

The complete flexibility of precision of data stored in vertical mode gives continuous space and speed trade-offs. Arithmetic function times vary approximately linearly with precision: multiplication more sharply, addition less sharply.

The flexibility of representation allows radix 2 or radix 4 floating point representations to be used with consequent improvements in the arithmetic times. Still larger gains may often be made by using block floating point representation in which all elements of an array are normalised to a common exponent. In this case addition and subtraction are approximately four times faster.

III. SOFTWARE

The different modes of processing available in a computer system incorporating an NxN DAP are given below along with the associated storage mode and degree of parallelism:

Processing Mode	Storage Mode	Data Parallelism
Host - Scalar	Horizontal	1
DAP - Scalar	Horizontal	1
DAP - Vector	Horizontal	N
DAP - Matrix	Vertical	N^2

The key to using such a system effectively is to allow the mode appropriate to the degree of parallelism to be selected at any instant provided that the cost of switching modes is not dominant. The high level language system which is being implemented satisfies this requirement in a natural and efficient way.

A. Program Structure

A program to be run on the DAP comprises a standard FORTRAN program and a number of subprograms written in a language developed for the DAP called DAP-FORTRAN. The standard FORTRAN part is processed by the host and gives access to all the facilities of the host including input-output and fast scalar processing. When a subroutine written in DAP-FORTRAN is called, the host processor activates the DAP to process the subroutine and any further levels of DAP-FORTRAN routines. Processing of the FORTRAN part by the host computer may proceed asynchronously, including access to data held in the DAP store.

Data communication between FORTRAN and DAP-FORTRAN is by common blocks held in the DAP store. Since these are immediately accessible to both the host and DAP, no time is required for data transfer. Any changes in the format of stored data necessitated by different processing modes are performed either explicitly or implicitly by the DAP-FORTRAN code.

B. The Language DAP-FORTRAN

DAP-FORTRAN provides a good match between problem and hardware and encourages users to rethink algorithms to good effect. It is an array processing dialect of FORTRAN permitting vectors and matrices, as well as scalars, as the basic elements of an expression. Much of the explicit coding of inner loops of a program is thus eliminated with consequent advantages in code conciseness and readability.

The vectors and matrices of the language have their dimensions constrained to match directly the size of the DAP and are mapped in horizontal mode and vertical mode respectively. Implicit mode changes occur during the processing of statements when a row or column is selected from a matrix or when a vector is expanded to a matrix. Larger arrays are represented as indexed sets of vectors or matrices. When handling such arrays the code tends to expand to about the size of the equivalent FORTRAN code.

Operators are provided for arithmetic, logical and relational operations as in standard FORTRAN and they are also applied in an element-by-element manner to arrays. Standard functions provide regular mappings of arrays as well as element-by-element application of arithmetic functions such as logarithm.

Indexing operations are a powerful generalization of the conventional single element selection; in particular they permit the selection of elements by logical masks, a task particularly suited to the DAP.

Indexed assignment, where subscripts are used to select arguments for updating, is generalized in a way compatible with the generalized indexing. In particular the use of a logical mask as a subscript is directly implemented via the activity control of the DAP hardware.

C. DAP Code Translation

As well as DAP-FORTRAN, there is a macro-assembly language for the DAP in which all code run to date has been written. The DAP-FORTRAN compiler will produce target code in a form suitable for input to the assembler. This intermediate form is the assembly language itself, supplemented by a number of system macros.

After assembly the DAP subprograms, including any common blocks to be held in the DAP store, are consolidated into a contiguous block of code and data to be loaded into the DAP store. Entries from the host processor to specific DAP-FORTRAN subroutines within this block are engineered by "interface procedures" which enable the subroutines to be called by the standard FORTRAN subroutine calling mechanism. These interface procedures are generated by the DAP-FORTRAN compilation system but are executed by the host.

D. Operating System Software

The DAP makes minimal demands on the host operating system. The only special actions required are:

(a) The DAP program block is loaded into a contiguous area of the DAP store where it resides for the duration of the program (i.e. it is non-paged and locked into store).

(b) The host treats sub-blocks of the program block as data areas corresponding to FORTRAN common blocks.

(c) The host controls the DAP in a peripheral-like manner. This entails initiating processing by the DAP, after setting up store protection registers in the DAP, and servicing interrupts generated by the DAP when it stops processing.

IV. ALGORITHMS IMPLEMENTED ON THE PILOT DAP

A. Matrix Multiply

The usual method of calculating matrix products is to compute each element in turn as the inner product of a row of one operand with a column of the other. However the

method implemented calculates all elements in parallel by using outer products.

The DAP-FORTRAN code for multiplying 32x32 matrices A and B to produce result C on a 32x32 DAP is:

```
          C = 0.0
          DO 100 I = 1,32
      100 C = C + A( ,*I) * B(*I, )
```

The expression A(,*I) selects column I of matrix A and forms a new matrix each of whose columns is equal to this column. At the hardware level this is done by extracting and re-broadcasting each bit of the numbers in turn using the row highways. Similarly row I of matrix B is extracted and broadcast using the column highways. These and all other non-arithmetic operations account for only about 10% of the total time. The two temporary matrices are multiplied element by element and added to the partially accumulated result. It is interesting to note that this method makes no use of the nearest neighbour connections.

Performance details are given in section 4.5.

B. Matrix Inversion

The matrix inversion program run on the DAP uses the algorithm of Gauss Jordan (i.e. complete elimination above and below the pivot elements in a single forward sweep). In each of the N steps required to transform an NxN matrix into its inverse the elements of the new value A' are given in terms of the current value A by the equations:

$$A'_{pq} = \frac{1}{A_{pq}}; \qquad A'_{ij} = A_{ij} - \frac{A_{iq}A_{pj}}{A_{pq}} \quad \text{for } i \neq p \text{ and } j \neq q$$

$$A'_{pj} = \frac{A_{pj}}{A_{pq}} \quad \text{for } j \neq q; \qquad A'_{iq} = -\frac{A_{iq}}{A_{pq}} \quad \text{for } i \neq p$$

where p and q are the numbers of the pivot row and column respectively.

The DAP-FORTRAN code to invert a 32x32 matrix on a 32x32 DAP, along with some explanation, is given below. It minimizes the number of arithmetic operations since these account for most of the execution time. At each step the pivot element is selected from all the remaining elements. All row and column interchanges are left until the end.

DAP-FORTRAN Code for Matrix Inversion with Pivoting

```
01    SUBROUTINE INVP(A)
02    REAL A(,),B(,)
03    LOGICAL PROW(,),PCOL(,),PMASK(,),PIV(,),MASK(,),PIVS(,)
04    INTEGER RN( )
05    MASK=.TRUE.
06    PIVS=.FALSE.
07    DO 1 K=1,32
08    PIV=FRST(MAXP(ABS(A),MASK))
09    S=A(PIV)
10    PIVS=PIVS.OR.PIV
11    PROW=MATC(ORC(PIV))
12    PCOL=MATR(ORR(PIV))
13    PMASK=.NOT.(PROW.OR.PCOL)
14    A(PIV)=1.0
15    A=MERGE(A,0.0,PMASK)-A(,*PCOL)*MATR(A(PROW,)/S)
16    A(PROW)=-A
17 1  MASK=MASK.AND.PMASK
18 C  RESHUFFLE ROWS AND COLUMNS
19    RN=ROWN(PIVS)
20    DO 2 K=1,32
21 2  B(K,)=A(RN(K),)
22    DO 3 K=1,32
23 3  A(,RN(K))=B(,K)
24    RETURN
25    END
```

Line 2 declares A, the matrix to be inverted, and B to be 32x32 matrices of 32-bit floating point numbers. Line 3 declares a number of 32x32 matrices with logical elements. Line 4 declares RN to be a 32-element vector of 32-bit integers.

In lines 5 and 6 the elements of MASK and PIVS are all set to TRUE and FALSE respectively.

Line 7 specifies the loop control in a way similar to FORTRAN.

Lines 8 and 9 select a pivot element equal to the element of the array with largest modulus, setting the scalar S equal to the value of the element and the logical matrix PIV equal to FALSE for every element except the one in the pivot row and column. When selecting the pivot element, only elements for which the corresponding element of MASK have the value TRUE are considered.

PIVS records the positions of all the pivot elements and is used later in the reshuffling of rows and columns. Line 10 sets the element of PIVS to TRUE at the position of the current pivot element.

The elements of PROW and PCOL are set in lines 11 and 12 to have the value TRUE if and only if they are in the pivot row and pivot column respectively. The elements of PMASK are set in line 13 to have the value TRUE if and only if they are in neither the pivot row nor the pivot column.

Line 14 sets the pivot element equal to one. Line 15 contains the arithmetic operations of the loop and accounts for most of the execution time. The standard function MERGE with arguments A, 0.0 and PMASK produces a matrix whose elements are the same as the elements of A where PMASK has the value TRUE, and 0.0 elsewhere. The indexing operation A(,*PCOL) produces a matrix with every column equal to the value of the pivot column of A. The expression MATR(A(PROW,)/S) produces a matrix with every row equal to the value of the pivot row of A divided by S.

Line 16 negates the values of A in the pivot row only. This is a trivial operation for the DAP and has negligible execution time.

The last statement of the loop in line 17 sets the elements of MASK in the pivot row and pivot column to FALSE so that subsequent pivot elements are not selected from this row or column.

Lines 18 to 25 perform the reshuffling of rows and columns.

C. Fourier Transformation

The Fast Fourier Transform (FFT) of an initial set of N points involves $\log_2(N)$ steps each of which is as follows:

(a) Divide each set into two equal sets;
(b) Interchange adjacent sets in pairs;
(c) Multiply by a factor dependent on set number;
(d) Add to data at beginning of step.

For a DAP implementation with one data point per PE interchanging the data sets involves shifting using the neighbour connections. The algorithm considers the data as a linear chain, but advantage is taken of the two-dimensional connectivity of the DAP to reduce the amount of routing.

Consider a 1024 point complex transform on a 32x32 DAP. In step 1, the interchange is done in a single shift by a distance of 16 PEs making use of a cyclic connection at the array edges. The multipliers are (1,-1) so it is only necessary to negate half the array which can be done very rapidly. In step 2 there are two shifts by 8 PEs in opposite directions with the results being merged. The multipliers are (1,-1,i,-i) which are implemented as conditional negation and conditional interchange of real and imaginary parts. Subsequent multiplications require arithmetic work. For steps 6 to 10, the routing is similar to that for steps 1 to 5

respectively but is in the orthogonal direction.

Further routing is needed to arrange the results in a natural order. However, in many applications, such as convolution, the forward transform is followed ultimately by a backward transform, making the intermediate order irrelevant.

The alogrithm implemented uses a standard radix four technique to reduce the arithmetic compared with the basic description given above. Data routing accounts for about 10% of the overall time (or 20% with re-ordering).

D. Convolution using Fermat Number Transforms

It is well known that a fast convolution may be achieved by multiplying the Fourier Transforms of the operand vectors term by term and taking the inverse transform of the result. There are however other transforms which can be used in this way and one which is particularly attractive for the DAP is based on the use of Fermat Numbers (2). In this, addition and multiplication in the field of complex numbers, as in the FFT, are replaced by addition and multiplication modulo F_t in the ring of integers, where F_t is the Fermat Number $2^{2^t}+1$. The advantage is that for suitable choices of t and the number of points N, all the multiplier factors have very simple binary representation; in almost all cases the multiplies are implemented simply as cyclic shifts. The DAP takes advantage of this at the bit level and hence achieves very fast transforms. Since integers are used throughout, the convolution obtained is exact, in contrast to the FFT method which is subject to rounding error.

The code implemented carries out cyclic 2-D convolution on 32x32 points using t=4, i.e. $F_t = 65537$, so all numbers occupy 17 bits.

E. Performance Summary

The measured times for routines written in the assembly language on the pilot 32 x32 DAP are as follows:

Routine	Time on Pilot
Matrix Multiply: 32-bit floating point (32x32)	16 mSec.
Matrix Inversion: 32-bit floating point with full pivoting (32x32)	29 mSec.
Fast Fourier Transform: 32-bit floating point including re-ordering (1024 point complex)	14 mSec.
Convolution: 16-bit integer (32x32)	4.3 mSec.

These times can be reduced by:

(a) improving standard functions;
(b) making functions more specific to the application;
(c) using an internal number representation better suited to the DAP;
(d) improving the algorithms.

A DAP-FORTRAN implementation should only be slightly slower than the corresponding assembly language implementation since most of the time is spent in arithmetic routines rather than in organisational work.

V. APPLICATIONS

Most problems, especially large ones, have a lot of parallelism - usually much more than is apparent at first. Mapping of applications on to the DAP is best done at the problem level and with re-appraisal of the algorithms used; it is less effective to look at sections of an existing program in isolation. In nearly every case studied, almost all the computation can be written efficiently in DAP-FORTRAN and executed in the DAP. Performance estimates, derived from detailed studies, for a number of complete calculations are listed below. The comparisons are with existing implementations on particular machines.

Application	Estimated Performance (64x64 DAP)
Finite Element Analysis	2-6 x 360/195
Simple relaxation	20 x 360/195 5 x ILLIAC IV
Meteorology - complete suite of operational programs	13 x 360/195
3-D Magnetohydrodynamics	14-30 x 360/91
Many Body Simulation (Galactic Simulation)	10 x 7600
A data re-organisation problem	10 x 7600
A table look-up problem	3 x CRAY 1
A pattern matching problem	300 x 360/195
Operations Research (The Assignment Problem)	1200 x 370/145

Much of Finite Element Analysis involves matrix manipulations which are well suited to the DAP but the performance on other parts is less clear. Overall the performance is expected to be good, particularly on large problems.

The simple relaxation calculation (3) is an elementary method for solving Laplace's equation on a 64x64 grid. Reference 3 gives the code in CFD for ILLIAC IV and in FORTRAN for the 360/195 and gives times for 10 iterations as 17.2 and

67.2 mSec respectively. The problem can be programmed easily in DAP-FORTRAN, with each iteration needing only two additions and a multiplication by 0.25. Thus 10 iterations would take 3.3 mSec using standard floating point. On the DAP, fixed boundary conditions on an arbitrarily shaped area are dealt with by using the activity control, with no increase in computation time. Further dramatic performance improvements are possible by adopting block floating point, and by using an "average" routine to combine the addition and multiplication. The next three problems are essentially more complex grid calculations. All these are very natural for the DAP.

The Table Look Up problem studied was used recently by the Los Alamos Scientific Laboratory to measure the scalar performance of CRAY 1 (4). Conditional operations and table look-up caused the code to be regarded as "scalar" despite the fact that it is used to process arrays. Logarithm and exponential functions are also involved. Table look-up in this context means performing a number of indexing operations in parallel on a single array. In the DAP a matrix of indices is used to produce a matrix of corresponding selected elements. Successive elements of the table are written selectively to the result matrix, using as a mask a different boolean matrix for each table element. Each boolean matrix is formed by testing for equality between the index of the table element and the matrix of indices. Global tests on the indices are used to reduce the amount of work. Also, regularities of this problem allow four result matrices to be generated in one pass through the table.

The last two applications in the table are non-scientific and achieve outstanding performance because they involve mainly boolean operations.

A general observation is that the DAP performs well on many tasks which might appear to be unsuitable for parallel processing. This is a consequence of the flexibility of a bit organized array processor. Our studies, many of which are complete problems, indicate that virtually all the computation can be done effectively in the DAP and hence there is no necessity for a powerful host.

Applications studied so far cover only a subset of all computing, but it is expected that high performance will be obtained on a wide range of CPU bound activities.

VI. REFERENCES

(1) Reddaway, S.F., "DAP - A Distributed Array Processor", First Annual Symposium on Computer Architecture, Florida, December 1973.

(2) Agarwal, R.C., Burrus, C.S., "Fast Convolution Using Fermat Number Transforms with Application to Digital Filtering", Vol. ASSP-2, No.2, pp 87-97, IEEE Transactions on Acoustics, Speech and Signal Processing, April 1974.

(3) Walkden, F., McIntyre, H.A.J., Laws, G.T., "A User's View of Parallel Processors", CERN School of Computing, 1976.

(4) Keller, T.W., "Report of the CRAY-1 Evaluation", LA-6456 MS, Los Alamos Scientific Laboratory, 1976.

(5) Flanders, P.M., Hunt, D.J., Parkinson, D., Reddaway, S.F.. "Experience Gained in Programming the Pilot DAP, A Parallel Processor with 1024 Processing Elements", IMACS (AICA)-GI-Symposium on Parallel Computers - Parallel Mathematics, Munich, March 1977.

VII. ACKNOWLEDGEMENTS

ICL wish to thank the Department of Industry for permission to publish this information, which results from work which was supported by a normal cost-sharing contract by the Department's Advanced Computer Technology Project.

The many contributions by other members of the DAP project are gratefully acknowledged.