AD A031483

RADC-TR-76-244, Chapters 1 - 4
Final Technical Report
August 1976

ANALYSIS AND DESIGN OF REAL-TIME PROCESSING SYSTEM FOR SENSOR DATA AT AMOS

W. W. Gaertner Research Inc.

Sponsored by
Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 2646

Approved for public release;
distribution unlimited.

C007 789 20 Oct

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

DDC
RECEIVED
NOV 2 1976
D

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

Edward C Mahen

EDWARD C. MAHEN, Capt, USAF
Project Engineer

Do not return this copy. Retain or destroy.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and or SPECIAL
A

ANALYSIS AND DESIGN OF REAL-TIME PROCESSING SYSTEM FOR SENSOR DATA AT AMOS

W. W. Gaertner
M. P. Patel
S. S. Reddi
C. T. Retter
W. M. Schreyer
I. M. Singh

Contractor: W. W. Gaertner Research Inc.
Contract No: F30602-75-C-0180
Effective Date of Contract: 1 March 1975
Contract Expiration Date: 31 March 1976
Short Title of Work: Real-Time Sensor Data Processing of AMOS
Program Code Number: 5E20
Period of Work Covered: Mar 75 - Mar 76

Principal Investigator: W. W. Gaertner
Phone: 203 322-3673
Project Engineer: Capt Edward C. Mahen
Phone: 315 330-3145

Approved for public release;
distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Capt Edward C. Mahen (OCSE), Griffiss AFB NY 13441 under Contract F30602-75-C-0180.

DDC
RECEIVED
NOV 2 1976
D

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-76-244, Chapters 1 - 4 (Ch-1/4)	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ANALYSIS AND DESIGN OF REAL-TIME PROCESSING SYSTEM FOR SENSOR DATA AT AMOS	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 1 Mar 75 - 31 Mar 76
	6. PERFORMING ORG. REPORT NUMBER: GRI-4047-FR-Ch-1/4
7. AUTHOR(s): W. W. Gaertner, M. P. Patel, S. S. Reddi, C. T. Retter, W. M. Schreyer, I. M. Singh	8. CONTRACT OR GRANT NUMBER(s): F30602-75-C-0180, ARPA Order-2646
9. PERFORMING ORGANIZATION NAME AND ADDRESS: W. W. Gaertner Research Inc., 205 Saddle Hill Road, Stamford CT 06903	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62301E, 26460305
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington VA 22209	12. REPORT DATE: August 1976
	13. NUMBER OF PAGES: 278 (303 p.)
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Rome Air Development Center (OCSE), Griffiss AFB NY 13441	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Capt Edward C. Mahen (OCSE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel Processing, Image Processing, Photometric Sensors, Infrared Sensors, Space-Object Identification, Computer Architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report deals with the design of a computer system for the real-time processing of high-resolution video, photometric and long-wavelength infrared data. An analysis of the data rates generated by the various sensors and the algorithms involved in the subsequent data analysis, indicates that the combined processing requirements are in the order of 100 MIPS (million instructions per second), and that up to 16 Mbytes of high-speed memory may be needed. This processing power cannot be achieved with a sequential computer, and a parallel-processor architecture was therefore developed to achieve the necessary throughput. It

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

388 544

6pg

has a very attractive cost/performance ratio and appears to be useful in a number of other application areas such as radar, sonar, data-base search and update, mapping, solution of partial differential equations and simultaneous linear equations.

The conclusion is drawn that the G-471 is a computer system which, for the signal and image processing fields, provides a processing-power/cost ratio which is at least an order of magnitude higher than that of large-scale sequential scientific computers. It is well suited to handle the near-real time processing requirements at AMOS.

- TABLE OF CONTENTS -

Chapter No.		Page No.
1.	INTRODUCTION	1
2.	THE G-471 HARDWARE DESIGN	3
2.1	Control Computer Complex	6
2.1.1	The Control Computer CPU	6
2.1.2	Processing Element Array Controller (PEAC)	8
2.1.2.1	PEAC Control Registers	8
2.1.2.1.1	PEAC Input Status Register (PIS)	8
2.1.2.1.2	PEAC Input Data Register (PID)	10
2.1.2.1.3	PEAC Output Status Register (POS)	10
2.1.2.1.4	PEAC Output Data Register (POD)	11
2.1.2.1.5	PEAC Output Mask Registers (POMA & POMB)	11
2.1.2.2	PEAC Operation	11
2.1.3	CC Mass Storage	15
2.1.4	Other CC Peripherals	16
2.1.5	DREA Decoder Control Bus	17
2.1.6	The CC Mapper	19
2.2	The Processing Element Array	20
2.2.1	The Processing Element (PE)	20
2.2.1.1	The Processing Element Microprocessor	24
2.2.1.2	The PE Address Mapper	25
2.2.1.2.1	Address Mapping	25
2.2.1.2.2	Mode Conversion	27
2.2.1.2.3	Parity Detection/Insertion	29
2.2.1.2.4	Synchronization Hardware	29
2.2.1.3	PE Bus Control	31
2.2.1.4	The PE Channel	35
2.2.1.4.1	PE Channel Control Registers	35
2.2.1.4.1.1	Control and Status Register (CSR)	35
2.2.1.4.1.2	Word Count Register (WCTR)	37
2.2.1.4.1.3	Input Address Registers (IARL & IARH)	37

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
2.2.1.4.1.4	Input Increment Register (IIR)	37
2.2.1.4.1.5	Output Address Registers (OARL & OARH)	37
2.2.1.4.1.6	Output Increment Register (OIR)	37
2.2.1.4.2	PE Channel Operation	38
2.2.1.5	CC Communication Port	41
2.2.1.5.1	CC Communication Port Control Registers	41
2.2.1.5.1.1	CC Input Status Register (CCIS)	41
2.2.1.5.1.2	CC Input Data Register (CCID)	41
2.2.1.5.1.3	CC Output Status Register (CCOS)	41
2.2.1.5.2	CC Communication Port Operation	43
2.2.1.6	Processing Element Local Storage (PELS)	46
2.2.1.7	The Arithmetic Processor (AP)	49
2.2.1.7.1	Central Arithmetic Processing Unit (CAPU)	52
2.2.1.7.1.1	Functional Description	52
2.2.1.7.1.1.1	Program Sequencer	52
2.2.1.7.1.1.2	Data Flow	54
2.2.1.7.1.1.2.1	Inputs	54
2.2.1.7.1.1.2.2	Multiplication	56
2.2.1.7.1.1.2.3	Division	56
2.2.1.7.1.1.2.4	Addition/Subtraction	56
2.2.1.7.1.1.2.5	Special Functions	57
2.2.1.7.1.1.2.6	Register File (RF)	57
2.2.1.7.1.1.2.7	ALU	57
2.2.1.7.1.1.2.8	Outputs	58
2.2.1.7.1.2	CAPU Instruction Format	59
2.2.1.7.1.3	CAPU Algorithms	65
2.2.1.7.1.3.1	Division	65
2.2.1.7.1.3.2	Floating-Point to Integer Conversion and Vice Versa	66
2.2.1.7.1.3.3	Square Root	68
2.2.1.7.1.3.4	Natural Logarithm	69

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
2.2.1.7.1.3.5	Exponential	69
2.2.1.7.1.4	Floating-Point Multiplier	74
2.2.1.7.1.5	Floating-Point Adder	77
2.2.1.7.1.5.1	Functional Description	77
2.2.1.7.1.5.2	Floating Point Adder - Propagation Delays	79
2.2.1.7.2	Input-Output Address Generator (IOAG)	80
2.2.1.7.2.1	The Programmable IOAG - Functional Description	81
2.2.1.7.2.1.1	Program Sequencer	81
2.2.1.7.2.1.2	Data Flow	83
2.2.1.7.2.1.2.1	IOAG Data Flow Section Operation	85
2.2.1.7.2.1.2.2	Registers	85
2.2.1.7.2.1.2.3	Shifting	85
2.2.1.7.2.1.2.4	ALU	85
2.2.1.7.2.1.2.5	Bit Reversal	85
2.2.1.7.2.1.2.6	Outputs	86
2.2.1.7.2.1.3	IOAG Instruction Format	87
2.2.1.7.2.2	Special-Purpose Input/Output Address Generator (IOAG)	92
2.2.1.7.3	AP FIFO's	103
2.2.1.7.3.1	FIFO Operation	103
2.2.1.7.4	Data Fetch Unit and Data Store Unit	106
2.2.1.7.4.1	DFU Operation	106
2.2.1.7.4.2	DSU Operation	108
2.2.1.7.5	AP Bus Control	109
2.2.1.7.5.1	Operation of AP Bus Control	109
2.2.1.7.6	AP Controller	111
2.2.1.7.6.1	CAPU Control	111

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
2.2.1.7.6.2	IOAG Control	114
2.2.1.7.6.3	FIFO Status	117
2.2.1.7.7	Multimode Shifter	119
2.3	The Central Working Storage (CWS)	121
2.3.1	Organization	121
2.3.2	Layout of CWS Bank	121
2.3.3	CWS Operation	123
2.3.4	DREA - CWS Interface Signals	125
2.4	Data Routing Element Array	126
2.4.1	Crossbar Switch Array	128
2.4.2	Memory Address Decoder	132
2.4.3	Arbiter	134
2.4.4	DREA System Timing	137
2.5	Mass Storage System	140
2.5.1	The Mass Storage Device (MSD)	140
2.5.2	The Mass Storage Adapter (MSA)	140
2.6	Real Time Input-Output	144
2.7	Packaging, Cost and Power Consumption	145
2.7.1	Packaging	145
2.7.2	Cost and Power Consumption	145
2.7.2.1	Control Computer Complex	147
2.7.2.2	Processing Element	147
2.7.2.3	PELS	147
2.7.2.4	CWS	149
2.7.2.5	Data Routing Element Array	149
2.7.2.6	Mass Storage	151
2.7.2.7	Real Time I/O	151
2.7.3	Calculation of Total Cost	153
2.7.3.1	Cost of a Typical System	153

- TABLE OF CONTENTS -
(continued)

Chapter No.		Page No.
3.	SOFTWARE	154
3.1	The Multi-Processor Operating System (MPOS/471)	154
3.2	System Modules	156
3.2.1	The Scheduler and Dispatcher	157
3.2.2	The Process Manager	157
3.2.3	CC Disk Handler and File Manager	157
3.2.4	CC Memory Manager	158
3.2.5	Drivers for Various Peripherals	158
3.2.6	Command Processor	158
3.2.7	Terminal Handler	158
3.2.8	PE Array Control and Communication Package (PEACCP)	159
3.2.8.1	The Processing Element Allocation Manager (PEAM)	160
3.2.8.1.1	PEAM Routines	160
3.2.8.1.1.1	INPEAM (PELST)	160
3.2.8.1.1.2	ALPE (N, PELST)	160
3.2.8.1.1.3	DEALPE (PELST)	160
3.2.8.1.1.4	CATPEA (LIST)	161
3.2.8.1.1.5	DISAB (PEN)	161
3.2.8.1.1.6	ENAB (PEN)	161
3.2.8.2	The Processing Element Task Dispatcher (PETD)	162
3.2.8.2.1	Processing Element Task List (PETL)	163
3.2.8.2.2	PETD Routines	166
3.2.8.2.2.1	PETLNO = INPETL (PELST, BUFAD, BUFLEN)	166
3.2.8.2.2.2	ADDENT (PETLNO, RTN, PAR)	166
3.2.8.2.2.3	DISPATCH (PESLT, PETLNO, PTLPTR)	166
3.2.8.2.2.4	RELPTL (PETLNO)	166
3.2.8.2.2.5	PETDJOIN(PETLNO,CTR)	166
3.2.8.2.2.6	PETDWAIT (PETLNO, CTR)	167
3.2.8.2.2.7	PETDDNSG (PETLNO, CTR)	167

- TABLE OF CONTENTS -
(continued)

Chapter No.		Page No.
3.2.8.3	The PEAC Driver	168
3.2.8.3.1	PEAC Driver Routines	168
3.2.8.3.1.1	DISCON (PEN)	168
3.2.8.3.1.2	CON (PEN)	168
3.2.8.3.1.3	RUN (PEN, ADR)	168
3.2.8.3.1.4	HALT (PEN)	168
3.2.8.3.1.5	SEND (PEN, BUFAD, BUFLEN)	169
3.2.8.3.1.6	BCAST (PELST, BUFAD, BUFLEN)	169
3.2.8.3.1.7	RECV (PEN, BUFAD, BUFLEN)	169
3.2.8.3.1.8	RECWDR (PEN, NAVAIL)	169
3.2.8.3.1.9	PESIG (PEN, PROC)	169
3.2.8.3.1.10	LOADPE (PEN, BUFAD, BUFLEN, RELOC)	169
3.2.8.3.1.11	LOADBC (PELST, BUFAD, BUFLEN, RELOC)	169
3.2.8.3.1.12	RECLIN (PEN, BUFAD, BUFLEN, BRKCH, RCNT)	170
3.2.8.4	Inter-PE Communication and Synchronization Monitor (IPECSM)	171
3.2.8.4.1	IPECSM Message Format	171
3.2.8.4.2	IPECSM Routines	173
3.2.8.4.2.1	SETCOM (PELST)	173
3.2.8.4.2.2	SENDSIG (PEN, SIGNO)	173
3.2.8.4.2.3	SENDMES (PEN, MESBUF, MESL)	173
3.2.8.4.2.4	RECSGR (PEN, SIGNO, PROC)	173
3.2.8.4.2.5	RECSIG (PEN, SIGNO)	173
3.2.8.4.2.6	RECMSR (PEN, BUFAD, LEN, PROC)	173
3.2.8.4.2.7	RECMSG (PEN, BUFAD, LEN)	174
3.2.8.4.2.8	EXEC (PEN, STADDR)	174
3.2.8.4.2.9	EXECBC (PELST, ADDR)	174
3.2.9	The Central Working Storage Manager (CWSM)	175
3.2.9.1	EXBANK (I, PELST)	175
3.2.9.2	FRBANK (I)	175
3.2.9.3	ALCBLK (SIZE, LOC, BNK)	175
3.2.9.4	DEABLK (ADDR)	176
3.2.10	Mass Storage Access System (MSAS)	177
3.2.10.1	MSAS Routines	177
3.2.10.1.1	MCREAT (NAME, MODE, SIZE)	177

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
3.2.10.1.2	MOPEN (NAME, MODE)	178
3.2.10.1.3	MCLOSE (FILDES)	178
3.2.10.1.4	MUNLINK (NAME)	178
3.2.10.1.5	MREAD (FILDES, BUF, NBYTES)	178
3.2.10.1.6	MREADR (FILDES, BUF, NBYTES)	179
3.2.10.1.7	MREADF (FILDES, BUF, NBYTES, FUNC)	179
3.2.10.1.8	MWRITE (FILDES, BUF, NBYTES)	179
3.2.10.1.9	MWRITR (FILDES, BUF, NBYTES)	179
3.2.10.1.10	MWRITE (FILDES, BUF, NBYTES)	179
3.2.10.1.11	MWAIT (FILDES)	179
3.2.10.1.12	MSEEK (FILDES, OFFSET, MODE)	179
3.2.10.1.13	MRDC (FILDES, BUF, NBYTES) MRDR (FILDES, BUF, NBYTES) MRDF (FILDES, BUF, NBYTES, FUNC) MWRT (FILDES, BUF, NBYTES) MWRTR (FILDES, BUF, NBYTES) MWRTF (FILDES, BUF, NBYTES, FUNC)	180
3.2.11	PE/CWS Simulator	181
3.2.12	The CAPU Cross-Assembler	182
3.2.12.1	Introduction to CAPU Cross-Assembler	182
3.2.12.2	CAPU Cross-Assembler Input Format	182
3.2.13	The IOAG Cross Assembler	198
3.2.13.1	Introduction to IOAG Cross Assembler	198
3.2.13.2	Input Format	198
3.2.14	The Processing Element Resident Monitor	209
3.2.14.1	The PE Bootstrap	209
3.2.14.2	System Communication Handler (SCH)	210
3.2.14.2.1	PSENDWD (WRD)	210
3.2.14.2.2	PSENDCC (BUFAD, BUFLEN)	210
3.2.14.2.3	PSENDPE (BUFAD, BUFLEN, PEN)	210
3.2.14.2.4	PSIGCC (SIGNO)	210
3.2.14.2.5	PSIGPE (SIGNO, PEN)	210

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
3.2.14.2.6	PRECV (BUFAD, BUFLEN, LEN, SRC, FUNC)	210
3.2.14.2.7	PRCWDR (WD, AVAIL)	211
3.2.14.2.8	PINSIG (FUNC)	211
3.2.14.2.9	PDISP (PETLPTR)	211
3.2.14.2.10	PDEXIT ( )	212
3.2.14.2.11	LUPBAK (PTLB)	212
3.2.14.2.12	RETRY ( )	212
3.2.14.2.13	DUMMY ( )	212
3.2.14.2.14	PEJOIN (CTR)	212
3.1.14.2.15	PWAIT (SEM)	213
3.1.14.2.16	PSIG (SEM)	213
3.2.14.3	The PE Channel Handler (PECH)	214
3.2.14.3.1	CHMOVE (SRCADR, SRCINC, DSTADR, DSTINC, WRDCNT, MODE)	214
3.2.14.3.2	CHWAIT ( )	215
3.2.14.3.3	CHMOVF (SRCADR, SRCINC, DSTADR, DSTINC, WRDCNT, MODE, FUNC)	215
3.2.14.3.4	CHSTAT (BUF)	215
3.2.14.4	Arithmetic Processor Control System	217
3.2.14.4.1	CAPU Control Functions	217
3.2.14.4.1.1	APRUN (STAD)	217
3.2.14.4.1.2	APHALT ( )	217
3.2.14.4.1.3	APFLAG (F)	217
3.2.14.4.1.4	APFLSET (F,VAL)	217
3.2.14.4.1.5	APSTAT ( )	218
3.2.14.4.1.6	APINT (MASK, FUNC)	218
3.2.14.4.1.7	APDIR ( )	218
3.2.14.4.1.8	APSEND (APIN)	218
3.2.14.4.1.9	APGET (APOUT)	218
3.2.14.4.1.10	APPC ( )	218
3.2.14.4.1.11	APSSTEP ( )	218
3.2.14.4.1.12	HLTERR (M)	219
3.2.14.4.2	IOAG Control Functions	220
3.2.14.4.2.1	IORUN (STAD)	220
3.2.14.4.2.2	IOHALT ( )	220
3.2.14.4.2.3	IOFLAG (F)	220
3.2.14.4.2.4	IOFLSET (F,VAL)	220

- TABLE OF CONTENTS -
(continued)

Chapter No.		Page No.
3.2.14.4.2.5	IOSTAT ( )	220
3.2.14.4.2.6	APINT (MASK, FUNC)	220
3.2.14.4.2.7	IODIR ( )	221
3.2.14.4.2.8	IOGETI (II)	221
3.2.14.4.2.9	IOGETO (IO)	221
3.2.14.4.2.10	IOPC ( )	221
3.2.14.4.2.10	IOSSTEP ( )	221
3.2.14.4.3	FIFO Control Functions	222
3.2.14.4.3.1	FISTAT ( )	222
3.2.14.4.3.2	FICONT (IF)	222
3.2.14.4.3.3	FISIG (FUNC)	222
3.2.14.5	The Mapper Handler	223
3.2.14.5.1	SEGSET (SEGNO, BASE, LEN)	223
3.2.14.5.2	SEGACT (SEGNO, VAL)	223
3.2.14.5.3	PHYSAD (AD)	223
3.2.14.5.4	LOGAD (EXTAD)	223
3.2.15	The Multiprocessor Linkage Editor (MPLE)	224
3.2.15.1	MPLE Command Format	225
3.2.16	The Multiprocessor Loader	227
3.2.17	Programs for Systems Reliability	228
3.2.17.1	Diagnostic Routines	228
3.2.17.2	System Reconfiguration	229
3.2.17.3	Error Handling	230
3.2.18	Standard Subroutine Library (SSL)	231
3.2.18.1	FFT (PELST, VI, VO, N)	232
3.2.18.2	IFFT (PELST, VI, VO, N)	232
3.2.18.3	FFT2D (PELST, AI, AO, N)	232
3.2.18.4	IFFT2D (PELST, AI, AO, N)	232
3.2.18.5	VSUM (PELST, VA, VB, VO, N)	232
3.2.18.6	VSUMI (PELST, IVA, IVB, IVO, N)	233
3.2.18.7	VSUMC (PELST, CVA, CVB, CVO, N)	233
3.2.18.8	ASUM (PESLT, AA, AB, AO, N)	233
3.2.18.9	ASUMI (PELST, IAA, IAB, IAO, N)	233
3.2.18.10	ASUMC (PELST, CAA, CAB, CAO, N)	233
3.2.18.11	VMUL (PESLT, VA, VB, VO, N)	233

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
3.2.18.12	VMULI (PELST, IVA, IVB, IVO, N)	233
3.2.18.13	VMULC (PELST, CVA, CVB, CVO, N)	233
3.2.18.14	AMUL (PELST, AA, AB, AO, N)	233
3.2.18.15	AMULI (PELST, IAA, IAB, IAO, N)	233
3.2.18.16	AMULC (PELST, CAA, CAB, CAO, N)	233
3.2.18.17	VDIV (PELST, VA, VB, VO, N)	234
3.2.18.18	VDIVI (PELST, IVA, IVB, IVO, N)	234
3.2.18.19	VDIVC (PELST, CVA, CVB, CVO, N)	234
3.2.18.20	ADIV (PELST, AA, AB, AO, N)	234
3.2.18.21	ADIVI (PELST, IAA, IAB, IAO, N)	234
3.2.18.22	ADIVC (PELST, CAA, CAB, CAO, N)	234
3.2.18.23	SQRT (PELST, VA, VO, N)	234
3.2.18.24	ASQRT (PELST, AA, AO, N)	234
3.2.18.25	VLOG (PELST, VA, VO, N)	234
3.2.18.26	ALOG (PELST, AA, AO, N)	234
3.2.18.27	VABSPH (PELST, CVI, VO, VP, N)	234
3.2.18.28	AABSPH (PELST, CAI, AVO, AVP, N)	234
3.2.18.29	MTMUL (PELST, AA, AB, AO, N)	234
3.2.18.30	MTINV (PELST, AI, AO, N)	235
3.2.18.31	VIP (PELST, VA, VB, S, N)	235
3.2.18.32	CONV (PELST, VA, VB, VO, N)	235
4.	MAINTAINABILITY AND RELIABILITY	236

- TABLE OF CONTENTS -

(continued)

Chapter No.		Page No.
APPENDIX I	- COMPUTATIONAL ASPECTS OF SIMPLE FILTERS	I-1
APPENDIX II	- TWO-DIMENSIONAL IMAGE RESTORATION AND ENHANCEMENT FILTERS	II-1
APPENDIX III	- A NOVEL IMPLEMENTATION SCHEME FOR LEAST SQUARES FILTERS	III-1
APPENDIX IV	- DERIVATION OF THE OPTIMAL FILTER IN FREQUENCY DOMAIN	IV-1

(The reverse of this page is blank).

1. INTRODUCTION

The military is faced with formidable computational problems in a number of areas such as real-time radar, sonar, video, photometric and infrared signal processing, EW, ballistic missile defense, real-time simulation, high-speed search of large data bases, digital cartography, weapon design and others. In the past these tasks have been carried out on large sequential and pipelined computers. These are expensive, and in some cases the latency of the pipeline prevents the use of such computers in real-time applications. In addition, further increases in the throughput of sequential machines are becoming more difficult to obtain because of fundamental propagation delays between logic and memory elements, and the high power dissipation and high cost of high-speed solid-state circuits.

Fortunately, many of the algorithms of interest exhibit a very substantial degree of parallelism in the sense that they allow the simultaneous execution of many computational steps or program modules. As an example, in the 2-dimensional Fast Fourier Transform (FFT) which is a common image-processing algorithm, the 1-dimensional FFT of each row or column in the image can be carried out completely independent of any other, such that by using 1000 processors one could speed up the FFT of a 1000x1000 pixel image by 3 orders of magnitude over a single sequential processor.

This parallel property of many algorithms thus allows their execution to be speeded up by the use of computer architectures with parallel processing elements. This design approach is aided by the continuing rapid drop in the cost of logic and memory elements.

The following report describes one such parallel computer architecture, the G-471, which has been designed specifically for the real-time applications listed above. Through the use of the latest available mainstream technology, it achieves a very attractive performance/cost ratio, as can be seen from the following budgetary prices:

CONFIGURATION	MIPS*	MBYTES OF 500 ns CENTRAL WORKING STORAGE	MBYTES OF DISK MASS STORAGE	$
1	20	2	300	423,000
2	50	4	300	613,000
3	80	8	600	904,000
4	128	16	900	1,320,000

*million instructions per second

The hardware design is presented in Chapter 2 and the software plan in Chapter 3. Chapter 4 contains a reliability analysis. The parallelism of several image processing algorithms is explained in the Appendices.

2. The G-471 Hardware Design

In this chapter, we describe the design of the G-471 system hardware. A block diagram of a typical configuration is shown in Figure 2.0-1.

The architecture exploits parallelism and pipelining at different levels to achieve extremely high processing rates. A system containing 20 processing elements can execute 80 million floating point multiplications and 120 million floating point additions per second in parallel with other integer and logical operations. These high processing rates are matched with a corresponding high memory bandwidth and memory capacity to truly achieve a system throughput of this magnitude.

The system contains an array of processing elements (PEs) that can communicate with a large system-wide storage facility - the Central Working Storage (CWS) through the Data Routing Element Array (DREA). The CWS is divided into up to 32 banks that can be accessed in parallel by different PEs. A typical 20 PE system has 1 to 16 Mbytes of CWS and a CWS bandwidth of 320 Mbytes/sec.

The effective bandwidth is increased further and the effect of memory conflicts is reduced by including fast bipolar memories within the PEs - the PE Local Storage (PELS). The PE has a channel which can transfer data between the PELS and CWS in parallel with other PE operations. The PELS has a cycle time of 100 nsec providing an internal memory bandwidth at this level of 80 Mbytes/sec within a PE.

Furthermore, each of the processors within a PE, the PE microprocessor, the Arithmetic Processor and the Input/Output Address Generator, have their own local memories to hold their programs. Thus the bandwidths mentioned here apply strictly to data.

The PE is organized around a microprocessor, the LSI-11, which performs the functions of control, communication, as well as computation. Well-structured parts of a computation, especially iterative floating-point operations of the kind that are usually found in inner loops of sequential programs, are executed by an Arithmetic Processor (AP) within the PE. The AP is a high-speed pipe-lined processor that operates under the control of the PE microprocessor and in parallel with it to provide the PE with its computational power. It consists of several units operating concurrently to compute operand addresses, fetch and store the operands, and perform

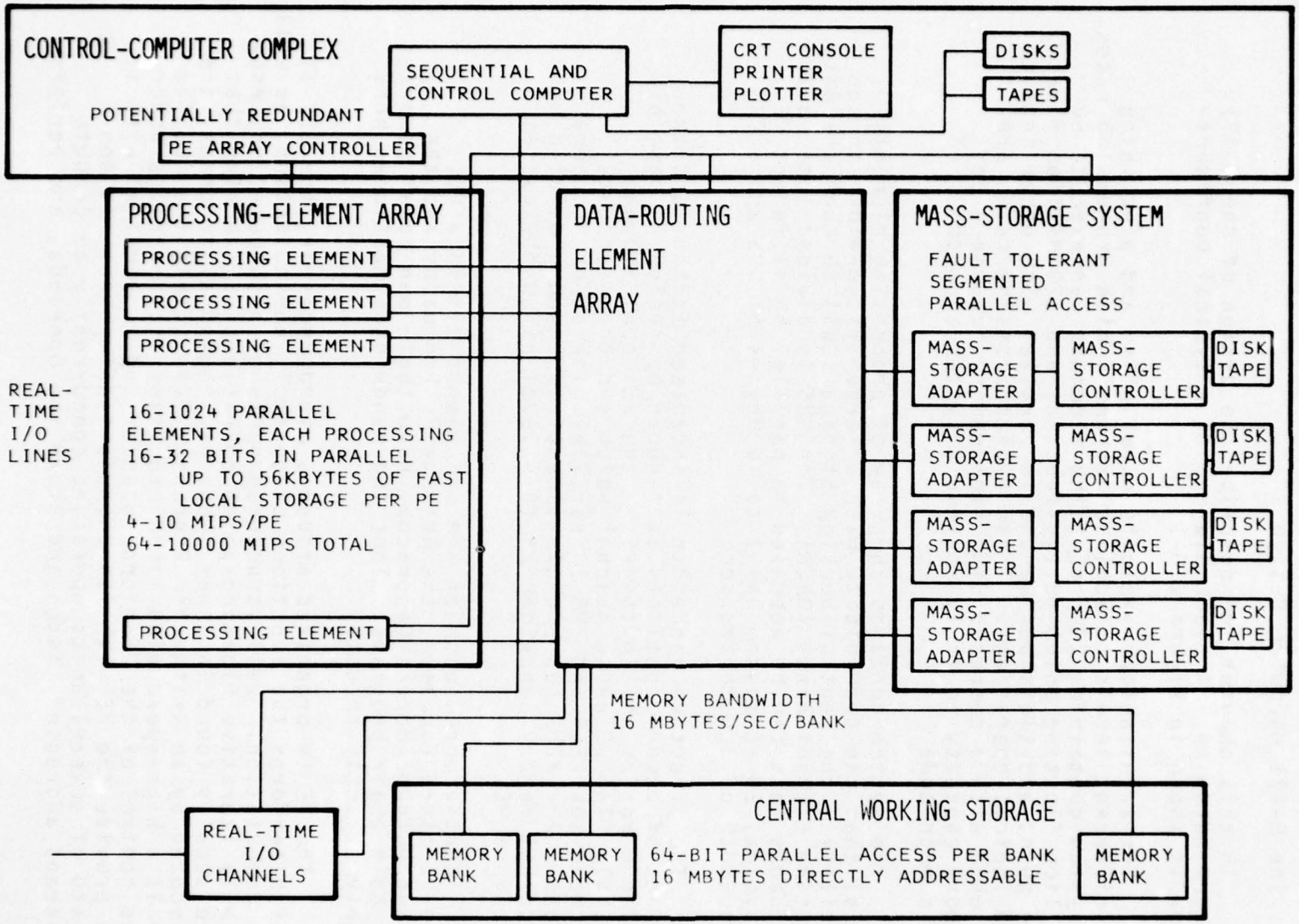


Figure 2.0-1. Block Diagram of the G-471 Parallel/Associative Computer System

floating-point, integer and logical operation.

The Mass Storage System provides on-line disk storage for large data files. The disks in the mass storage system transfer data directly to or from any part of the CWS through the DREA.

The whole system operates under the supervision of the Control Computer Complex. This is a PDP-11 system with its own complement of peripherals. It is connected to the PE Array through the Processing Element Array Controller (PEAC). It also has direct access to the CWS through a DREA port.

The whole system is designed to be highly modular and reconfigurable to achieve considerably greater reliability and availability than normal uniprocessor systems.

The individual modules that make up the G-471 system are described in detail in the following sections (Sections 2.1 through 2.6). Section 2.7 summarizes the system packaging, power consumption, cost and reliability estimates.

2.1 Control Computer Complex

The Control Computer (CC) is a minicomputer system that serves two major functions:

(i) It is utilized to control the operations of the processing element (PE) array and other hardware modules that comprise the G-471 array-processor system. The CC can control and communicate with the PEs via the PE Array Controller (PEAC - Section 2.1.2) and the shared Central Working Storage (CWS - Section 2.3). Programs running on the CC control the allocation of PEs and other system resources, sequencing and task-dispatching, system hardware reconfiguration, synchronization, inter-PE communication, and perform other functions required for controlling the multi-processor system.

Further, an application program on the G-471 typically consists of a control part that runs on the CC and implements the overall program logic along with programs that run on PEs under the control of the CC program and perform most of the actual computations.

(ii) The CC also provides various services such as program development aids including editors, compilers and a file system; interfacing to various peripherals, remote terminals and other computers; text-processing and report preparation; etc. It can also run any other kind of sequential programs concurrently with the PEs; for example a CC program could process and display the results of computations being run on the PE array.

For further details, refer to Chapter 3 which describes the system programming environment.

The various hardware modules comprising the Control Computer Complex are described next in Sections 2.1.1 through 2.1.6.

2.1.1 The Control Computer CPU

This is a standard minicomputer such as a DEC PDP-11/40 or PDP-11/45. Since this is an item that is available off-the-shelf, and the manufacturer provides adequate documentation, it will not be described here.

The CC CPU is connected to standard memory on its bus (32 to 256 kbytes) via a memory management unit. A number of standard PDP-11 peripherals are connected directly to its bus (Section 2.1.4). In addition, it is connected to the system-wide Central Working Storage via the CC mapper (Section 2.1.6), the PE Array via the PEAC (Section 2.1.2) and the system Mass Storage Disks via the Mass Storage Adapter (Section 2.5.2).

2.1.2 Processing Element Array Controller (PEAC)

The Processing Element Array Controller provides the Control Computer with full control of the processing elements (PEs). The CC can halt and restart any PE, load its memory and examine the memory and registers of the PE microprocessor. 16-bit data words can be transferred in either direction and the CC and any PE can interrupt each other, permitting very flexible synchronization schemes to be implemented. The CC can communicate with an individual PE, or in broadcast mode (BC) in which it communicates with all PEs selected by a Mask Register.

Further, the CC can physically disconnect any malfunctioning PEs from the PE array.

The PEAC is programmed similarly to other PDP-11 peripherals: a number of control and data registers appear in the address space of the CC, and they can be read and/or written in the same way as memory words.

At the PE end, the PEAC looks like a console terminal to the PE microprocessor so that the LSI-11 ODT microcode as well as the standard loader and other software can be utilized without any modifications. However, the communication is over a high-speed parallel bus unlike the usual serial connection to a console terminal.

In the following, the PEAC is described as it appears to the CC. The PE end is described under CC Communication Port (Section 2.2.1.5).

2.1.2.1 PEAC Control Registers

Figure 2.1.2.1-1 shows the registers in the CC Address Space used for programming the PEAC. The individual registers are described next.

2.1.2.1.1 PEAC Input Status Register (PIS)

Bus Address = 764000
Interrupt Vector = 300, 302

BIT	NAME	FUNCTION
12-8	PE Number	When the ready bit is set, this field gives the ID of the PE that has data for the CC. If more than one PE tries to send data to the CC simultaneously by loading its CCOD register, an arbitration network selects

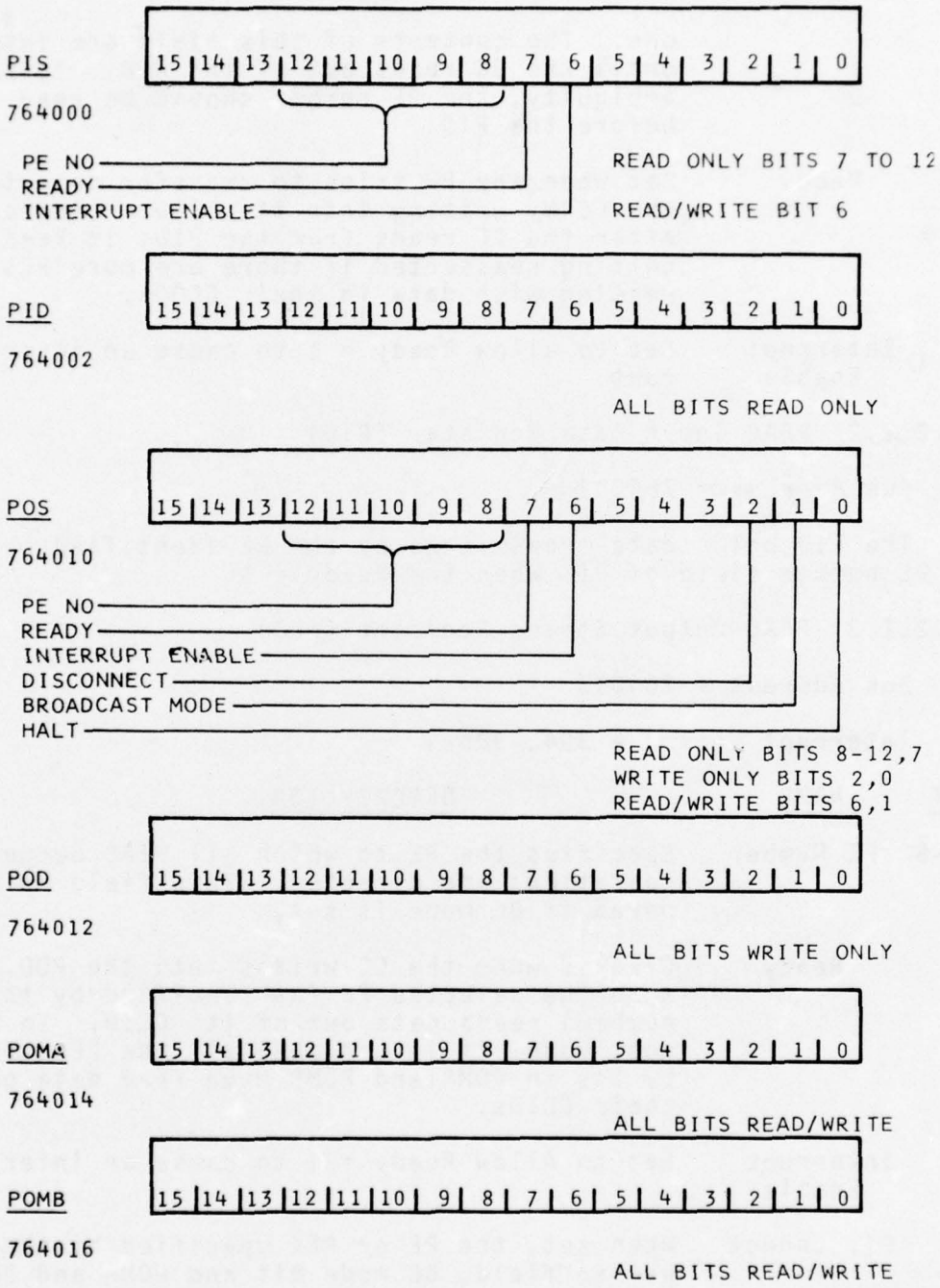


Figure 2.1.2.1-1. PEAC Control Registers

BIT	NAME	FUNCTION
		one. The contents of this field are latched until the CC reads out of the PID. To avoid ambiguity, the PE number should be read before the PID.
7	Ready	Set when any PE tries to transfer data to the CC by writing into its CCOD. Cleared after the CC reads from the PID; it keeps getting reasserted if there are more PEs waiting with data in their CCODs.
6	Interrupt Enable	Set to allow Ready = 1 to cause an interrupt.

2.1.2.1.2 PEAC Input Data Register (PID)

Bus Address = 764002

The PID holds data transmitted by the PE identified by the PE number field of PIS when the Ready = 1.

2.1.2.1.3 PEAC Output Status Register (POS)

Bus Address = 764010

Interrupt Vector = 304, 306

BIT	NAME	DESCRIPTION
12-8	PE Number	Specifies the PE to which all PEAC output operations are directed. This field is ignored if BC mode is set.
7	Ready	Cleared when the CC writes into the POD. Set when the selected PE (as specified by the PE number) reads data out of its CCID. In broadcast mode, it is set when all the PEs specified by 1's in POMA and POMB have read data out of their CCIDs.
6	Interrupt Enable	Set to Allow Ready = 1 to cause an interrupt.
2	Disconnect	When set, the PE or PEs specified by the PE number field, BC mode bit and POMA and POMB are powered down, and critical signals are physically disconnected from the rest of the system. Writing a zero in this bit turns the selected PEs back on and connects them to the rest of the system.

BIT	NAME	DESCRIPTION
1	Broadcast Mode	When set, all the PEs specified by 1's in the mask registers POMA and POMB are selected. If data is written into COD, the ready bits are set in the CCIS status registers in all the selected PEs and the Ready bit in the COS is cleared. It goes back to a 1 only when all the selected PEs have referenced the corresponding data registers. The Halt and Disconnect similarly apply to all the selected PEs. When BC Mode is cleared, only the one PE specified by the PE number field is selected for the above-mentioned operations.
Ø	Halt	Set to put the selected PEs (specified by BC = 0 and PE Number or BC = 1 and POMA and POMB) into Halt mode. In this mode, the PE microprocessor stops any running program and executes ODT microcode. By sending appropriate ASCII characters, the CC can read or write any location in the address space of the microprocessor, as well as various internal registers or direct the microprocessor to execute a program at any location.

2.1.2.1.4 PEAC Output Data Register (POD)

Bus Address = 76412

The POD register is used to transfer data to one or more PEs. When the CC writes into the POD, the Ready bit in the POS is cleared and the ready bit in the CCIS of the selected PEs is set. The data in the POD is available to the selected PEs as the CCID.

2.1.2.1.5 PEAC Output Mask Registers (POMA & POMB)

Bus Address = 776014, 776016

The mask registers are used in broadcast mode to select a subset of the PEs to be addressed. When the BC Mode bit is set in the POS, the PEs corresponding to 1's in these registers are selected for data transmission or the Halt/Restart or Connect/Disconnect operations.

2.1.2.2 PEAC Operation

A block diagram of the Processing Element Array Controller (PEAC) is shown in Figure 2.1.2.2-1. The PEAC is connected to the CC Communication Port in the PEs (Section 2.2.1.5).

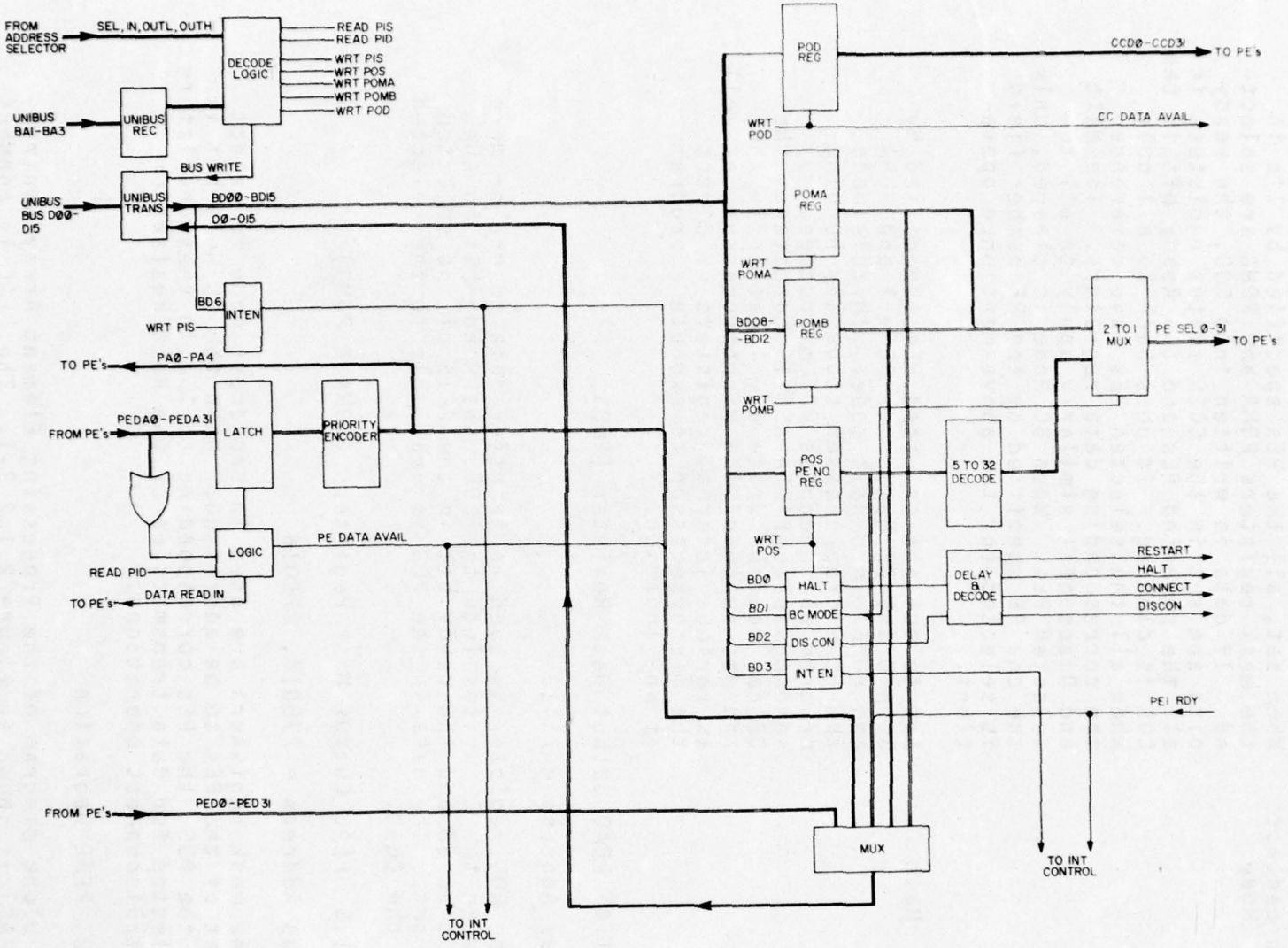


Figure 2.1.2.2-1. PEAC Block Diagram

When a PE has to transmit data to the CC, the PE loads the CCOD register in its CC Communication Port. This asserts the PE data available line into the PEAC (one of PEDA Ø - PEDA 31). These signals are effectively ORed together to produce the signal PE DATA AVAIL used as the Ready bit in the PIS. An interrupt is generated if the PIS Interrupt Enable is a "1". When one of the PEDAn lines is asserted, they are latched and encoded to produce the PE number field in the PIS. Since more than one PE might happen to assert its PEDAn simultaneously, a priority network selects one of the latched bits. The encoded PE number is sent back to the PEs (PAØ - PA4). The selected PE puts the contents of its CCOD onto the data lines (PEDØ - PED31) and the CC can read them in as its CID register. Reading in the CID produces a Data Read In signal which is sent to the PEs. The PE selected by PAØ - PA4 then removes its PEDAn line, and releases the data lines (PIDØ - PED31). The PIS Ready bit is cleared. If any other PEs had its PEDAn line on, the new state of the PEDAn lines are latched in, the Ready bit is set again and the whole cycle is repeated.

Output operations apply to one or more PEs, said to be "selected". If the broadcast mode bit in the POS is "Ø", the PE whose number is in the PE number field of the POS is selected. If the broadcast mode bit is "1", the PEs corresponding to "1"s in the mask registers POMA and POMB are selected. The PE SEL Signals are obtained by decoding the PE number and selecting either the decoder output or the contents of POMA and POMB under the control of the broadcast mode bit.

To transmit data, the CC loads the POD register. This results in sending the CC DATA AVAIL pulse to the PEs. The selected PE or PEs use this to set the Ready flip-flops in their CCIS registers. The ready flip-flops of selected PEs are inverted and wire-ored to produce the PEI RDY signal which thereby gets cleared. It gets reasserted when all the selected PEs read in their CCID registers, thereby clearing their CCIS ready bits. The PEI produces the ready bit in the PEAC POS register. Thus, when the PEs for whom the data was intended read it in, the ready bit is set and an interrupt is generated if the interrupt enable is on.

Writing a "1" in the write-only Halt bit of the POS puts the selected PEs in Halt mode and writing a "Ø" restarts the selected PEs.

About 1 microsecond after the POS is loaded a pulse is produced on the Halt or Restart Lines. The delay makes it possible to set the broadcast mode bit or the PE number field in the same instruction to specify the PEs to be halted or restarted.

Similarly, writing into the Connect bit produces a pulse on the Connect or Disconnect line going to the PEs. A CON/DISCON flip-flop is set or cleared in the selected PEs. This flip-flop controls a relay which removes power for the rest of the PE and disconnects critical signals.

2.1.3 CC Mass Storage

The Control Computer (CC) is provided with one or more disks for mass storage. This mass storage houses a generalized file-system (Section 3.2.3) that is utilized primarily for storage of programs and other data to be retained on-line over long periods of time.

Any of the several available PDP-11 compatible disks and controllers could be utilized for this purpose. In our design, we have utilized a Diva DD-54 system with a capacity of about 80 Mbytes, access time of 30 milliseconds, and transfer rate of 810 kbytes/sec.

2.1.4 Other CC Peripherals

The CC Complex includes a number of peripherals connected to the CC bus. Most of these would be standard off-the-shelf units available with PDP-11 interfaces. A typical G-471 installation may include the following, in addition to the disks already described:

(i) console CRT terminal or printer terminal

(ii) line printer

(iii) tape drives

(iv) modems for remote terminals

(v) graphics display.

2.1.5 DREA Decoder Control Bus

The purpose of the Decoder Control Bus is to permit the Control Computer to change the assignment of logical addresses to physical addresses of CWS memory banks. The Control Computer may make such changes at any time; and may change the assignments for any or all of the devices. Thus, it is possible for the Control Computer to prohibit a particular device from accessing any of the CWS banks (in case of device failure), or to prohibit any device from accessing a particular CWS bank (in case of memory failure).

The operation of the Control Bus in each Decoder is described in Section 2.4.2. Each Decoder receives the logical address and physical address on the bus. A separate enable signal is provided for each decoder. When a decoder receives its enable signal, it places the logical address from the bus in the register corresponding to the physical address from the bus.

The operation of the bus in the Control Computer is illustrated in Figure 2.1.5-1. The Control Computer contains three registers which control the bus, two mask registers and an address register. If the system contains more than 32 devices, additional mask registers are required.

To assign addresses, the Control Computer first loads the mask registers with 1's in the bits corresponding to the devices to be affected. Then it loads the address register with the logical and physical addresses and a 1 in the load bit. The addresses are placed on the bus immediately, and after a delay, enable signals are provided to the decoders corresponding to the 1's in the mask registers.

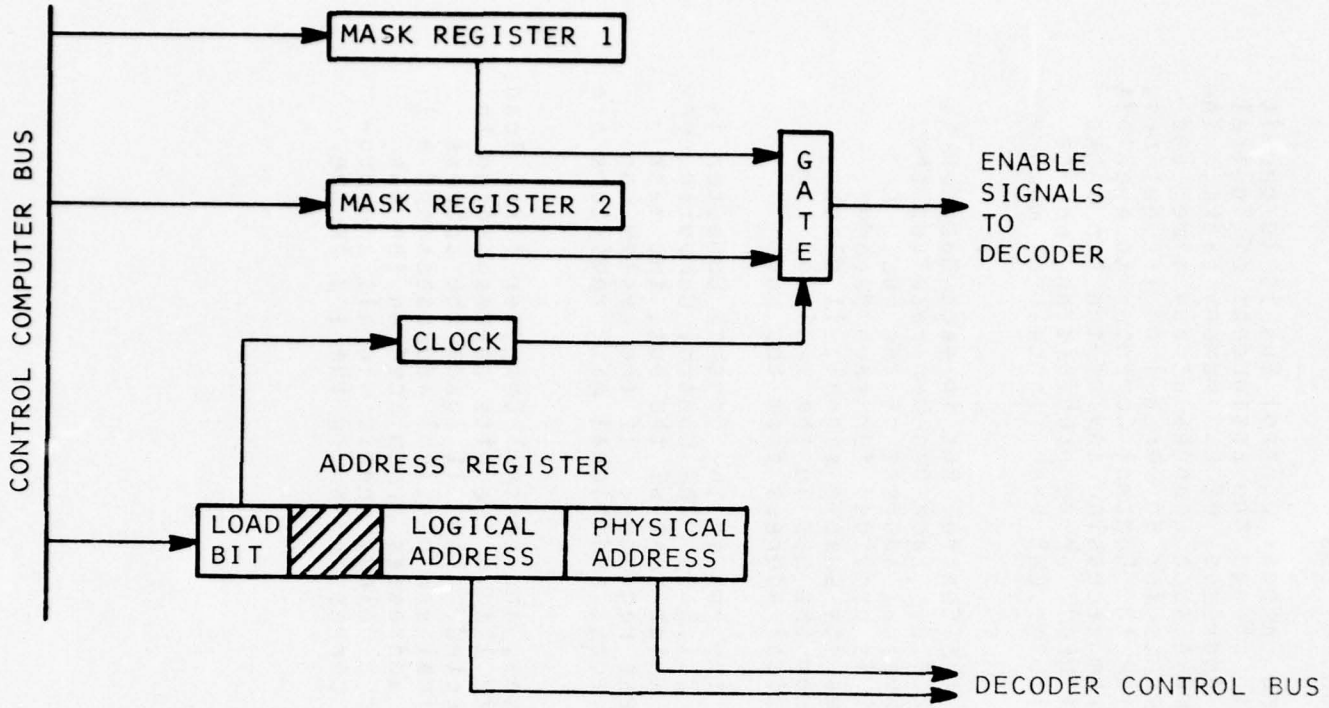


Figure 2.1.5-1. Decoder Control Registers.

2.1.6 The CC Mapper

This unit provides an interface between the CC and the system-wide Central Working Storage (CWS). The CWS can include up to 16 Mbytes and requires a 24 bit address. Moreover, it is organized to access 64 bits in parallel. The Mapper provides address mapping from the 64 kbyte CPU address space to the 16 Mbyte CWS address space and data-shifting to account for the different bus widths.

The Mapper also includes a special register (the LIS) that holds the data read in most recently from the CWS. This is utilized to provide unambiguous operation when the CC and one or more PEs attempt to access the same CWS word simultaneously.

In addition, the Mapper inserts parity bits for data going to the CWS and checks the parity of the data received from CWS.

Since the CC Mapper is almost identical to the PE Mapper, it will not be described further here. See Section 2.2.1.2 for a description of the PE Mapper.

2.2 The Processing Element Array

The Processing Element Array consists of up to 32 processing elements, which provide most of the computational power of the G-471 system. Up to 32 such PE arrays can be connected together providing a total of 1024 PEs in a system. The PE array is controlled by the Control Computer through the Processing Element Array Controller (PEAC - Section 2.1.2).

2.2.1 The Processing Element (PE)

The processing element is a major building block of the G-471 system. A block diagram of the PE is shown in Figure 2.2.1-1.

The PE is organized around a 16-bit microprocessor (see Section 2.2.1.1) such as the DEC LSI-11. A programmable high-speed Arithmetic Processor (AP) operating in parallel with the microprocessor provides additional processing power, especially for array-oriented floating-point computations of the kind frequently encountered in signal processing applications. The PE can execute 4 million floating point multiplications per second. Parallel operation of various units makes it possible to actually achieve this speed in addition to floating point additions, integer and logical operations, program flow control and data transfers.

The PE includes a high-speed memory, the PE Local Storage (PELS - Section 2.2.1.6) which can be accessed by both the microprocessor and the AP as well as by the PE Channel (see Section 2.2.1.4). The PELS contains up to 56 kbytes of bipolar memory with a cycle time of under 100 nanoseconds. Eight bytes can be accessed in parallel yielding a memory bandwidth of 80 Mbytes/sec. The PELS is partitioned and multi-ported so that the different PE processors can simultaneously access different parts of the PELS. In addition, they can access a system-wide storage facility, the Central Working Storage (CWS - Section 2.3) through the Data Routing Element Array (DREA - Section 2.4).

The PE channel (Section 2.2.1.4) is used to transfer blocks of data in parallel with the microprocessor and the arithmetic processor. It can transfer data between the CWS, PELS and the microprogram memories of the AP.

The PE includes three major buses: the Extended Microprocessor Bus, the Arithmetic Processor Bus and the PE Channel Bus. Each bus has 24 address lines, 64 data lines, 8 parity lines (one per data byte), 2 mode lines and some other control lines. Data is transferred in units of 8, 16, 32 or 64 bits as specified by the two mode lines. The 16 Mbyte address space of

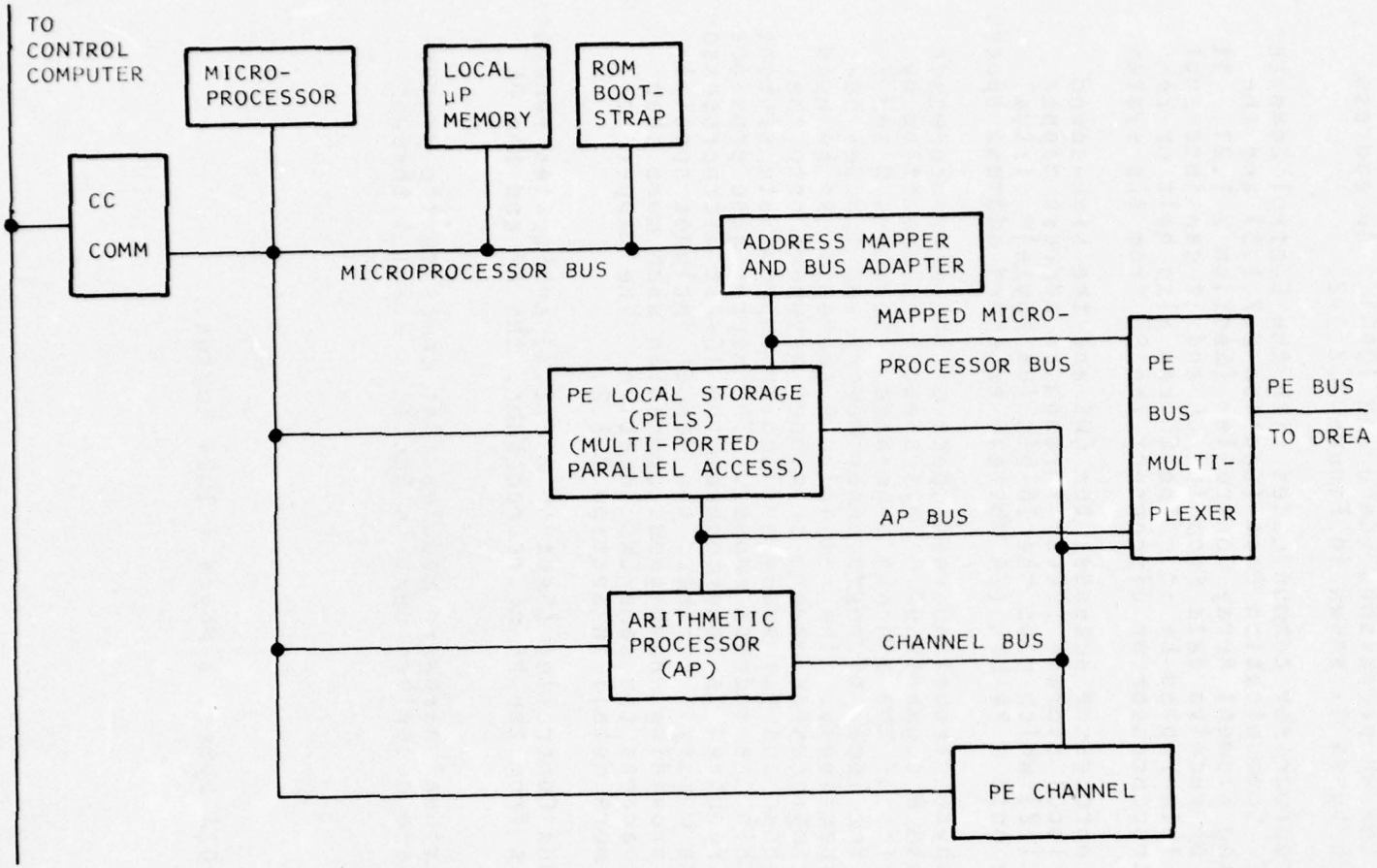


Figure 2.2.1-1. Block Diagram of Processing Element

the buses includes the CWS, the PELS and the microprogram memories for the AP processors (CAPU and IOAG). An address map for the PE buses is shown in Figure 2.2.1-2.

The microprocessor communicates with the Control Computer through its CC Communication Port (Section 2.2.1.5) and the CC's Processing Element Array Controller (Section 2.1.2). It can transmit or receive data from the CC and it can interrupt the CC or be interrupted by it. The CC can also halt or restart the microprocessor or disconnect the PE from the system.

The microprocessor accesses the CWS and the high-speed (bipolar) PE local storage (PELS) through an address mapper (Section 2.2.1.2) which maps the 16-bit (64 kbyte)* virtual address space onto a 24-bit (24 Mbyte)* extended address space.

The 64 kbyte virtual address space of the microprocessor is divided into 8 segments of 8 kbytes each, as specified by the upper 3 bits of the address generated. Segments Ø and 7 are reserved for local microprocessor memory and I/O device registers respectively. The remaining 6 segments can be used for local microprocessor memory or mapped anywhere onto the 16 Mbyte extended address space by loading appropriate segment registers within the address mapper. The mapper also provides the shifting required to interconnect the 16-bit microprocessor to 64-bit wide memory: Further, the Mapper includes special hardware for providing unambiguous operation when more than one processor access the same CWS location. The mapper is described in more detail in Section 2.1.2.

The PE Bus Controller (Section 2.2.1.3) arbitrates requests for CWS access from the PE microprocessor, the AP and the PE channel.

The individual hardware modules that comprise the processing element are described next in Sections 2.2.1.1 through 2.2.1.7.

---

* 1 kbyte = 1024 bytes, 1 Mbyte = 1024 kbytes.

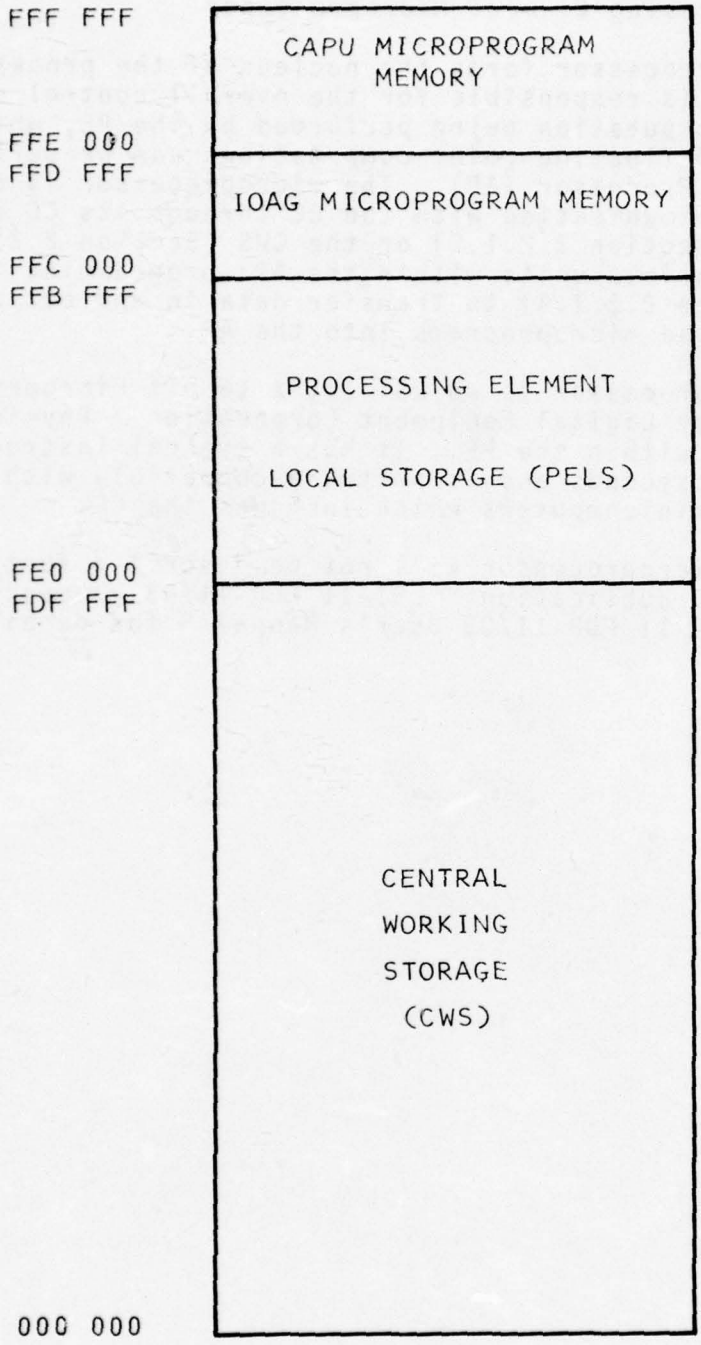


Figure 2.2.1-2. Address Space of PE Buses

2.2.1.1 The Processing Element Microprocessor

The PE microprocessor forms the nucleus of the processing element. It is responsible for the overall control of the flow of the computation being performed by the PE, while array-oriented and floating-point computations can be performed by the Arithmetic Processor (AP). The microprocessor is also responsible for communicating with the CC through its CC communication port (Section 2.2.1.5) or the CWS (Section 2.3); controlling the various units within the AP; programming the PE channel (Section 2.2.1.4) to transfer data in and out of the PELS and to load microprograms into the AP.

The PE microprocessor is an LSI-11, a 16-bit microprocessor manufactured by Digital Equipment Corporation. Physically, it is on one card within the PE. It has a typical instruction cycle of 2-5 microseconds and is software-compatible with the PDP-11 family of minicomputers which includes the CC.

The LSI-11 microprocessor will not be described further here. See the DEC publications "LSI-11 PDP-11/03 Processor Handbook" and "LSI-11 PDP-11/03 User's Manuals" for details.

2.2.1.2 The PE Address Mapper

The PE Address Mapper provides an interface between the PE microprocessor, and the Extended Microprocessor Bus (EMB) which has access to the Central Working Storage (CWS) and the PE Local Storage (PELS). The EMB can access up to 16 Mbytes of memory and requires 24-bit addresses, while the microprocessor generates 16 bit addresses for a 64 kbyte address space. The primary function of the PE address mapper is to map these 16-bit addresses generated by the microprocessor into 24-bit addresses so that the microprocessor has direct access to the PELS and the CWS.

The EMB provides 64 data bits and the CWS and PELS are organized to access 8 bytes of data in parallel. The mapper provides the data conversion to adapt the EMB to the 16-bit microprocessor. It also inserts parity bits (one per byte) during write operations and checks parity during read operations.

When several PE microprocessors access the same location simultaneously, the results could be ambiguous. The mapper includes a special register, the LIR, that helps in overcoming these kinds of problems and can be used in programming synchronization primitives. This will be described in more detail later in Section 2.2.1.2.4.

2.2.1.2.1 Address Mapping

This is the primary function of the address mapper. For this purpose, the address space of the microprocessor is divided into eight 8-kbyte segments, addressed by the three high order bits (A15-A13) of the generated address. Segment Ø is always used to access local memory on the microprocessor bus while segment 7 is used to access the I/O registers on the bus. The remaining six segments can be mapped to provide the microprocessor with up to six 8-kbyte windows that can be located anywhere within the 16 Mbyte address space of the EM Bus.

The mapper includes six 24-bit base registers (MBR1 through MBR6) as shown in Figure 2.2.1.2-1. The contents of the appropriate base register are added to the address from the microprocessor to generate the address for the EMB. The Mapper Control and Status Register (MCSR) included an Active Segment (AS) bit corresponding to each of the six segments that may be mapped. If the AS bit for a segment is "1", any address within it is mapped on to the EMB. Otherwise, it can be used to access local memory on the microprocessor bus.

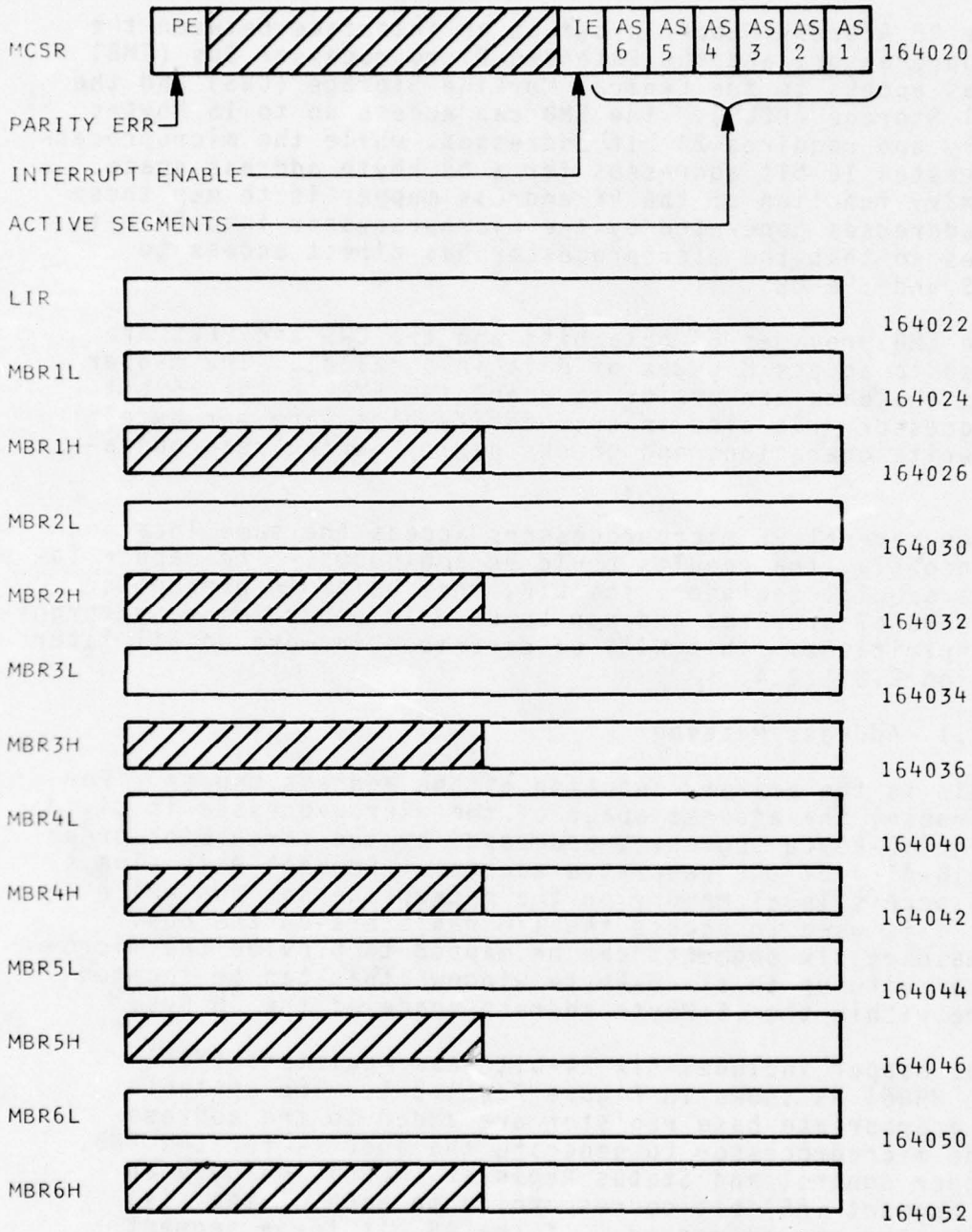


Figure 2.2.1.2-1. Address Mapper Control Registers

Figure 2.2.1.2-2 shows a block diagram of the mapper. The base registers are implemented as the 16x24 RAM consisting of six 16x4 RAM devices. When a base register is accessed directly by a microprocessor program, two cycles are required to access all 24 bits (the MBRnL and MBRnH registers). However, when the base register is used for mapping, all 24 bits are accessed in parallel.

During each cycle, bits 13 through 15 of the microprocessor bus are examined. If the value lies between 1 and 6, the corresponding AS bit in the MCSR is examined. If it is a 1, an EMB address is generated by adding the offset part of the microprocessor address (bits Ø-12) to the contents of the base register specified by bits 13 through 15, and an EMB cycle is initiated. If the AS bit is a zero, the mapper does not take any action and it is assumed that local memory will respond.

The control registers are interfaced to the microprocessor bus in a straightforward manner. Four bus transceivers are used for the electrical interface to the BADL signals. The address is latched into the Address Latches during a SYNC. The I/O Reg Decode circuit checks if one of the mapper registers is addressed by comparing against a switch-selectable address. In such a case, the appropriate input or output operation is performed to the addressed register within the mapper logic during DIN or DOUT cycles. All these registers are readable by the microprocessor to facilitate diagnostics.

2.2.1.2.2 Mode Conversion

The G-471 system is designed to allow memory operations (for the CWS and PELS) in four modes: byte (8 bits), halfword (16 bits), word (32 bits) or double word (64 bits). The memories themselves are accessed 64 bits at a time, but during a write cycle, only selected bytes within the 64-bits are written. A Multi-Mode Shifter (Section 2.2.1.7.7) in the arithmetic processor and the channel provides this function by shifting the data to or from the appropriate part of the 64 bit word.

In the case of the microprocessor, this function is greatly simplified since it can only use the 8 and 16-bit modes. Moreover, the microprocessor itself performs the 8 to 16 bit mode conversion. The mapper only needs to convert between 16 and 64 bits. On input, the appropriate 16-bit field of the 64 bits from the EMB is connected to the microprocessor using a 4 to 1 multiplexer. In byte mode, the microprocessor internally selects one of the two bytes. On

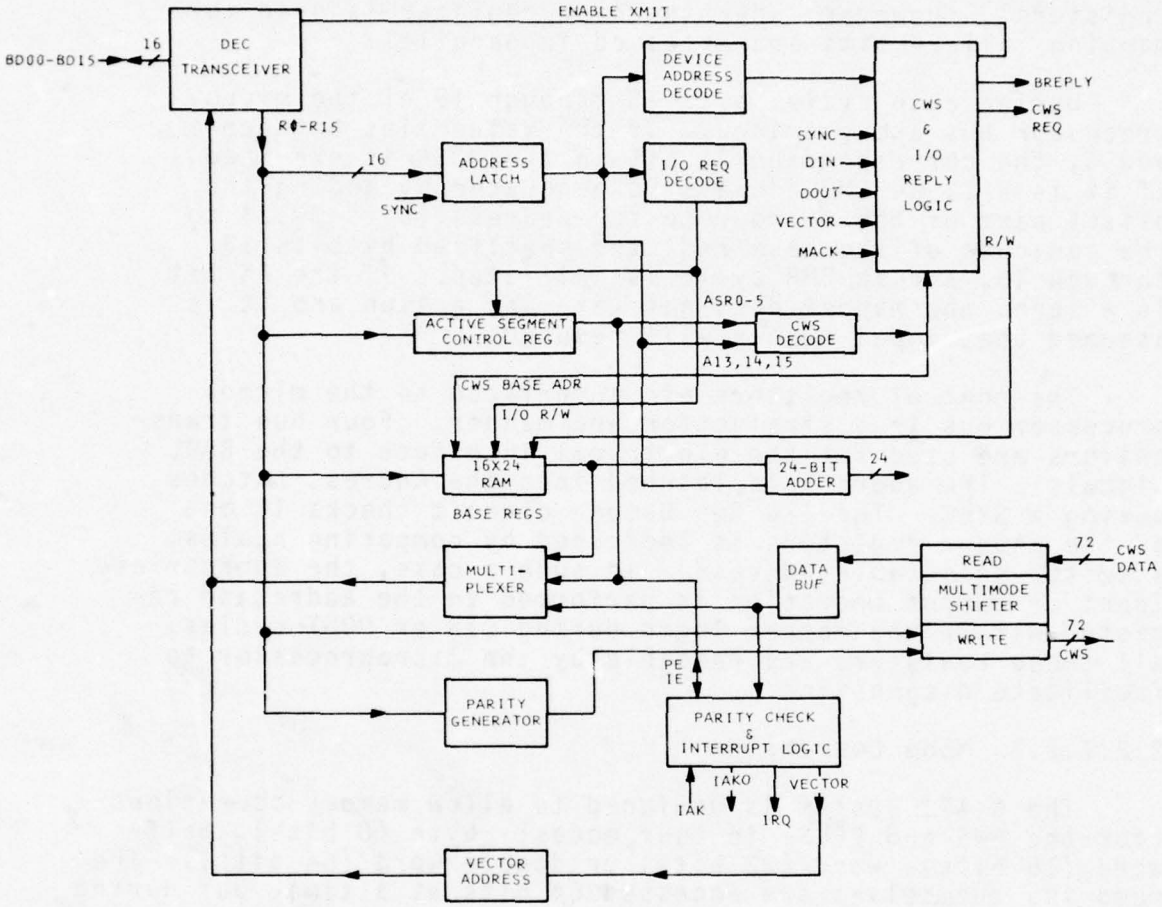


FIGURE 2.2.1.2-2 ADDRESS MAPPER BLOCK DIAGRAM
4047-760227-02

Figure 2.2.1.2-2. Address Mapper Block Diagram

output, the 16 data bits are simply fanned out to each of the four 16-bit fields in the 64-bit word sent to the EMB. The mode bits and the lowest order address bits are used by the addressed memory modules to determine which of the 8 bytes in the word actually need to be written.

2.2.1.2.3 Parity Detection/Insertion

On output, the mapper inserts a parity bit for each byte being written; even parity is utilized. On input each byte being read is checked for even parity. The remaining bytes in the 64-bit word received from the EMB are ignored. If an error is detected, the parity error bit (bit 15) is set in the mapper control register, and an interrupt signal is sent to the interrupt logic. If the interrupt enable bit is set, the interrupt logic initiates an interrupt sequence on the microprocessor bus.

2.2.1.2.4 Synchronization Hardware

When several PEs access the same word in CWS simultaneously, it is possible to get erroneous results unless certain special precautions are taken. This problem is specially critical when a CWS word is used for synchronization or communication (say as one of Dijkstra's semaphores).

The problem occurs because the several read/write cycles required to be executed by each processor can get interleaved. The usual solution is to provide special instructions for the processors such as the Test and Set for the IBM 370 computers. Since we are using a standard microprocessor, we cannot change the instruction set without losing the advantages of software-compatibility.

The scheme utilized is as follows: Microprocessor instructions that modify a memory word (as opposed to simply writing it) utilize Read-Modify-Write cycles. Such instructions include ADD, SUB, BIS, BIC, INC, DEC, etc. The DREA (Section 2.4) and the mapper are designed such that a microprocessor retains control of a CWS bank during a read-modify-write cycle, and no other processor can access the bank between the read and the write. This permits any of the above-mentioned instructions to be used for synchronization types of operations.

Further, the last input from the EMB is available to the program in the LIR register within the mapper (Figure 2.2.1.2-1). An example of the use of these features is the use of a pointer word in CWS to fetch tasks for the PE (See Section 3.2.14.2.6 for details). The PE microprocessor increments the pointer using an instruction with a read-modify-write cycle (ADD) to

ensure unambiguous operation. Then it uses the contents of the LIR to get the value of the pointer just before it was incremented and uses it to fetch its task. If it read the pointer directly instead, there is the possibility that it was modified meanwhile by another PE.

The LIR feature is implemented by inserting a Data Buffer in the read path as shown in Figure 2.2.1.2-2 and connecting it to the bus both during normal read operations from the EMB and when the microprocessor executes a read cycle using the address of the LIR.

2.2.1.3 PE Bus Control

Each processing element (PE) has a PE bus over which it can access the Central Working Storage via the Data Routing Element Array. Internally, there are three devices within the PE that operate in parallel and have their own buses. These devices and their busses are:

	device	bus
(i)	PE microprocessor (LSI-11)	Extended microprocessor bus
(ii)	PE channel	Channel bus
(iii)	Arithmetic Processor	AP bus.

Each of these buses can simultaneously access different PELS banks or the CWS. The PE bus control provides the multiplexing and arbitration to connect any of these three buses to the PE bus when CWS access is required.

Each of these devices presents the following signals on its bus when it requests a cycle:

(i) memory address
(ii) write data (for a write cycle)
(iii) Read/Write direction
(iv) Mode
(v) Request Signal

The address space of these three buses also includes the PE Local Storage (PELS) in the top 64 kbytes. Thus a PELS cycle is requested if the address bits A16-A23 are "1"s.

Figures 2.2.1.3-1 and 2.2.1.3-2 show the organization of the PE Bus Control. The PE Bus Control address decoders examine the higher-order address bits of the three buses along with their request signals. If any of these indicates a CWS request, the bus control arbiter generates a signal to clock the three priority flip-flops, FF1 through FF3. The priority arbitration circuit allows only one of these to get set in the case of simultaneous requests. The priority assignments are:

(i) AP (highest priority)
(ii) PE Channel
(iii) PE microprocessor (LSI-11)

Once a priority flip-flop is set, it inhibits further clocking until the memory cycle is over (when the request signal of the selected bus goes away). The multiplexers are addressed so that the addresses, data, REQ and mode signals of the selected bus are connected to the PE bus.

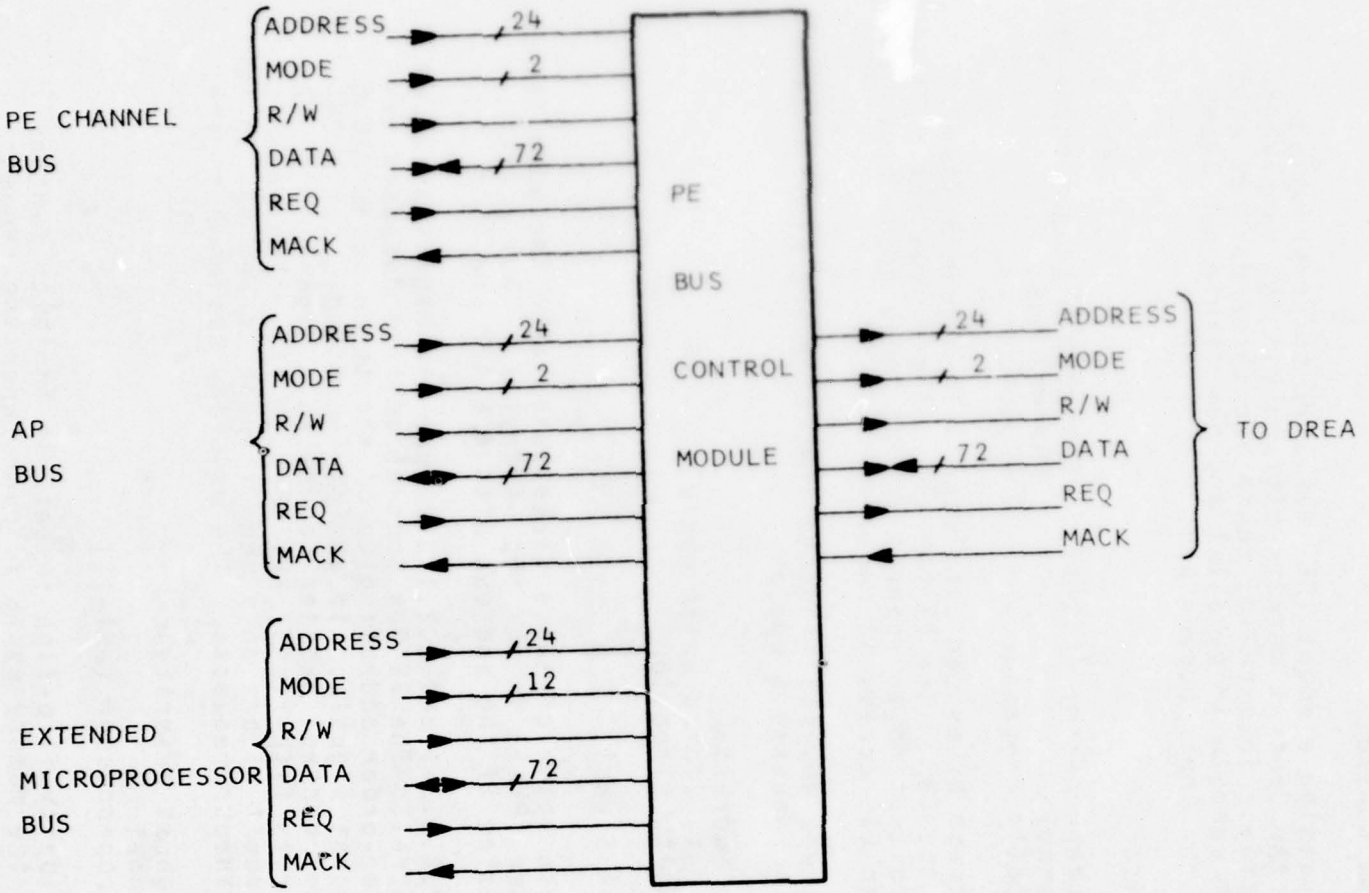
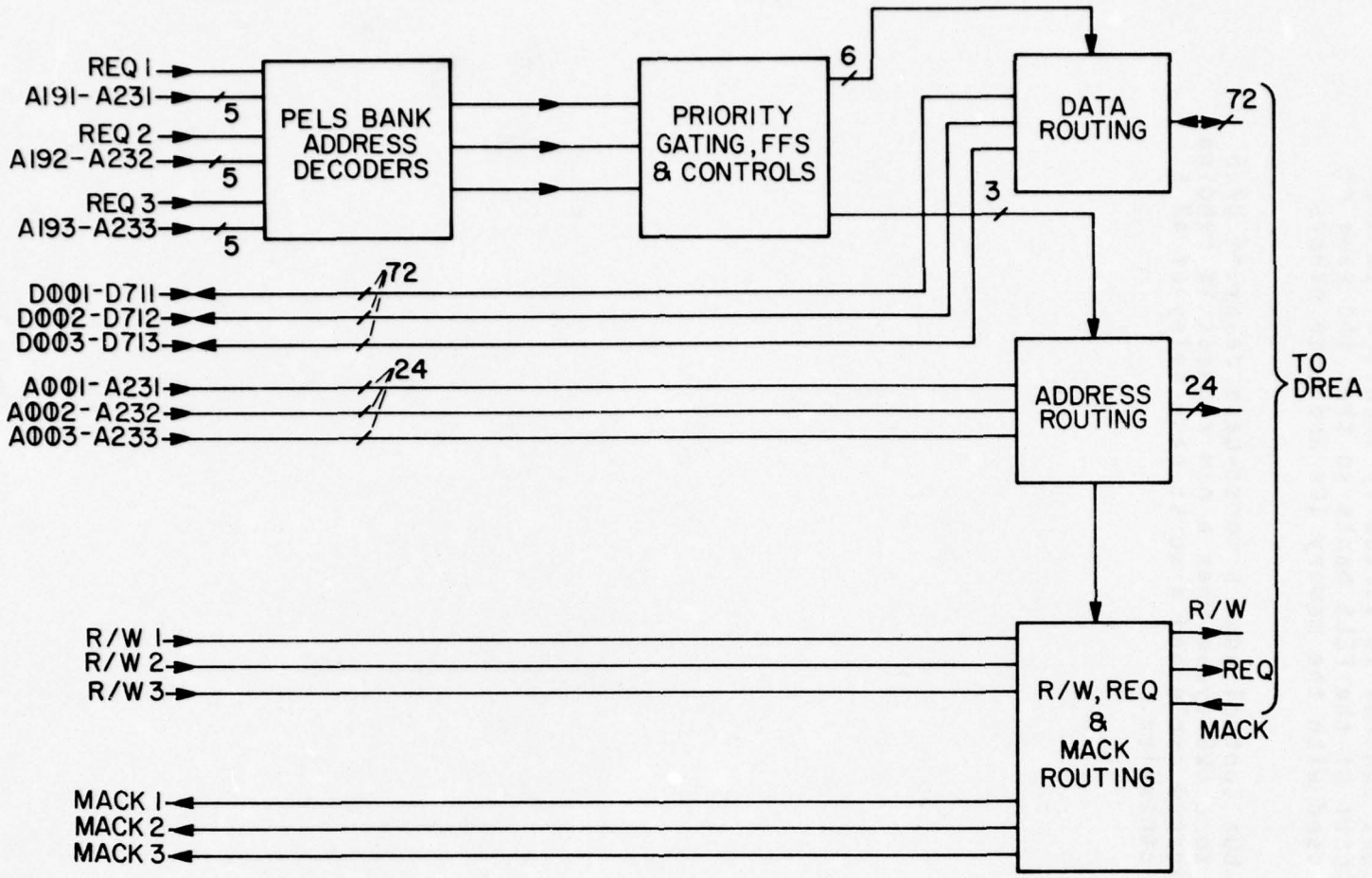


Figure 2.2.1.3-1. PE Bus Control



SUFFIX 1= CAPU
2= PE CHANNEL
3= ADDRESS MAPPER

FIGURE 2.2.1.3-2 PE BUS CONTROL BLOCK DIAGRAM
4047-760222-10A

The PE bus control is virtually identical to the arbitration part of the PELS banks so that the same pc card can be used with the memory ICs and some others left out.

The PE bus control adds a worst-case delay of 27.5 nanoseconds to a CWS cycle when a new request is pending during a previous cycle and a worst-case delay of 47.5 nanoseconds otherwise.

2.2.1.4 The PE Channel

The PE channel operates under the control of the PE microprocessor to transfer blocks of data between the Central Working Storage (CWS), the PE Local Storage (PELS) or the microprogram memories within the Arithmetic Processor. The data at either the source or the destination may be contiguous or scattered with a fixed increment between the words. This can be utilized, for example, to read or write a column of a matrix stored in row order. The data can be transferred in units of 8, 16, 32 or 64 bits. The CWS, PELS and AP microstores occupy different parts of the address space of the channel bus. The source and destination of a channel transfer can be anywhere within this space; this may lie in the same or different memories and may even overlap.

2.2.1.4.1 PE Channel Control Registers

The PE microprocessor controls the channel through the PE Channel Control registers in its address space. These registers are shown in Figure 2.2.1.4.1-1. The individual registers are described next.

2.2.1.4.1.1 Control and Status Register (CSR)

Bus Address = 164000
Interrupt Vector = 300, 302

BIT	NAME	FUNCTION
Ø	Go bit	This is a write-only bit used to initiate the operation of the channel
1,2	Mode	These two bits indicate the width of data to be used for the transfer (ØØ-64 bits, Ø1-32 bits, 1Ø-16 bits, 11-8 bits)
6	Interrupt Enable	If set, the Ready or Error bits will cause an interrupt
7	Ready	If set, it indicates that the channel is ready for a new transfer. Cleared when the Go bit is loaded, set when the transfer is completed.
13	Time out Error	Set if no response is received from a memory for 50 microseconds, usually indicating an invalid memory address.
14	Parity Error	Set if a word read by the channel had incorrect parity.

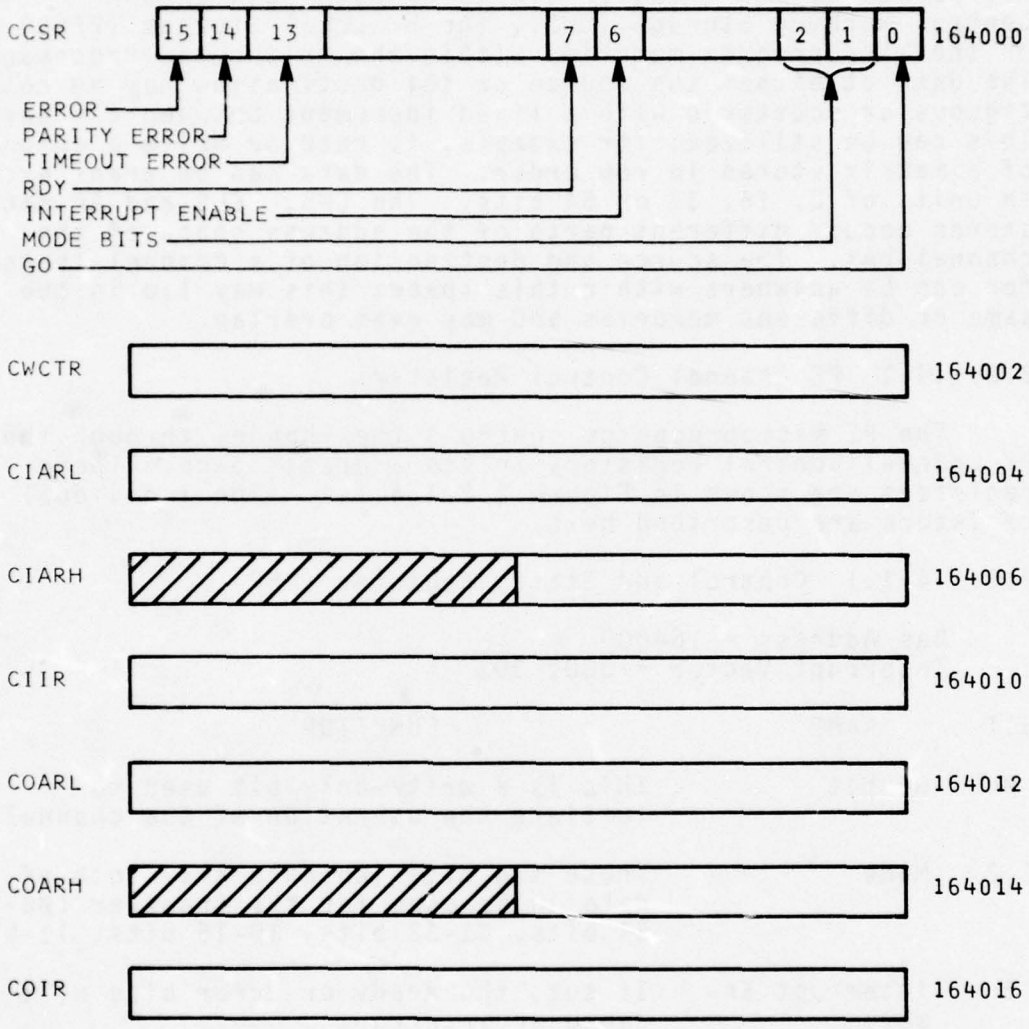


Figure 2.2.1.4.1-1. PE Channel Control Registers

15	Error	This is the OR of Timeout Error and Parity Error

2.2.1.4.1.2 Word Count Register (WCTR)

Bus Address = 164002

The number of words to be transferred is loaded into this register by the microprocessor. It is decremented after each cycle of the channel, and the Ready bit in the CSR is set when it goes to Ø.

2.2.1.4.1.3 Input Address Registers (IARL and IARH)

Bus Address = 164004, 164006

This is a 24 bit register that appears as two 16-bit registers to the microprocessor (IARL = bits Ø-15, IARH = bits 16-23). It is initially loaded by the microprocessor with the starting source address for the channel transfer. After each channel cycle, it is updated by adding the contents of the Input Increment register. It provides the address of the word to be read by the channel. The width of the word is given by the mode bits of the CSR.

2.2.1.4.1.4 Input Increment Register (IIR)

Bus Address = 164010

This register holds the read increment value. It is loaded by the microprocessor prior to a transfer. It is added to the Input Address Register during each cycle to generate read addresses.

2.2.1.4.1.5 Output Address Registers (OARL and OARH)

Bus Address = 164012, 164014

This is a 24 bit register that appears as two 16-bit registers in the microprocessor's address space (OARL = bits Ø-15, OARH = bits 16-23). It is loaded by the microprocessor with the starting source address before executing a channel transfer. During each channel cycle, it is used as the write address, and then updated by adding the contents of the Output Increment Register to it.

2.2.1.4.1.6 Output Increment Register (OIR)

Bus Address = 164016

The OIR holds the value of the distance (in bytes)

between successive words to be written by the channel. It is loaded by the microprocessor prior to a channel transfer. It is added to the COAR during each cycle to generate successive write addresses.

2.2.1.4.2 PE Channel Operation

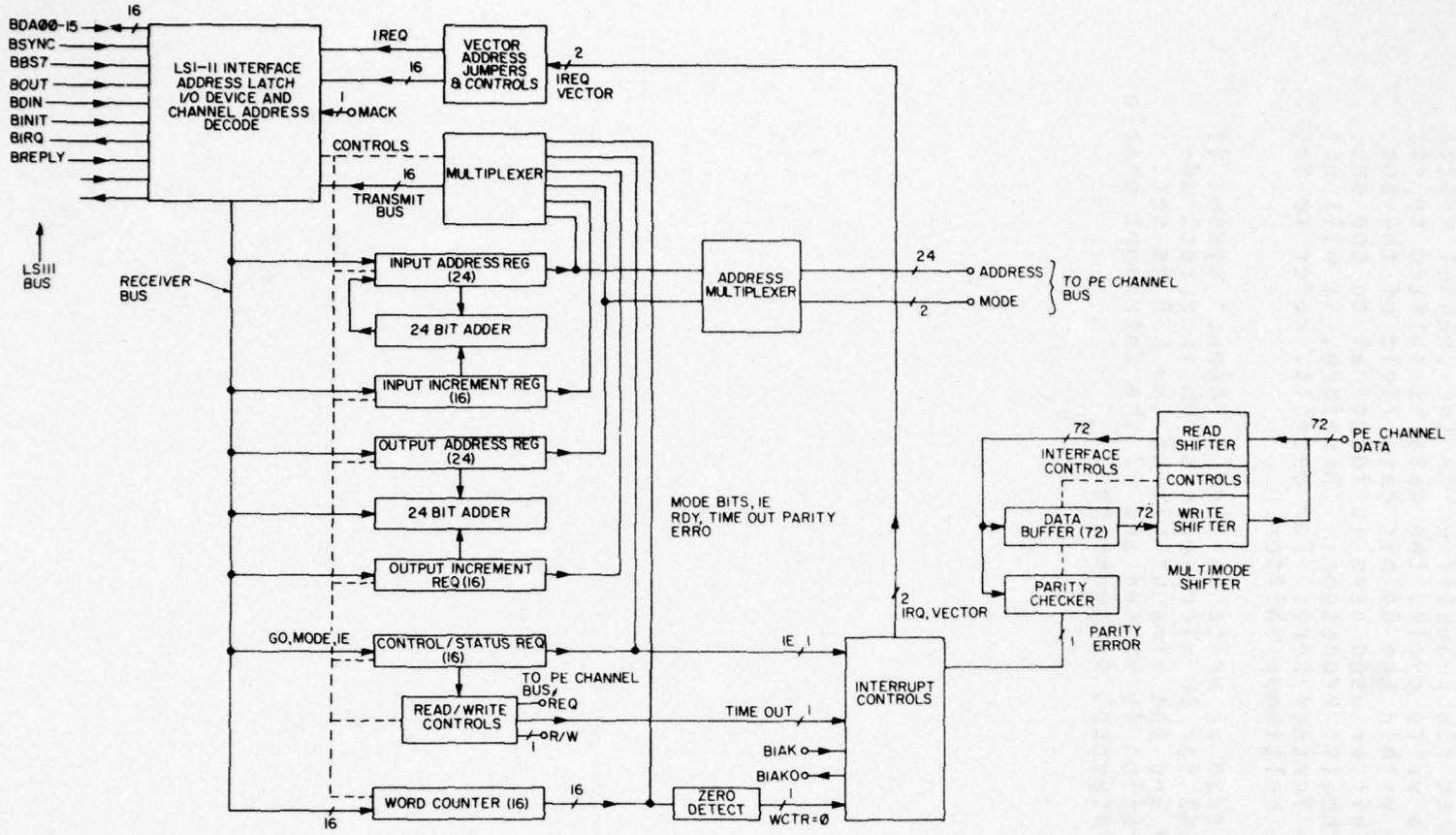
Figure 2.2.1.4.2-1 shows a block diagram of the PE Channel. The PE channel control registers are connected to the microprocessor bus via four bus transceivers. The bus address is latched at SYNC time and compared by the Address Comparators against a switch-selectable value to see if one of the channel control registers has been addressed. If so, the appropriate transfers take place, and the required signals are generated by gating the DIN and DOUT with the low order address bits.

The various registers are loaded by the microprocessor prior to a channel transfer operation. Then, a '1' is written into the Go bit of the CSR, which starts the channel sequence. The channel outputs the read address from the IAR onto the address lines of the channel bus. The address is sent to all the memories on the bus, and the addressed memory comes back with a reply at the end of its read cycle and presents data to the channel. The channel loads this data into the Data Buffer. Then it outputs the write address from the OAR onto the bus address lines, and the contents of the Data Buffer onto the data lines. The addressed memory executes a write cycle and replies back at the end of the cycle.

The IAR and OAR are updated by adding the contents of the IIR and OIR respectively. At the same time, the word counter is decremented; if the result is zero, the channel stops the operation and sets the Ready flip-flop. If the Interrupt Enable flip-flop (CSR bit 6) is a "1", the interrupt control logic initiates an interrupt sequence and presents the interrupt vector address to the microprocessor bus. If the word counter is non-zero, however, the channel repeats the read-write cycle.

During a read cycle, the input data is checked for even parity using parity checker/generator ICs. If the parity is correct, the data, along with the parity bits, is used in the write cycle. If a parity error is detected, the PE bit of the CSR (bit 14) is set and the channel operation is halted. An interrupt is generated if specified by the Interrupt Enable bit.

A Multimode Shifter is used to correctly position the data in the different access modes, as specified by the mode bits of the CSR. During a read cycle, the data from the



PE CHANNEL BLOCK DIAGRAM
D.N. 4047-760227-04C

Figure 2.2.1.4.2-1. PE Channel Block Diagram

memory is presented right-justified to the channel's Data Buffer. During a write cycle, the data is shifted to its correct location within the 64-bit data field of the bus. The multi-mode shifter used here is identical to the one used in the Arithmetic Processor. Therefore, it will not be described any further here. For details, refer to Section 2.2.1.7.7 - Multimode Shifter.

During the read or write part of a channel cycle, if no memory responds for 50 microseconds, an incorrect address is assumed and the Timeout bit of the CSR is set. The channel operation is halted and if the interrupt enable bit is set, an interrupt is generated.

2.2.1.5 CC Communication Port

The CC Communication Port is the interface between the PE microprocessor and the PE Array Controller (PEAC) in the control computer. Refer to Section 2.1.2 for a description of the PEAC.

The CC Communication Port is designed to make the CC appear as a console device to the microprocessor. This makes it possible for the CC to communicate with the microprocessor when it is in ODT mode so that the CC can have complete control over it. Further, it allows standard PDP-11 software such as loaders to be used unmodified in the microprocessor.

2.2.1.5.1 CC Communication Port Control Registers

The registers in the address space of the PE microprocessor (LSI-11) utilized for program control of the CC communication port are shown in Figure 2.2.1.5.1-1. The individual registers are described next.

2.2.1.5.1.1 CC Input Status Register (CCIS)

Bus Address = 177560
Interrupt Vector = 60, 62

BIT	NAME	DESCRIPTION
7	Ready	This is set when data is available from the CC in the CCID register. Cleared when the microprocessor reads out of CCID.
6	Interrupt Enable	When set, causes Ready = 1 to cause an interrupt.

2.2.1.5.1.2 CC Input Data Register (CCID)

Bus Address = 177562

The CCID holds data transmitted to the PE by the CC when Ready = 1.

2.2.1.5.1.3 CC Output Status Register (CCOS)

Bus Address = 177564
Interrupt Vector = 64, 66

BIT	NAME	DESCRIPTION
7	Ready	Cleared when the microprocessor writes into the CCOD. Set when the data in the CCOD has been read in by the CC.

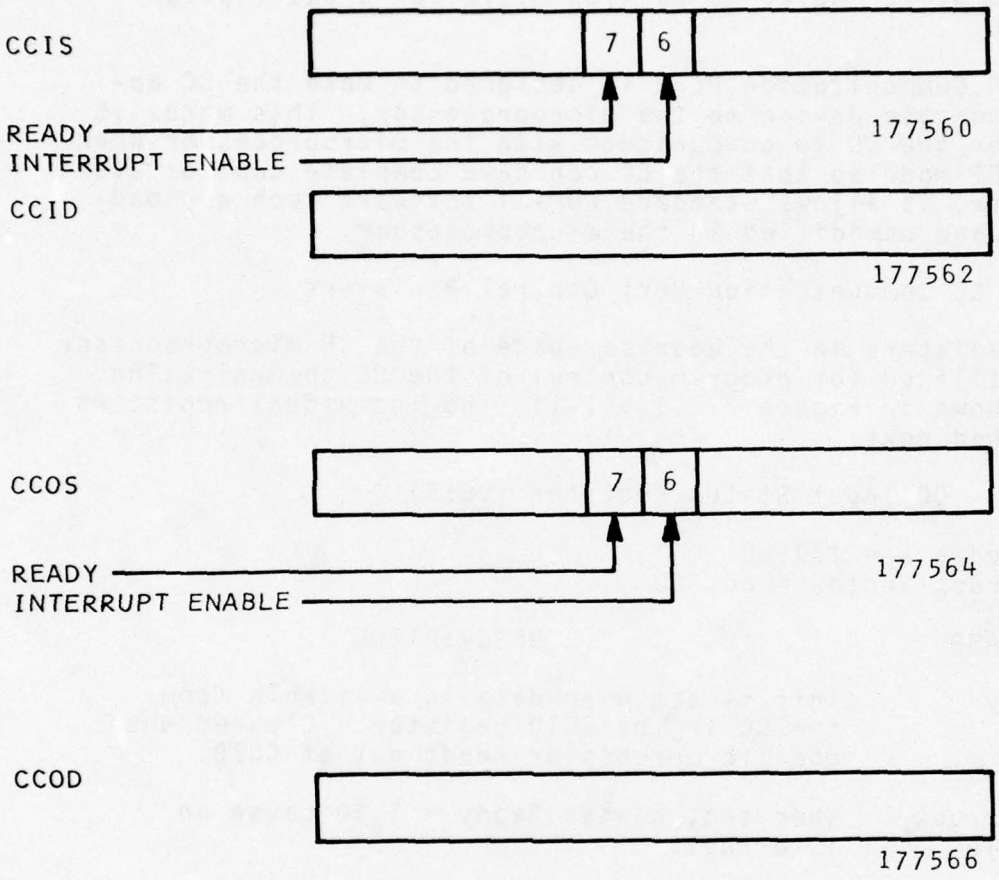


Figure 2.2.1.5.1-1. CC Communication Port Control Registers

6	Interrupt Enable	When set, Ready = 1 causes an interrupt

2.2.1.5.2 CC Communication Port Operation

A block diagram of the CC Communication Port is shown in Figure 2.2.1.5.2-1. The description of the PEAC (Section 2.1.2) should be referred to in order to understand the following description fully.

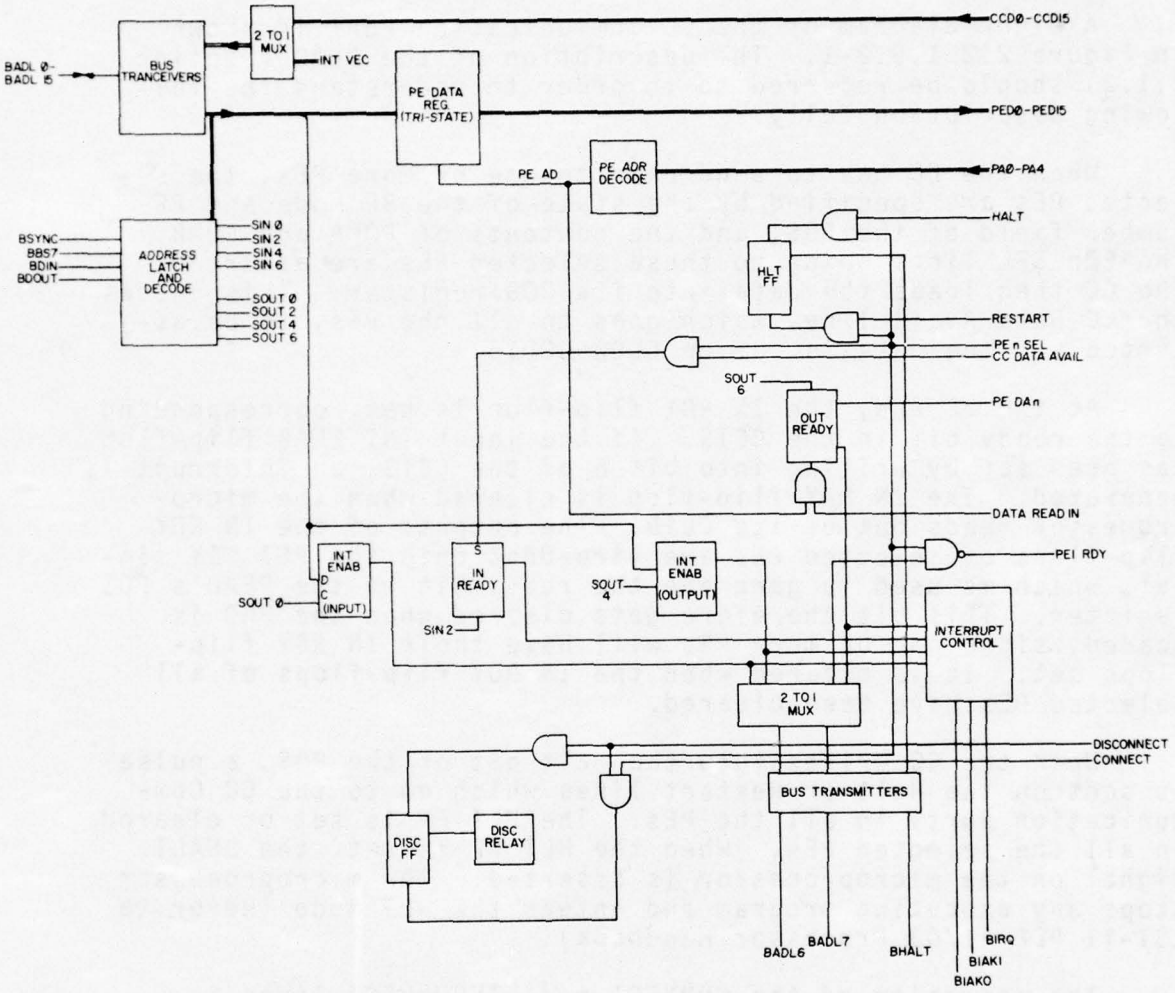
When the CC has to send data to one or more PEs, the selected PEs are specified by the state of the BC Mode and PE number field of the POS; and the contents of POMA and POMB. The PEn SEL lines going to these selected PEs are asserted. The CC then loads the data into its POD register. This causes the CC Data Avail line, which goes to all the PEs, to be asserted and the data is put on CCDØ-CCD15.

At the PE end, the IN RDY flip-flop is set, corresponding to the ready bit in the CCIS. If the input INT ENAB flip-flop has been set by writing into bit 6 of the CCIS, an interrupt is generated. The IN RDY flip-flop is cleared when the microprocessor reads out of its CCID. The outputs of the IN RDY flip-flops of selected PEs are wire-ORed onto the PEI RDY signal, which is used to generate the ready bit of the PEAC's POS register. This bit therefore gets cleared when the POD is loaded, since one or more PEs will have their IN RDY flip-flops set. It is cleared when the IN RDY flip-flops of all selected PEs have been cleared.

When the CC writes into the Halt bit of the POS, a pulse is sent on the Halt or Restart lines which go to the CC Communication Ports in all the PEs. The HLT FF is set or cleared in all the selected PEs. When the HLT FF is set, the BHALT signal on the microprocessor is asserted. The microprocessor stops any executing program and enters the ODT mode (Refer to LSI-11 PDP-11/03 Processor Handbook).

The operation of the CONNECT and DISCONNECT lines is similar. A pulse is produced on one of these when the CC writes into the Disconnect bit of its POS. The DISC FFs of selected PEs are set or cleared. The DISC FF drives a relay which turns off the power to the rest of the PE and disconnects critical signals. Note that the card containing the CC Communication Port remains powered up.

When data is to be transmitted by the PE microprocessor to the CC, it writes into its CCOD. The data is loaded into the tri-state PE Data Register, and the OUT READY flip-flop is set. This causes the Ready bit in the CCOS to be cleared, and the PEDAn line to the PEAC to be asserted. The PEAC



DWG. NO. 4047-760304-IC

Figure 2.2.1.5.2-1. Block Diagram of CC Communication Port

arbitrates between all the PEDAn lines and returns the address of the selected PE on the PAØ - PA4 lines. At the PE end, when this address is decoded, the contents of the PE Data Register are connected to the PEDØ - PED15 lines by enabling its tri-state output buffers. When the CC accepts this data by reading in its PID, a pulse is sent to all PEs on the Data Read In line. The PE selected by PAØ - PA4 uses this to reset its Out Ready flip-flop. This sets the Ready bit in its CCOS and if the output Interrupt Enable flip-flop was set, an interrupt is generated by the Interrupt Control logic.

2.2.1.6 Processing Element Local Storage (PELS)

The Processing Element Local Storage is a high-speed (100 nanosecond) bipolar multiported memory within the processing element. It is used to hold data that needs to be accessed frequently in order to obtain higher throughput and reduce contention for the Central Working Storage. Data to be processed is generally moved into PELS from CWS and the computed results moved back into CWS by the PE Channel (Section 2.2.1.4).

The PELS is organized as up to fifteen 1024x64 PELS banks (1024x72 including parity). Each bank has three ports connected to the AP Bus, the PE Channel Bus and the Extended Microprocessor Bus respectively. Thus it is possible for the microprocessor, the channel and the arithmetic processor to access different banks at the same time. A typical situation might be the following: the microprocessor executes its program from one bank, the AP operates on data in a second bank and concurrently, the channel writes out processed data from a third PELS bank to CWS and reads in fresh data for the AP to process.

The PELS occupies a part of the 4 Megabyte address space of the three buses; (bus addresses FEØØØØ to FFBFFF) located just above the Central Working Storage address area. See Figure 2.2.1.2-1 for an address map of the three PE buses.

Each PELS bank consists of a 1024x72 memory stack and associated control circuitry, a bank address decoder, an arbiter for the three ports, and address and data routing circuitry. Physically, a PELS bank is split into two cards each containing 1024x36 bits, primarily because of constraints on the number of pins available per card. The memory controls are duplicated on both cards to avoid skewing problems between the cards. Each card also contains the related data routing logic. In addition to the above, one of the cards also contains the PELS bank decoder and arbiter.

A block diagram of a PELS bank is shown in Figure 2.2.1.6-1. The Arbiter and Control module receives requests for memory over one or more of the three buses along with addresses and mode information. The bank decoders compare bits 13 to 23 of the address on each bus with the address of the bank and generate corresponding local requests. These are clocked into the priority flip-flops FF1 through FF3 through a priority network that allows only one to be set. The priorities assigned to the three buses are:

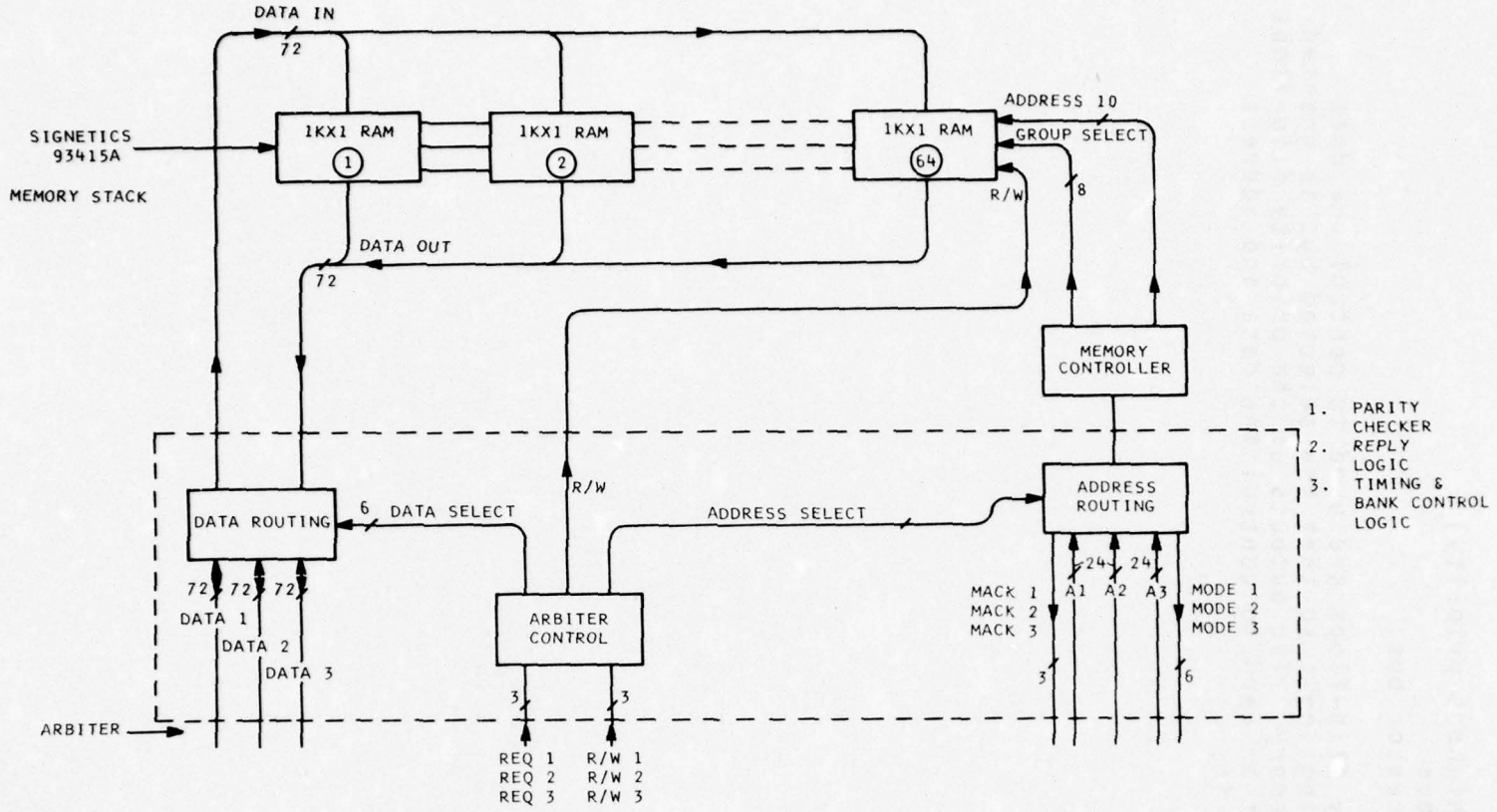


Figure 2.2.1.6-1. Block Diagram of PELS Bank

(i) AP bus (highest priority)
(ii) Channel bus
(iii) Microprocessor bus.

The priority flip-flops are used to control the data and address routing logic so that the selected bus is granted control of the memory. The outputs of the priority flip-flops also go to the other card to control the data and address routing logic on it.

2.2.1.7 The Arithmetic Processor (AP)

The Arithmetic Processor provides the processing element with a capability for high-speed floating point processing, especially for array-oriented computations. The AP operates under the control of the PE microprocessor but in parallel with it. Internally, the AP consists of several independent processing units operating in a pipeline to obtain extremely high throughput. The AP can simultaneously execute close to 4 million floating-point multiplications, 6 million floating-point additions and 10 million integer arithmetic or logical operations per second. As an example, it can compute a 1024 point FFT in approximately 5 msec.

A high memory bandwidth is generally required to utilize a high processing rate of this kind. The AP can input or output a 64 bit word (two real floating-point numbers or a complex number) every 100 nanoseconds.

A block diagram of the AP is shown in Figure 2.2.1.7-1. The addresses of the input and output data are computed by the Input-Output address generator (IOAG) and stored in the Input Address FIFO and Output Address FIFO respectively. The IOAG computes 24-bit addresses for the AP bus, whose address space includes the high-speed PE Local Storage (PELS) and the system-wide Central Working Storage (CWS). Two designs for the IOAG were developed: a fully-programmable version based on a stripped-down version of the CAPU, and a hard-wired version. These are described further in Sections 2.2.1.7.2.1 and 2.2.1.7.2.2.

The Data Fetch unit picks up the input addresses from the Input Address FIFO, fetches the corresponding data from memory via the AP Bus Control, and deposits it in the Input Data FIFO.

The data is picked up from the Input Data FIFO by the Central Arithmetic Processing Unit (CAPU) and processed. The CAPU is the number-cruncher of the system. It features a high-speed floating-point multiplier (270 nanoseconds); adder (180 nanoseconds); 32-bit ALU (90 nanoseconds); special ROM-based hardware for other functions such as division, logarithms, exponentials and trigonometric functions; 16 32-bit general purpose registers and two 32-bit buses. All these units can operate in parallel; the operations are specified by different fields in the 64-bit CAPU instruction word. The results of the CAPU's computations are loaded into the Output Data FIFO. In addition to regular data from the Input Data FIFO, the CAPU can also fetch data directly when the addresses are not known until run

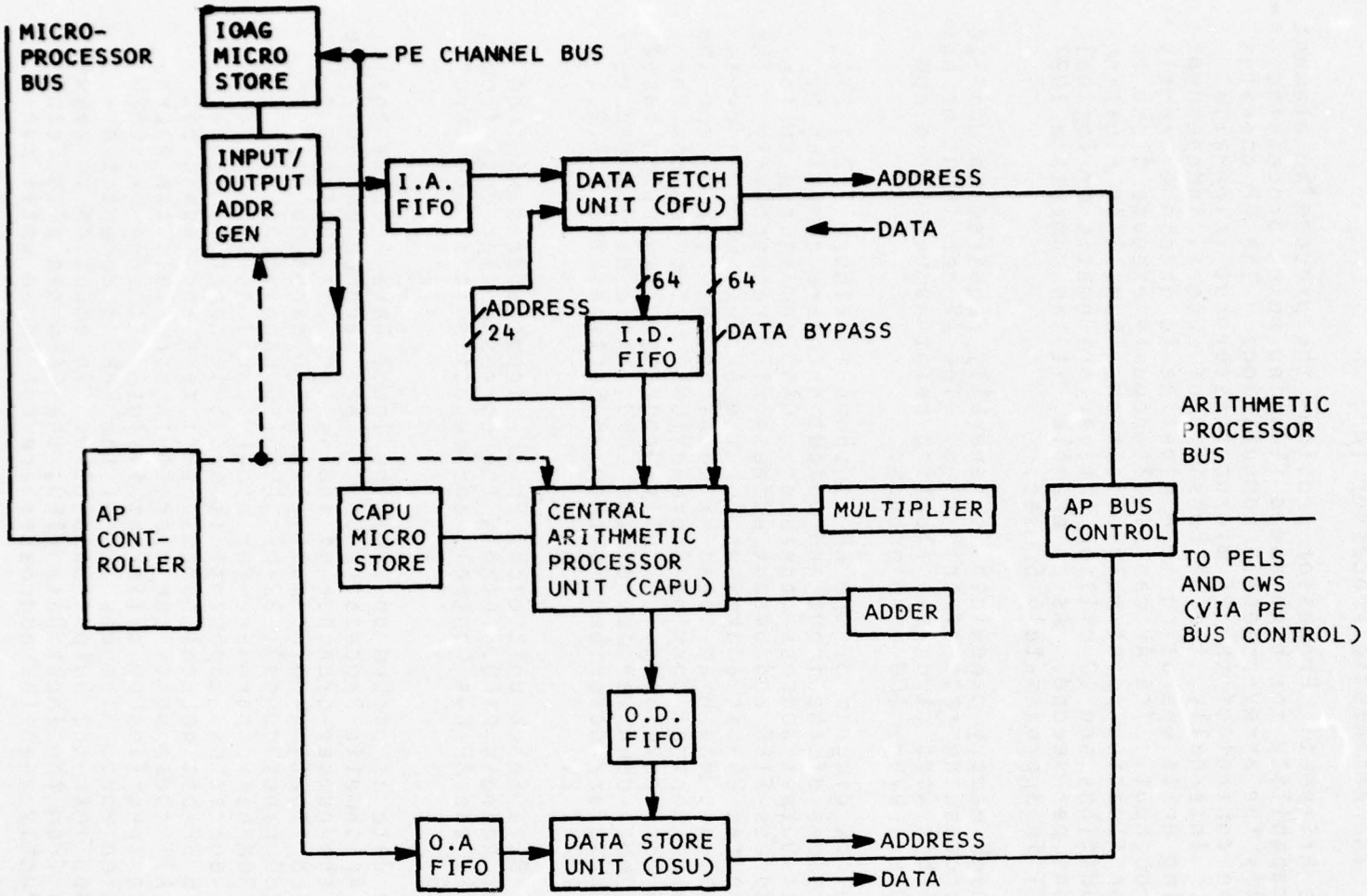


Figure 2.2.1.7-1. Block Diagram of Arithmetic Processor (AP)

time. Examples of this include table look-up, array-indices, parameters, etc.

The data in the Output Data FIFO is stored by the Data Store Unit, utilizing addresses from the Output Address FIFO.

The AP Bus Control arbitrates between the Data Fetch Unit and the Data Store Unit and controls the AP bus which goes to the PE Local Storage banks and to the Central Working Storage via the PE Bus Control.

The whole AP operates under the control of the PE microprocessor which can control or monitor the various units of the AP through registers in the AP Controller. The microprocessor also programs the PE channel to load the microprograms for the IOAG and the CAPU.

2.2.1.7.1 Central Arithmetic Processing Unit (CAPU)

2.2.1.7.1.1 Functional Description

The CAPU is a special purpose, arithmetic processing unit, consisting of two major sections: a Program Sequencer and a Data Flow section. The primary purpose of the CAPU is to perform arithmetic operations on a sequence of 32-bit floating-point numbers. For this reason, all of the buses and registers in the Data Flow section are 32-bits, and most of the circuitry is devoted to floating-point arithmetic.

In operation, programs are loaded into the CAPU program memory by the Channel. The AP Controller can then load a starting address into the CAPU and start the program. The CAPU obtains input data from the Input Data FIFO, processes it according to the program, and sends output data to the Output Data FIFO. Exceptions to this type of operation will be mentioned in the following sections.

Most programs for the CAPU will consist of simple loops to be executed as long as data continues to appear in the input FIFO. For this reason, the program memory is small (256 words) and the branching facilities are limited. However, since all branch locations are calculated from the current PC, programs may be relocated anywhere in memory without changes. The program memory may be expanded to 1024 words, or further if complete branching capability is not required.

2.2.1.7.1.1.1 Program Sequencer

A block diagram of the Program Sequencer is shown in Figure 2.2.1.7.1.1.1-1. The sequencer consists of a 256 word x 64 bit program memory, an instruction register, and various circuits for determining the next address.

In operation, the Channel will begin by loading a program into the CAPU memory. It does this by disabling the AM2909 tri-state outputs and using its own inputs to the memory. To start a program, the AP Controller places the starting address in the START register and sets the RUN bit, which initializes the IR and enables the CAPU clock. When the IR contains 01 in the Next Address Type field, the next address is taken from the START register.

During execution of the program, the next address is normally obtained from the incremented PC. However, two types of branches can be used. For a conditional branch, the next address is taken from the incremented PC if the

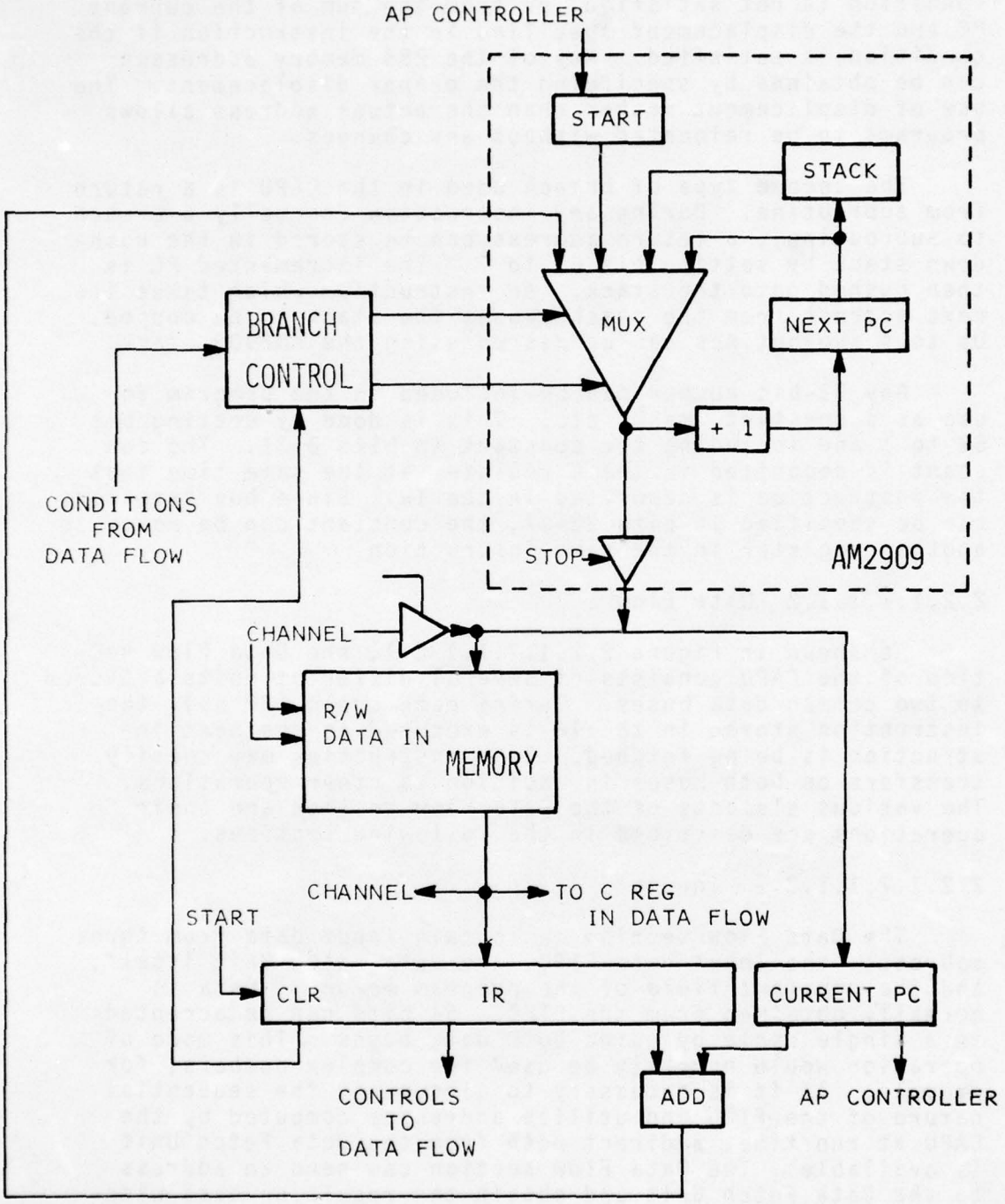


Figure 2.2.1.7.1.1.1-1. CAPU Program Sequencer

condition is not satisfied, or from the sum of the current PC and the displacement specified in the instruction if the condition is satisfied. Any of the 256 memory addresses can be obtained by specifying the proper displacement. The use of displacement rather than the actual address allows programs to be relocated without any changes.

The second type of branch used in the CAPU is a return from subroutine. During any instruction (normally a branch to subroutine), a return address can be stored in the push-down stack by setting bit 60 to 1. The incremented PC is then pushed onto the stack. An instruction which takes its next address from the stack causes the stack to be popped. Up to 4 subroutines can be nested using the AM2909 stack.

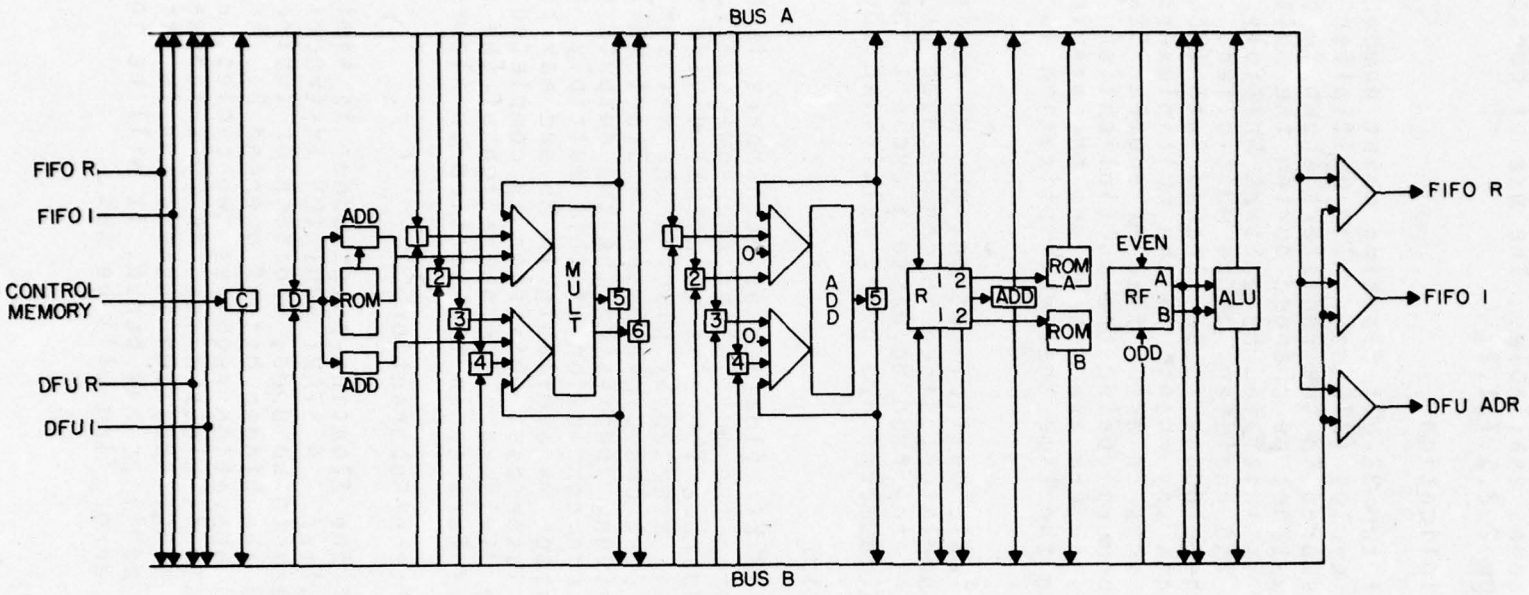
Any 32-bit number can be included in the program for use as a constant, mask, etc. This is done by setting bit 63 to 1 and including the constant in bits Ø-31. The constant is deposited in the C register at the same time that the instruction is deposited in the IR. Since bus transfers can be specified in bits 32-47, the constant can be moved to another register in the same instruction.

2.2.1.7.1.1.2 Data Flow

As shown in Figure 2.2.1.7.1.1.2-1, the Data Flow section of the CAPU consists of several different units attached to two common data buses. During each cycle (90 ns), the instruction stored in the IR is executed as the next instruction is being fetched. Each instruction may specify transfers on both buses in addition to other operations. The various elements of the Data Flow section and their operations are described in the following sections.

2.2.1.7.1.1.2.1 Inputs

The Data Flow section can obtain input data from three sources: the Input Data FIFO, the Data Fetch Unit itself, and the constant field of the program memory. Data is normally obtained from the FIFO. 64 bits can be accepted in a single cycle by using both data buses. This mode of operation would normally be used for complex numbers, for example. If it is necessary to circumvent the sequential nature of the FIFO and utilize addresses computed by the CAPU at run-time, a direct path from the Data Fetch Unit is available. The Data Flow section can send an address to the Data Fetch Unit and obtain the resulting data without disturbing the contents of the FIFO. This data path also permits 64-bit numbers to be accepted using both buses. In either case, if the data is not available, the CAPU clock



CAPU DATA FLOW BLOCK DIAGRAM
DWG NO. 4047-760308-IB REV.-A
FIGURE 2.2.1.7.1.1.2-1

is halted until it becomes available. The use of constants is described in Section 2.2.1.7.1.2.

2.2.1.7.1.1.2.2 Multiplication

Multiplication of two 32-bit floating point numbers can be performed in 3 cycles (270 ns). The multiplier and multiplicand must be stored in the registers shown in Figure 2.2.1.7.1.1.2-1 and must not be changed during the multiplication. Multiplication is initiated by a Start Multiply Instruction, which specifies the registers to be multiplied. A hardware counter causes the product to be clocked into (M5, M6) after 3 cycles, prevents any change in the multiplexers during that time, and sets an error flag if another Start Multiply instruction occurs during the multiplication. Normally the unused registers are loaded and the previous product removed during the time that multiplication is taking place.

In addition to simple multiplication, the M5 register can be used as an accumulator, since M5 can be used as an input. This eliminates the need to waste a cycle transferring data when several numbers are being multiplied together.

2.2.1.7.1.1.2.3 Division

Division of two 32-bit floating point numbers is performed in 7 cycles using the multiplier and some special-purpose division hardware. There is no actual divide instruction. Division is started by placing the divisor in the D register. At least one cycle later, a Start Multiply instruction must be given, specifying the two outputs from the divide unit. At the conclusion of this multiply, the reciprocal of the divisor is available in M5 and may be stored elsewhere for later use. Division is completed by multiplying M5 by the dividend. The operation of the division unit is explained further in the section on Algorithms.

2.2.1.7.1.1.2.4 Addition/Subtraction

The operation of the floating-point adder is similar to that of the multiplier. A START ADDITION instruction specifies the registers to be used, addition or subtraction (A + B or A - B), and the hidden bit of operand B. An addition (or subtraction) operation requires two cycles, at the end of which the result is clocked into A5 by a hardware counter. If another START ADDITION instruction is attempted immediately after an addition has begun, it will be ignored by the adder, and an error flag will be set.

Since register A5 is available as an input to the adder, it can be used as an accumulator. This saves a cycle which would otherwise be used in moving data if more than two numbers are being added.

Since the adder contains hardware for efficient normalization of floating-point numbers, it can be very useful in converting fixed-point numbers to floating-point and vice-versa. In order to facilitate these conversions, the hidden bit of operand B is controlled by bit 19 of the instruction word. For normal addition or subtraction, bit 19 is a zero, which causes the hidden bit to be a one. For the conversion algorithms, bit 19 is set to one, which makes the hidden bit zero. This is described in more detail in the Algorithms section.

2.2.1.7.1.1.2.5 Special Functions

The R register is used to obtain initial estimates used in the algorithms for the following functions: square root, logarithm, exponential, and trig functions. The operation of the hardware for each function is described in the Algorithm section.

2.2.1.7.1.1.2.6 Register File (RF)

A 16-register random access memory is provided for temporary storage of data. In addition, all shifting and logical operations performed by the ALU are performed on the outputs from the RF.

The RF is accessible from both buses. However, data from bus A can be written only into even-numbered registers, and data from bus B can be written only into odd-numbered registers. If data is being entered from both buses simultaneously, consecutive registers must be used, since only one address field is used to specify the write address.

The read operation is more versatile in that data from any of the 16 registers can be read out onto either bus. Since there are two read address fields, the buses have independent access to all of the registers. The use of the address fields is described more fully in the section on Instruction Format.

2.2.1.7.1.1.2.7 ALU

The ALU is used for all functions other than floating-point arithmetic. Its capabilities include integer addition and subtraction, all simple logic functions, and rotation of a 32-bit word by any number of places. All functions are

executed in a single cycle, and the output is available to both buses. Notice that different bus source codes are used for shifts and for arithmetic/logic operations.

Arithmetic or logic operations are performed on the A and B outputs from the RF by specifying a function code in the instruction. In addition, bit 3 of the instruction specifies whether the condition register is to be clocked to test the sign and F=Ø of the output. If the output is to be placed in another register, the bus transfer should be specified in the same instruction.

A shift operation is specified by selecting the shift source for either bus. The A output from the RF will be rotated left by the number of bits specified in the RF B output, and placed on the bus.

2.2.1.7.1.1.2.8 Outputs

Two outputs are available to the Output Data FIFO to enable 64-bit numbers to be stored in a single cycle. Either bus can select either output as its destination. Conflicts will be resolved in favor of bus A.

An output to the Data Fetch Unit is also available to permit the CAPU to send an address to the DFU and to obtain the resulting data without going through the FIFO. This mode might be used for large look-up tables, for example. Either bus may select the DFU output as a destination. Conflicts will be resolved in favor of bus A.

2.2.1.7.1.2 CAPU Instruction Format

TYPE

TYPE

63

Bit 63
- Ø - Normal instruction
- 1 - Constant - Bits Ø - 31 are put in the C Register and may be moved using the same instruction. Next address is chosen and transfers on both buses are allowed, but arithmetic operations cannot be used, since they are normally specified in bits 0 - 31.

NEXT ADDRESS

TYPE	PUSH	CONDITIONS	DISPLACEMENT

62 61 60 59 58 57 56 55 54 53 52 51 50 49 48

Bits 61-62 Type
- 00 Address from PC
- 01 Address from Start Register
- 10 Address from Stack (POP)
- 11 Branch on Condition

Bit 60 Push
- Ø Normal
- 1 Push incremented PC onto Stack

Bits 56-59 Condition

0000	Unconditional	1000	
0001	ALU out = Ø	1001	ALU out ≠ 0
0010	ALU sign = Ø	1010	ALU sign = 1
0011	A5 sign = Ø	1011	A5 sign = 1
0100	Flag A	1100	
0101	Flag B	1101	
0110	Flag C	1110	
0111	Flag D	1111	

Bits 48-55 Displacement: Added to current PC to determine next address in the case of a branch

BUS A

SOURCE	DESTINATION

47 46 45 44 43 42 41 40

Bit 44-47 Source

0000	ZERO	1000	M5
0001	FIFO REAL	1001	M6
0010	FIFO IMAG	1010	A5
0011	DFU REAL	1011	LOG C
0100	DFU IMAG	1100	SQRT
0101	SHIFT OUT	1101	ROM A
0110	ALU OUT	1110	RF A OUT
0111	CONSTANT REG	1111	RF B OUT

Bits 40-43 Destination				
	0000	NONE	1000	DATA FETCH UNIT ADR
	0001	M1	1001	A1
	0010	M2	1010	A2
	0011	M3	1011	A3
	0100	M4	1100	A4
	0101	--	1101	FIFO REAL
	0110	D	1110	FIFO IMAG
	0111	R	1111	RF A (EVEN)

SET FLAGS

FLAG A	FLAG B	FLAG C	FLAG D				
				0	1	0	1
47	46	45	44	43	42	41	40

A 1 in the appropriate bit causes the corresponding flag to be set. A Ø implies that the flag is not changed. These flags can be used for conditional branches or for communication with the IOAG.

Note that transfers cannot be made on Bus A when flags are being set.

CLEAR FLAGS

FLAG A	FLAG B	FLAG C	FLAG D				
				0	1	0	1
39	38	37	36	35	34	33	32

A 1 in the appropriate bit causes the corresponding flag to be cleared. A Ø implies that the flag is not changed.

Note that transfers cannot be made on Bus B when flags are being cleared.

Attempting to set and clear a particular flag at the same time will cause it to toggle (i.e. switch to the opposite state).

BUS B

SOURCE				DESTINATION			
39	38	37	36	35	34	33	32

Bits 36-39				
	0000	ZERO	1000	M5
	0001	FIFO REAL	1001	M6
	0010	FIFO IMAG	1010	A5
	0011	DFU REAL	1011	LOG D
	0100	DFU IMAG	1100	SQRT
	0101	SHIFT OUT	1101	ROM B
	0110	ALU OUT	1110	RF A OUT
	0111	CONSTANT REG	1111	RF B OUT

Bits 32-35 Destination	0000	NONE	1000	DATA FETCH UNIT ADR
	0001	M1	1001	A1
	0010	M2	1010	A2
	0011	M3	1011	A3
	0100	M4	1100	A4
	0101	--	1101	FIFO REAL
	0110	D	1110	FIFO IMAG
	0111	R	1111	RF A (ODD)

Multiplier

S	A MUX	B MUX	F
31	30 29	28 27	26

Bit 31 Start	Ø	NO CHANGE IN OPERATION
	1	START MULTIPLICATION

NOTE: A multiplication cannot be started within 2 instructions of the previous start

Bits 29-30 A MUX	00	M1
	01	M2
	10	M5
	11	D OUTPUT A
Bits 27-28 B MUX	00	M3
	01	M4
	10	M5
	11	D OUTPUT B
Bit 26 Fix/Float	Ø	FLOATING POINT MULTIPLY
	1	FIXED POINT MULTIPLY

Product is available in (M5, M6) 3 cycles after Start

Adder

S	A/S	A MUX	B MUX	H
25	24	23 22	21 20	19

Bit 25 Start	Ø	NO CHANGE IN OPERATION
	1	START ADDITION

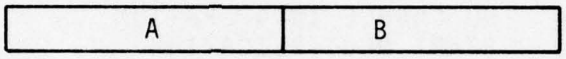
NOTE: An addition cannot be started within 2 instructions of the previous Start.

Bit 24 Add/Subtract	Ø	ADD	A + B
	1	SUBTRACT	A - B

Bits 22-23 A MUX	00	ZERO
	01	A1
	10	A2
	11	A5
Bits 20-21 B MUX	00	ZERO
	01	A3
	10	A4
	11	A5
Bit 19 Hidden Bit	Ø	NORMAL (BOTH HIDDEN BITS = 1)
	1	HIDDEN BIT A=1, HIDDEN BIT B=Ø

Sum is available in A5 2 cycles after Start

RAM Addresses



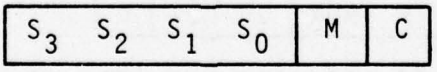
18 17 16 15 14 13 12 11

The RAM A and RAM B outputs to both buses and to the ALU are specified by the A and B fields. Each field may specify an output from any of the 16 registers.

In the case of A write operation, only bits 16-18 are used to specify the register or registers. Data on Bus A can be written only into even-numbered registers. Data on Bus B can be written only into odd-numbered registers. Thus, if bits 16-18 contain 110, data on Bus A can be written into register 12, while data on Bus B can be written into register 13.

Shifted outputs can be obtained by selecting the shift output on either bus. The RAM A output will be rotated left by the number of bits specified in the RAM B output.

ALU Function



8 7 6 5 4 3

The following functions are performed on the contents of the A and B registers as specified by bits 4-8. (For shifts see page A7).

SELECTION				ACTIVE HIGH DATA		
				M=H	M=L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	C_n=H (no carry)	C_n=L (with carry)
L	L	L	L	F=A		F=A PLUS 1
L	L	L	H	F=A+B		F=(A+B) PLUS 1
L	L	H	L	F=AB		F=(A+B) PLUS 1
L	L	H	H	F=0		F=ZERO
L	H	L	L	F=AB		F=A PLUS AB PLUS 1
L	H	L	H	F=B		F=(A+B) PLUS AB PLUS 1
L	H	H	L	F=A + B		F=A MINUS B
L	H	H	H	F=AB		F=AB
H	L	L	L	F=A+B	F=A PLUS AB	
H	L	L	H	F=A +B	F=A PLUS B	
H	L	H	L	F=B	F=(A+B)PLUS AB	
H	L	H	H	F=AB	F=AB MINUS 1	
H	H	L	L	F=1	F=A PLUS A	
H	H	L	H	F=A+B	F=(A+B) PLUS A	
H	H	H	L	F=A+B	F=(A+B) PLUS A	
H	H	H	H	F=A	F=A MINUS 1	

C_n = S3

Bit 3 Clock Conditions	Ø - No clock 1 - After the operation has been performed, the condition register is clocked to test the sign of F and F = Ø

ROM Function

ROM

2 1 0

000
001 SQRT
010 LOG
011 EXP
100 SIN
101 ARCSIN
110
111

2.2.1.7.1.3 CAPU Algorithms

The CAPU contains special-purpose hardware to permit the efficient calculation of several functions. For each of these functions, a short program, stored in the program memory, is required to obtain the final value of the function. In general, a first approximation of the value is obtained from a ROM, and an iterative procedure is used which quickly converges to the final value. In the following sections, each algorithm is justified and the procedures for using the special-purpose hardware are described.

2.2.1.7.1.3.1 Division

Division is performed by multiplying the dividend by the reciprocal of the divisor. The reciprocal is as follows.

Let $q = \frac{1}{d}$ where $\frac{1}{2} \leq d < 1$.

Let $d = d_A + 2^{-12} d_B$ where $\frac{1}{2} \leq d_A < 1$, $d_B < 1$

and d_A consists of 11 bits plus the hidden bit

Then $$q = \frac{1}{d_A + 2^{-12} d_B} \quad \frac{d_A - 2^{-12} d_B}{d_A - 2^{-12} d_B} = \frac{d_A - 2^{-12} d_B}{d_A^2 - 2^{-24} d_B^2}$$

$$\simeq (d_A - 2^{-12} d_B) \frac{1}{d_A^2}$$

The basic algorithm for finding q is simply to obtain $\frac{1}{d_A^2}$ from a 2048 word ROM, find $d_A - 2^{-12} d_B$ with a 24-bit adder, and multiply the results. The sign and exponent can easily be obtained at the same time.

In effect, the basic algorithm uses $1/d_A$ as a first approximation to q. A more accurate result can be obtained by using $1/d_A + 2^{-13}$ as a first approximation. This results in

$$q \simeq (d_A + 2^{-13} - 2^{-12}d_B) \frac{1}{d_A (d_A + 2^{-13})}$$

The ROM now contains the value of $\frac{1}{d_A (d_A + 2^{-13})}$ and a 24 bit adder is used to obtain $(d_A + 2^{-13}) - 2^{-12}d_B$. Again, the sign and exponent are obtained at the same time. A block diagram of the hardware required for division is shown in Figure 2.2.1.7.1.3.1-1. Since the two factors of q can be obtained in one cycle, division requires a total of seven cycles.

2.2.1.7.1.3.2 Floating-Point to Integer Conversion and Vice Versa

No special hardware is required for converting numbers from floating-point to integer format. However, the most difficult part of the conversion can be done efficiently using the floating-point adder. The conversion can be divided into several steps:

1) Test the sign bit

2) For positive numbers, add to 2^{23}.

   For negative numbers, add to 2^{24}.

3) For positive numbers, AND the result with
   00000000000000000111111111111111.

   For negative numbers, OR the result with
   11111111111111110000000000000000.

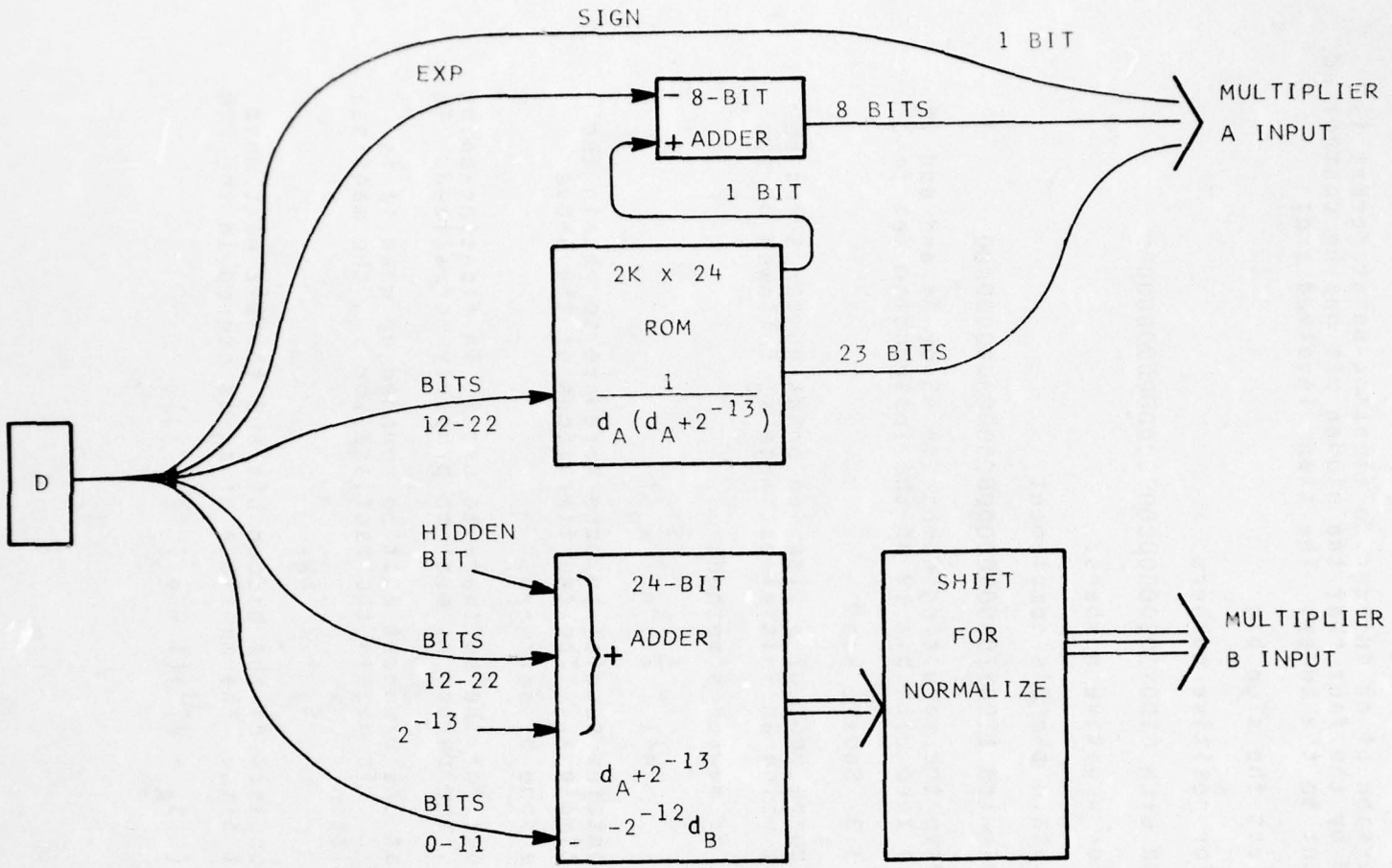


Figure 2.2.1.7.1.3.1-1. Division Hardware

Conversion of an integer to floating-point format is simplified by the fact that the hidden bit can be controlled at the input to the adder. The steps involved are:

1) Test the sign bit

   For positive numbers:

2) OR with 01001100000000000000000000000000

   For negative numbers:

2a) obtain the 2's complement

2b) OR with 11001100000000000000000000000000

3) Move the resulting number to A3 or A4 and add it to zero with bit 19 of the instruction set to 1.

2.2.1.7.1.3.3 Square Root

The square root of a floating point number S can be obtained by using an initial estimate x_0 followed by two iterations of Newton's method:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

The CAPU contains special-purpose hardware to obtain the initial estimate x_0. The two iterations of the above formula are done by software.

In order for the software to use it in floating-point arithmetic, the number x_0 must be properly normalized. This implies that the exponent must be rounded up when it is divided by 2. To obtain the mantissa for x_0, the mantissa of S is written

$$S_A + 2^{-7}S_B,$$

where S_A consists of the hidden bit and the six next most significant bits. The mantissa of x_0 is stored in the ROM as

$$(\ S_A + 2^{-8})(1 + e.(\ \ 2 - 1)),$$

where e_0 is the least significent bit of the exponent of S.

A block diagram of the hardware is shown in Fig. 2.2.1.7.1.3.3-1. A total of 27 cycles should be required to obtain the square root x_2.

2.2.1.7.1.3.4 Natural Logarithm

The natural logarithm of a number x is obtained by the following expansion.

Let $x = (M_a + M_b)\ 2^E$ where M_a consists of the hidden bit and the next 8 most significant bits. Then

$$\ln x = E \ln 2 + \ln (M_a + mM_b)$$

$$= E \ln 2 + \ln M_a + \ln (1 + \frac{M_b}{M_a})$$

Since $\frac{M_b}{M_a} < 2^{-8}$,

$$\ln x = E \ln 2 + \ln M_a + \frac{M_b}{M_a} - \frac{1}{2} \frac{M_b}{M_a}^2$$

In the CAPU, ROMs are provided to supply the values of E ln 2 and ln Ma. Ma is also available to the bus as a normalized floating point number. M_b is available to the bus, but it must be normalized by software. This can be done by adding it to zero with the hidden bit equal to zero. The hardware is shown in Figure 2.2.1.7.1.3.4-1. The above formula can be evaluated in 18 cycles.

2.2.1.7.1.3.5 Exponential

Exponentials can be obtained as follows.

Let the fixed-point representation of x be

x = xxxxxxx. xxxxxxxx xxx .....

x_1 x_2 x_3

That is, $x = x_1 + x_2 + x_3$, where x_1 is a 7-bit integer,

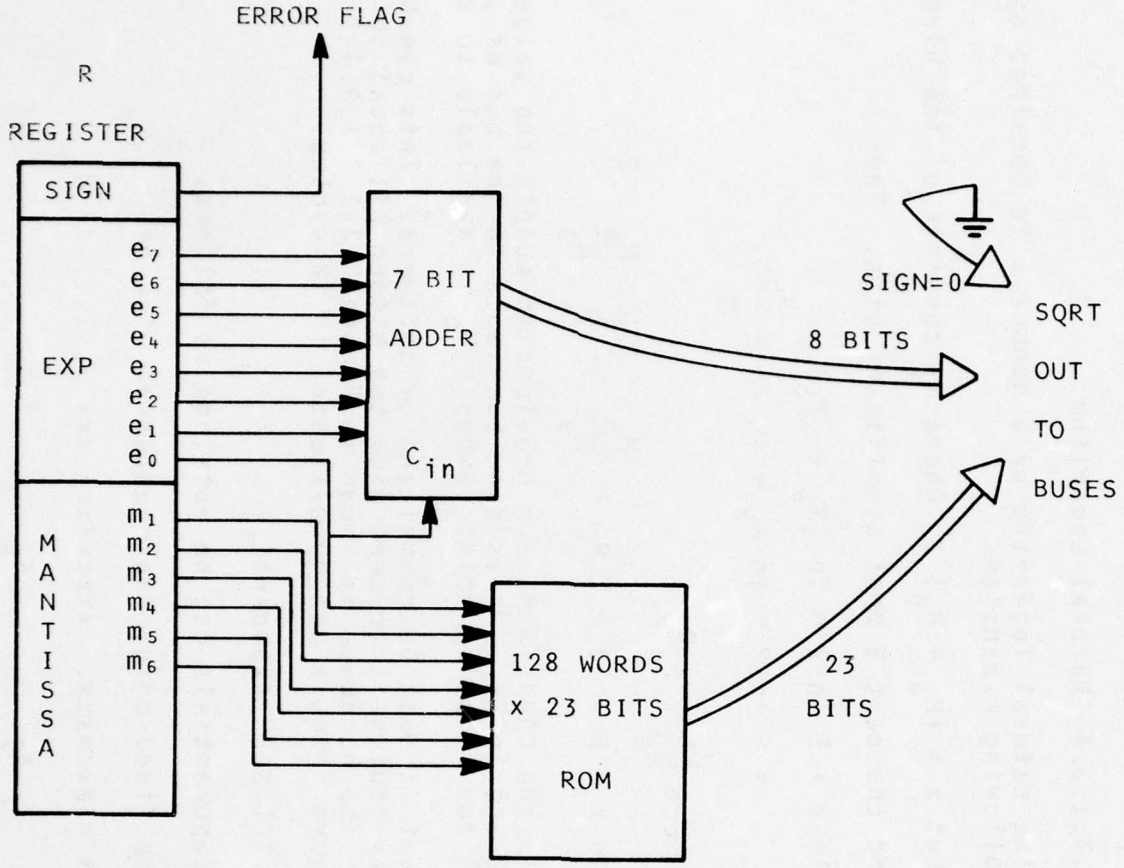


Figure 2.2.1.7.1.3.3-1. Square Root Hardware

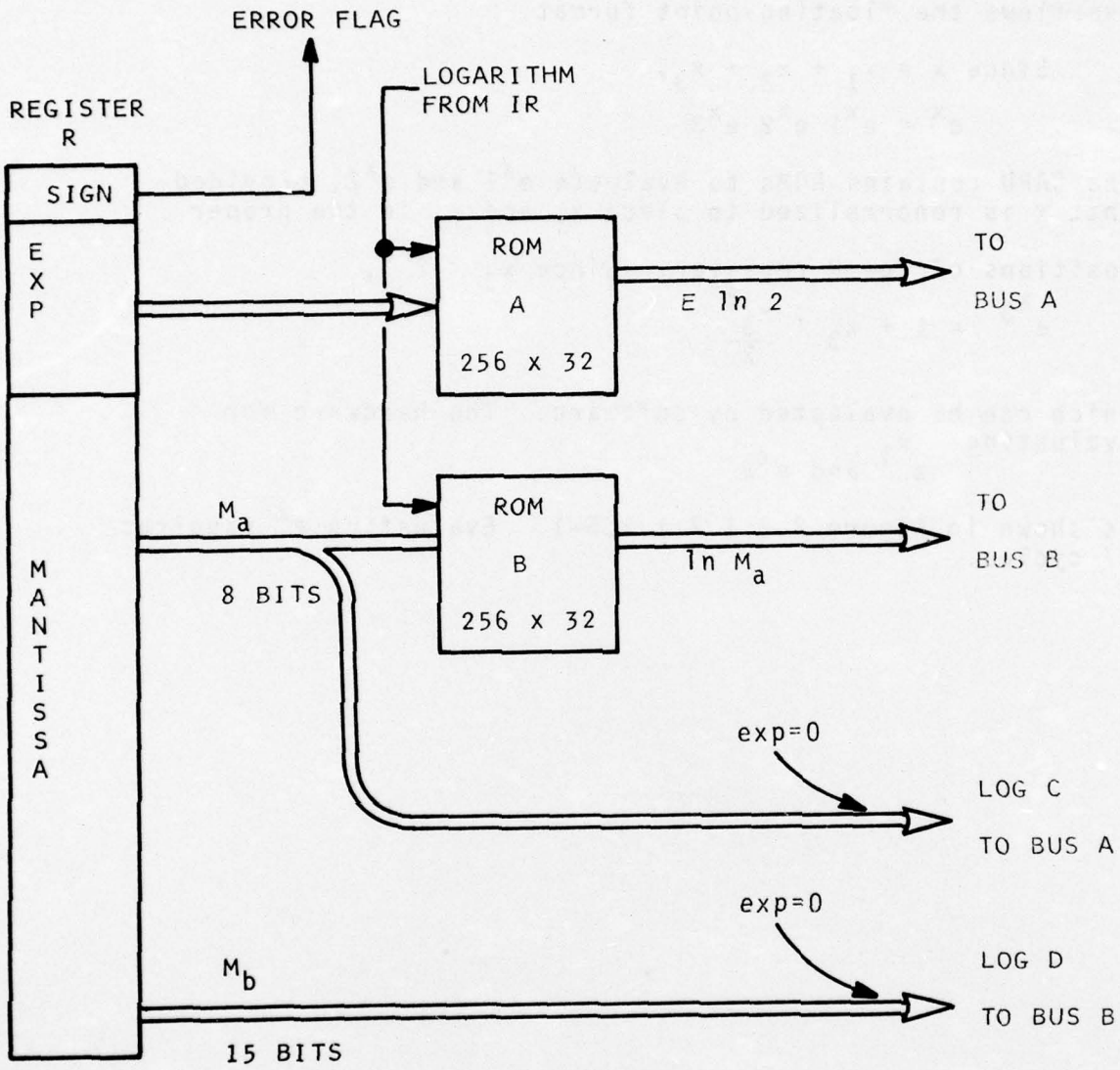


Figure 2.2.1.7.1.3.4-1. Logarithm Hardware

x_2 is an 8-bit fraction, and x_3 contains all of the remaining bits in x. Note that, if $x > 88$, $e^x > 2^{38}$, which overflows the floating-point format.

Since $x = x_1 + x_2 + x_3$,

$$e^x = e^{x_1} e^{x_2} e^{x_3}$$

The CAPU contains ROMs to evaluate e^{x_1} and e^{x_2}, provided that x is renormalized to place x_1 and x_2 in the proper positions of the R register. Since $x_3 < 2^{-8}$,

$$e^{x_3} \simeq 1 + x_3 + \frac{x_3^2}{2}$$

which can be evaluated by software. The hardware for evaluating e^{x_1} and e^{x_2} is shown in Figure 2.2.1.7.1.3.5-1. Evaluating e^x requires 27 cycles.

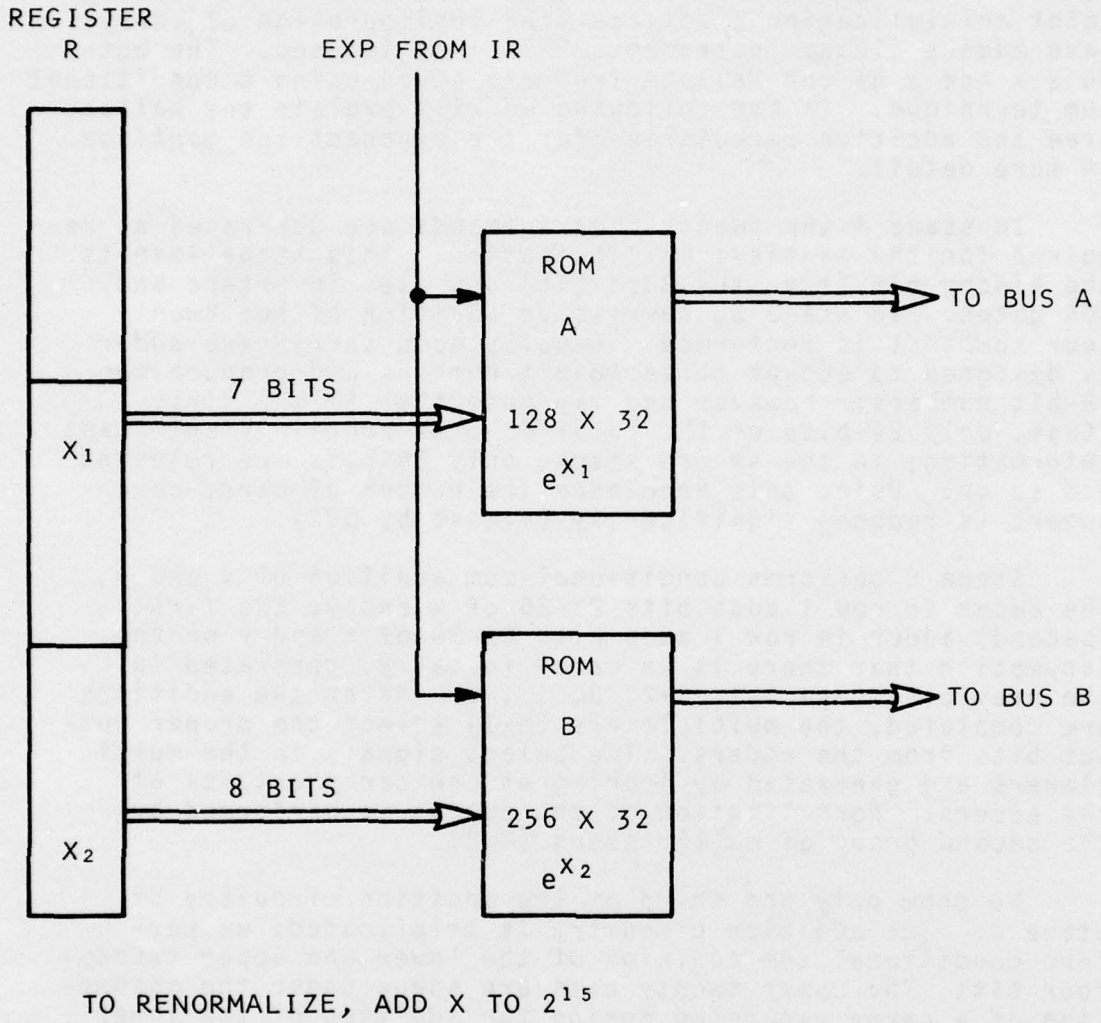


Figure 2.2.1.7.1.3.5-1. Exponential Hardware

2.2.1.7.1.4 Floating-Point Multiplier

A block diagram of the CAPU Floating Point Multiplier is shown in Figure 2.2.1.7.1.4-1. To perform fast floating-point multiplication a Wallace tree configuration of carry-save adders (Texas Instruments SN74H183) is used. The outputs x and y of the Wallace tree are added using a conditional sum technique. In the following we will explain the Wallace tree and addition circuitries for the exponent and mantissa in more detail.

In stage A the twenty-four summands are generated as required for the mantissa multiplication. This stage inserts the hidden bit (i.e. the 23rd bit) and uses inverters and NOR gates. In stage B, carry-save addition of the twenty-four summands is performed. Usually each carry-save adder is designed to accept three 48-bit numbers and produce two 48-bit numbers: however one may note that in the first stage, only 24 bits of the total 48 bits contain significant information; in the second stage, only 26 bits are relevant and so on. Using this knowledge the number of carry-save adders is reduced significantly (almost by 50%).

Stage C performs conditional-sum addition of x and y. The adder in row 1 adds bits 23-26 of x and y; the first (second) adder in row i adds bits 27-30 of x and y on the assumption that there is no carry (a carry) generated in the previous stage i-1, i=2, 3, .., 6. After the additions are completed, the multiplexers (Mx1) select the proper output bits from the adders. The select signals to the multiplexers are generated by looking at the carry outputs of the adders. Normalization of the output is performed by the second group of multiplexers (Mx2).

We show only one third of the addition circuitry of Stage C. The addition circuitry is triplicated; we perform conditional sum addition of the lower and upper twenty-four bits. The upper twenty bits are added under the assumption of a carry generated during the addition of the lower twenty-four bits and also under the assumption of no carry; once the lower twenty-four bits are added, the appropriate sum of the upper half is selected. Thus two 48-bit numbers are added in approximately 80 ns using 40 IC chips.

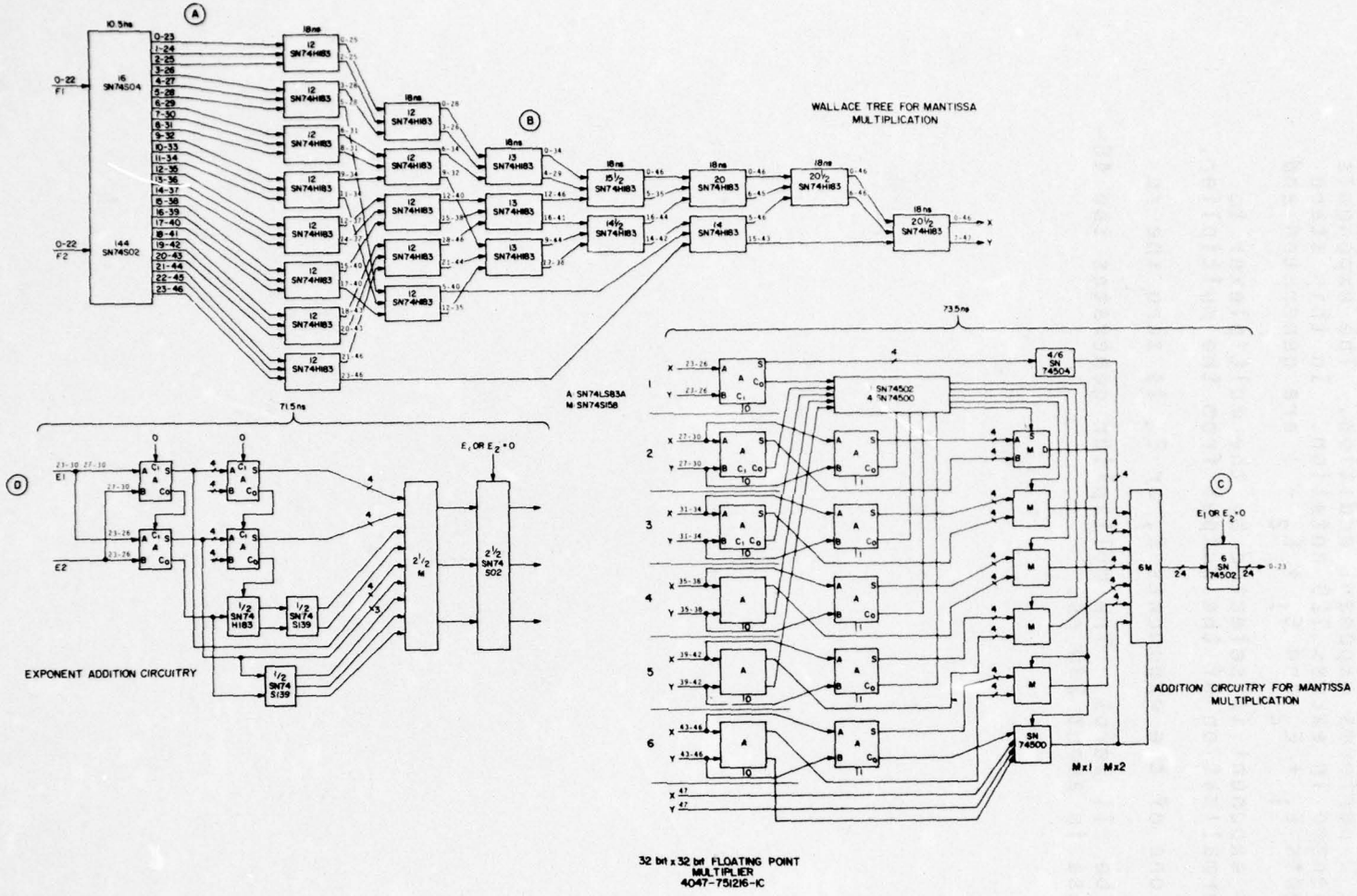


Figure 2.2.1.7.1.4-1. 32-Bit x 32-Bit Floating-Point Multiplier

Stage D performs exponent addition. The exponents are represented in excess 128 notation. In this stage two exponents $E_1 + E_2$ and $E_1 + E_2 - 1$ are generated and the proper exponent is selected by the multiplexer to achieve normalization of the output from the multiplier.

When one of the exponents E_1 or E_2 is zero the result will be all zeros. The multiplier generates the 48-bit mantissa in about 240 ns.

2.2.1.7.1.5 Floating-Point Adder

2.2.1.7.1.5.1 Functional Description

The Floating-Point Adder is designed to perform addition and subtraction of normalized floating-point numbers as rapidly as possible. Addition of floating-point numbers can be divided into several steps:

1) Subtract exponents to determine which number has the smaller magnitude and how much smaller it is.

2) Shift the mantissa of the smaller number to make the exponents equal.

3) Perform the actual addition or subtraction.

4) Locate the most significant bit in the resulting mantissa.

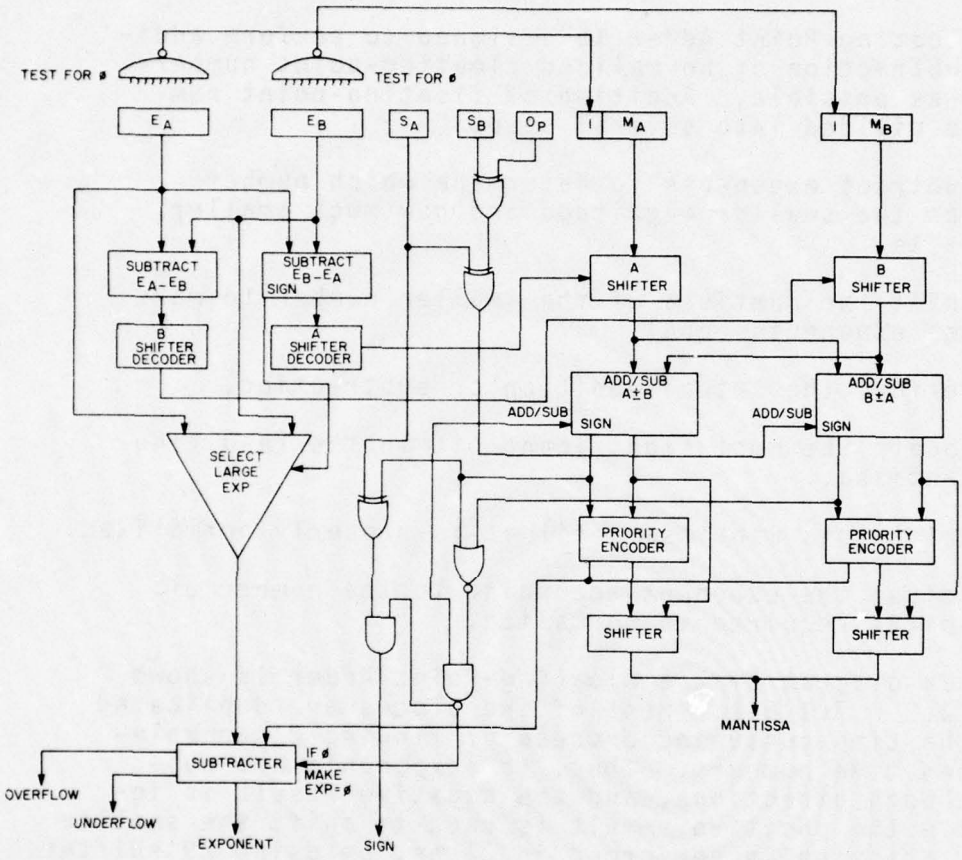
5) Shift this mantissa until it is properly normalized.

6) Adjust the exponent according to the number of shifts required to normalize.

A block diagram of the Floating-Point Adder is shown in Figure 2.2.1.7.1.5-1. Most of the blocks are duplicated to avoid the time-consuming process of finding 2's-complements of negative numbers. Thus, the exponents are subtracted in both directions, and the negative result is ignored, while the positive result is used to shift the smaller mantissa. Shifting is performed in 21 ns, by using 29 shifter circuits in parallel.

During the exponent subtraction and shifting process, the actual operation to be performed on the mantissa is determined. This is done simply by exclusive ORing the signs and the overall operation code. Thus, if the signs are different and the operation is subtraction, the mantissa must be added.

In the cases for which the mantissa must be subtracted, the subtraction must be performed in both directions. This is due to the fact that the final mantissa must be positive, and additional time would be wasted if a negative number had to be complemented.



FLOATING POINT ADDER

Figure 2.2.1.7.1.5-1. Floating Point Adder

The resulting mantissas are fed into priority encoders and shifters. The positive mantissa is examined by the priority encoder to find its most significant bit. Using this information, the shifter shifts the mantissa to place the most significant bit in the hidden bit location. At the same time the output of the priority encoder is used to correct the exponent for the shifting operation, and the sign of the A±B operation is used in obtaining the output sign.

In this system a normalized floating-point number is considered to be a zero if and only if its exponent is zero. Since it is necessary to avoid putting the hidden bit into the mantissa of a number which is zero, the exponents are tested at the beginning of the operation. After any subtraction of the mantissas, the result is tested for zero by comparing the signs of the two subtractions. If they are both zero, the result must be zero, and in that case the exponent and sign are set to zero also.

2.2.1.7.1.5.2 Floating Point Adder - Propagation Delays

FUNCTION	DELAY
Exponent Subtraction	29 ns
Shift Decoding	12 ns
Shifting	21 ns
Addition/Subtraction	45 ns
Priority Encoder	19 ns
Exponent Subtraction	29 ns
TOTAL:	155 ns

2.2.1.7.2 Input-Output Address Generator (IOAG)

The function of the IOAG is to compute the addresses of the input and output data of the Arithmetic Processor, and deposit them in the Input Address FIFO and Output Address FIFO respectively. In most cases, these addresses will consist of several sequences interleaved in some order, where each sequence represents successive elements of a data structure such as an array. Generally, each sequence is obtained by adding a fixed increment for each cycle. In addition to these sequences, the IOAG would also have to output addresses of scalars. A specialized bit-reversal feature is required in addition for the "shuffle" operation in FFT computations.

Two alternative IOAG designs were developed: the first is a general-purpose processor that is basically a stripped down version of the CAPU without floating point hardware. It has one bus, and the word size is 24 bits. The second alternative is a special-purpose hardware module designed to perform just the kinds of operations listed above.

2.2.1.7.2.1 The Programmable IOAG - Functional Description

The IOAG is a programmable unit used to produce the 24-bit addresses for the Data Fetch Unit and Data Store Unit. It is capable of storing up to 32 numbers in its registers, and of performing addition, subtraction, shifting, and bit reversal on these numbers.

The IOAG consists of two major sections: a Program Sequencer and a Data Flow section. Note that the Program Sequencer is almost identical to the sequencer used in the CAPU. This simplifies the design of both hardware and software.

In operation, programs are loaded into the IOAG program memory by the Channel. The AP Controller can then load a starting address into the IOAG and start the program. The program provides constants to the Data Flow section, which uses them to obtain addresses. The addresses are then placed in the FIFOs to the Data Fetch or Store Units.

Most programs for the IOAG will consist of simple loops. For this reason, the program memory is small (256 words) and the branching facilities are limited. However, since all branch locations are calculated from the current PC, programs may be relocated anywhere in memory without changes. The program memory may be expanded beyond 256 words if required.

The following sections describe the normal operation of the Program Sequencer and the Data Flow section. Additional details may be found in section 2.2.1.7.2.1.3.

2.2.1.7.2.1.1 Program Sequencer

A block diagram of the Program Sequencer is shown in Figure 2.2.1.7.2.1.1-1. The sequencer consists of a 256 word x 64 bit program memory, an instruction register, and various circuits for determining the next address.

In operation, the Channel will begin by loading a program into the IOAG memory. It does this by disabling the AM2909 tri-state outputs and using its own inputs to the memory. To start a program, the AP Controller places

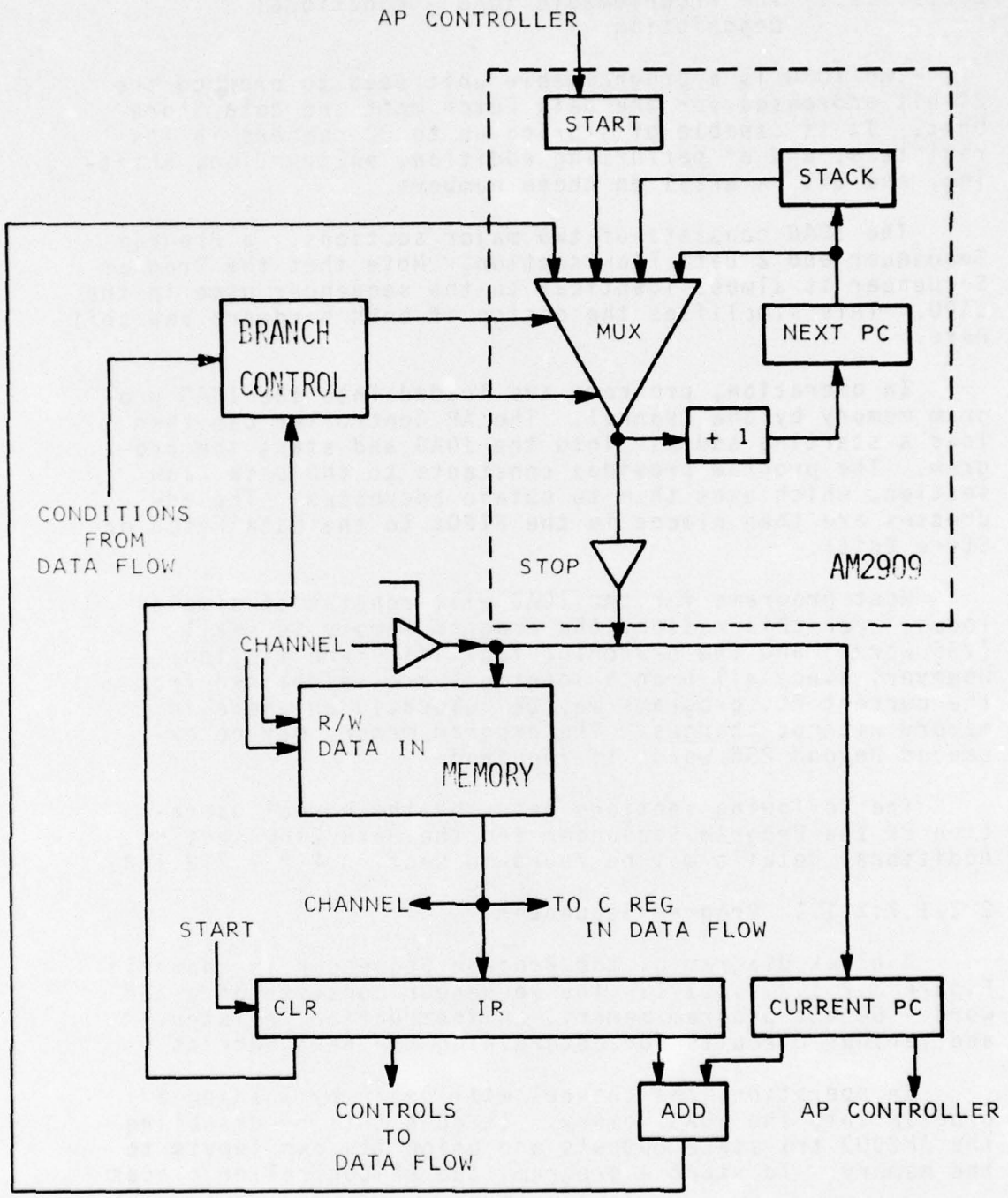


Figure 2.2.1.7.2.1.1-1. IOAG Program Sequencer

the starting address in the START register and sets the RUN bit, which initializes the IR and enables the IOAG clock. When the IR contains 01 in the Next Address Type field, the next address is taken from the START register.

During execution of the program, the next address is normally obtained from the incremented PC. However, two types of branches can be used. For a conditional branch, the next address is taken from the incremented PC if the condition is not satisfied, or from the sum of the current PC and the displacement specified in the instruction of the condition is satisfied. Any of the 256 memory addresses can be obtained by specifying the proper displacement. The use of displacement rather than the actual address allows programs to be relocated without any changes.

The second type of branch used in the IOAG is a return from subroutine. During any instruction (normally a branch to subroutine), a return address can be stored in the pushdown stack by setting bit 60 to 1. The incremented PC is then pushed onto the stack. An instruction which takes its next address from the stack causes the stack to be popped. Up to 4 subroutines can be nested using the AM2909 stack.

Any 24-bit number can be included in the program for use as a constant, mask, etc. This is done by setting bit 63 to 1 and including the constant in bits Ø - 23. The constant is deposited in the C register at the same time that the instruction is deposited in the IR. Since bus transfers can be specified in bits 44 - 47, the constant can be moved to another register in the same instruction.

2.2.1.7.2.1.2 Data Flow

The Data Flow section of the IOAG, shown in Figure 2.2.1.7.2.1.2-1, consists of three primary blocks: a RAM containing 32 registers, a shifter, and an ALU. In addition, the blocks are connected by a common 24-bit bus, and a register is provided to enter constants from the program. The following sections describe the operation of the Data Flow section.

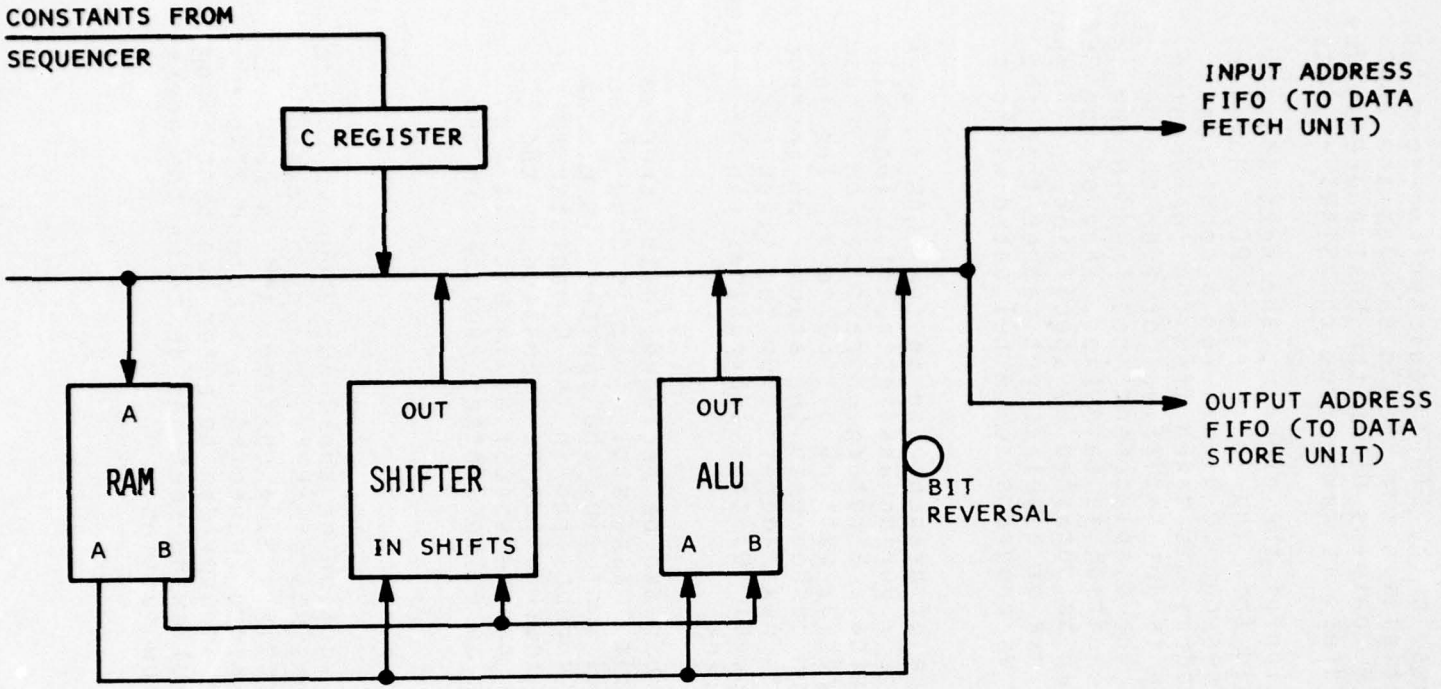


Figure 2.2.1.7.2.1.2-1. IOAG Data Flow

2.2.1.7.2.1.2.1 IOAG Data Flow Section Operation

Inputs to the Data Flow section must come from the program memory. Normally, a base value and an increment will be specified in the program, and successive addresses will be obtained from them. Any constant is loaded into the C register when the instruction is fetched. During the same cycle, the constant can be transferred to another register or to an output.

2.2.1.7.2.1.2.2 Registers

A set of 32 registers is provided for temporary storage. Outputs are available simultaneously from any two registers, using the A and B address fields in the instruction. The A field is also used to specify the register into which data is written if the RAM is specified as the bus destination. Thus, if two numbers are added together, for example, the sum cannot be placed in a third register.

2.2.1.7.2.1.2.3 Shifting

A shift operation is specified by selecting the shifter as the bus source. The number to be shifted must be stored in the RAM and specified by the A field. This number is rotated left by the number of bits specified in the RAM B output and placed on the bus.

2.2.1.7.2.1.2.4 ALU

The ALU can be used for addition, subtraction, logical operations, or simply to put a RAM output on the bus. In addition, the ALU output can be tested for sign and zero for later use in conditional branches. This test can be performed whether or not the ALU output is placed on the bus. Thus, the contents of one register can be tested while another register is being loaded from the program, for example.

2.2.1.7.2.1.2.5 Bit Reversal

For use in FFT operations, a path is provided from the RAM A output to the bus which reverses the bits in the number placed on the bus. All 24 bits are reversed. Therefore, if the reversal of a smaller number of bits is required, an additional shift operation must be used.

2.2.1.7.2.1.2.6 Outputs

Since a separate bit in the instruction is provided for each bus destination, the number on the bus may be transferred to either or both outputs at any time. Normally, the addresses produced by the IOAG will be sums of a cumulative address and an increment, so the output will come from the ALU. But the output of the shifter or the bit reverser can also be used, or an address can be transferred directly from the program to an output via the C register.

2.2.1.7.2.1.3 IOAG Instruction Format

TYPE

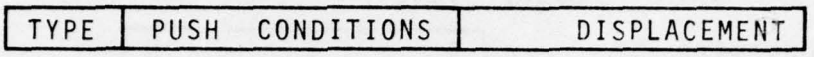


63

Bit 63 — Ø - Normal Instruction
1 - Constant - Bits Ø - 23 are put in the C register and may be used during the same instruction.

Note that flags cannot be set or cleared at the same time.

NEXT ADDRESS



62 61 60 59 58 57 56 55 54 53 52 51 50 49 48

Bits 61-62 Type
- 00 Address from PC
- 01 Address from START Register
- 10 Address from stack (POP)
- 11 Branch on Condition

Bit 60 Push
- Ø Normal
- 1 Push incremented PC onto stack

Bits 56-59 Condition

0000	UNCONDITIONAL	1000	
0001	ALU OUT = Ø	1001	ALU OUT ≠ Ø
0010	ALU SIGN = Ø	1010	ALU SIGN = 1
0011		1011	
0100		1100	
0101		1101	
0110	FLAG C	1110	FLAG E
0111	FLAG D	1111	FLAG F

Bits 48-55 Displacement — Added to current PC to determine next address in the case of a Branch

BUS



47 46 45 44 43

Bits 46-47 Source
- 00 ALU
- 01 C Register
- 10 Shifter
- 11 Bit Reversal

Bits 43-45 Destinations

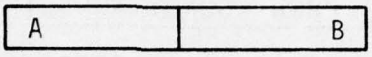
Bit 45 = 1 RAM is a destination. (The Register is specified by the A Field)

Bit 44 = 1 Input Address FIFO is a destination. (Address goes to Data Fetch Unit).

Bit 43 = 1 Output Address FIFO is a destination. (Address goes to Data Store Unit)

If bits 43-45 are all Ø, no bus transfer takes place.

RAM ADDRESSES



42 41 40 39 38 37 36 35 34 33

During each cycle, the Registers specified by the A and B address fields are read, and the contents are used as inputs to the ALU and the shifter. The contents of the A register are also used for bit reversal.

In the case of a write operation (specified by setting bit 45 = 1), the A address field specifies the register into which data is written from the bus.

Note that the result of an operation on two registers cannot be placed into a third register, since only two address fields are available.

MODE



32 31

Since each byte is individually accessible in CWS or PELS, addresses supplied to the Data Fetch Unit or Data Store Unit must include mode bits to specify the number of bytes to be read or written.

Any instruction which specifies the input address FIFO or output address FIFO as a destination must also specify the mode bits. These bits are then attached to the 24-bit

address and send to the FIFO.

MODE

00	64-bits
01	32-bits
10	16-bits
11	8-bits

ALU Function

S_3	S_2	S_1	S_0	M	C
30	29	28	27	26	25

The following functions are performed on the contents of the A and B registers, as specified by bits 26-30:

SELECTION				ACTIVE HIGH DATA		
				M=H	M=L; ARITHMETIC OPERATIONS	
				LOGIC	C_n=H	C_n=L
S3	S2	S1	S0	FUNC-TIONS	(no carry)	(with carry)
L	L	L	L	F=A		F=A PLUS 1
L	L	L	H	F=A+B		F=(A+B) PLUS 1
L	L	H	L	F=AB		F=(A+B) PLUS 1
L	L	H	H	F=0		F=ZERO
L	H	L	L	F=AB		F=A PLUS AB PLUS 1
L	H	L	H	F=B		F=(A+B) PLUS AB PLUS 1
L	H	H	L	F=A + B		F=A MINUS B
L	H	H	H	F=AB		F=AB
H	L	L	L	F=A+B	F=A PLUS AB	
H	L	L	H	F=A +B	F=A PLUS B	
H	L	H	L	F=B	F=(A+B)PLUS AB	
H	L	H	H	F=AB	F=AB MINUS 1	
H	H	L	L	F=1	F=A PLUS A	
H	H	L	H	F=A+B	F=(A+B) PLUS A	
H	H	H	L	F=A+B	F=(A+B) PLUS A	
H	H	H	H	F=A	F=A MINUS 1	

C_n = S3

Bit 25 — Clock Conditions

Ø - No Clock
1 - After the operation has been performed, the condition register is clocked to test the sign of F and F = Ø.

FLAGS

SET				CLEAR			
C	D	E	F	C	D	E	F
23	22	21	20	19	18	17	16

Set Flags Bits 20-23 — A 1 in any bit causes the corresponding flag to be set.

Clear Flags Bits 16-19 — A 1 in any bit causes the corresponding flag to be cleared.

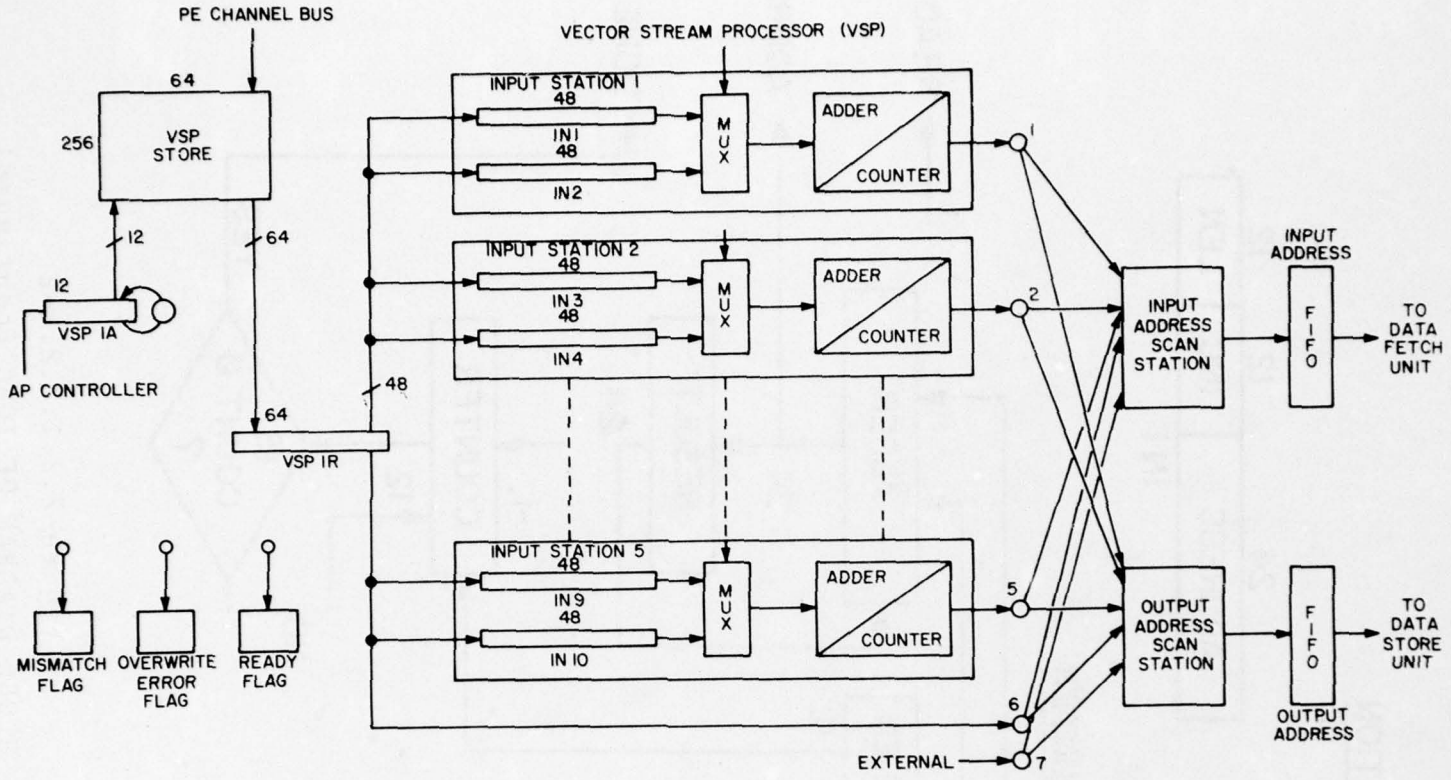
Placing 1's in both the set and clear bits for a particular flag causes it to toggle (i.e. switch to the opposite state).

Note that Flags C and D are shared with the CAPU, and can be used for communication between the two units.

2.2.1.7.2.2 Special-Purpose Input/Output Address Generator (IOAG)

The block diagram of the IOAG is shown in Figure 2.2.1.7.2.2-1. The function of the IOAG is to generate addresses of input as well as output operands and deposit them in Input and Output Address FIFOs. The IOAG is capable of generating five addresses in approximately 100 ns and thus helps to sustain the high operand rates required to keep the CAPU working at its maximum capability. The IOAG consists of a 256x64 semiconductor memory which is used to store instructions regarding generation of input and output addresses. The instructions are entered and stored into the IOAG microstore by the PE Channel under the control of the microprocessor. The five Input Stations generate operand addresses in accordance with the instruction to be executed. The Input Address and Output Address Scan Stations pick up the addresses generated by the Input Stations and deposit them appropriately into the FIFOs. Two of the five Input Stations are capable of generating addresses in a "bit-reversed mode"; this mode is frequently used in FFT calculations.

The operation of the Input Station can be explained by means of the diagram shown in Figure 2.2.1.7.2.2-2. The station consists of two 48-bit registers, a multiplexer which selects the contents of one of these registers and an adder/counter unit which is capable of adding a 12-bit number to a 24-bit number and counting up to 4096. A vector is specified by the address of its first element (referred to as the starting address in the following), length which indicates the number of the elements in the vector and increment which when added to the address of the ith vector element gives the address of the (i + 1)th vector element. (The increment can be positive or negative). To initiate the address generation of the components of a vector, it is necessary to load one of the registers of an Input Station in the following manner: the first twenty-four bits should specify the starting address of the vector, the next twelve bits specify the increment and the last twelve bits the length. Once the register is thus loaded, the adder/counter generates the addresses of the vector elements, terminates with the address of the last vector element and indicates that it is done with the address computation.



INPUT/OUTPUT ADDRESS GENERATOR

Figure 2.2.1.7.2.2-1. Input/Output Address Generator

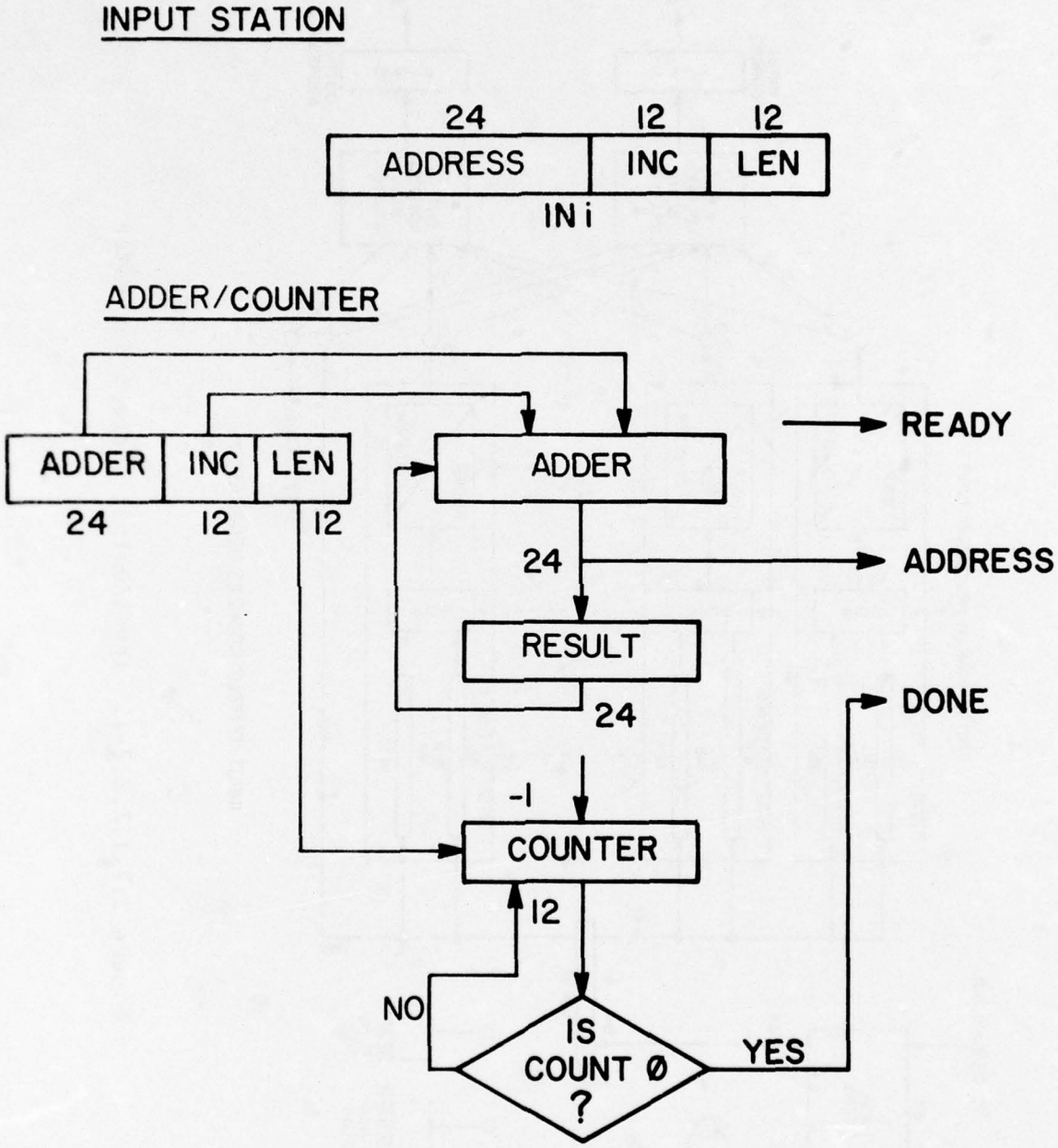


Figure 2.2.1.7.2.2-2
BLOCK DIAGRAM OF IOAG (continued)

A block diagram of the Scan Station is shown in Figure 2.2.1.7.2.2-3. Conceptually it consists of seven four-bit registers and the contents of the registers are shifted end-around four bits at a time. Each four bit register S_i identifies an Input Station. The contents of S_1 identify the Input Station from which the address generated by the station is tapped. Thus by shifting the contents of the registers end-around four bits at a time, it is possible to tap addresses from different Input Stations. The number of stations and the addresses of the stations to be scanned are specified by the programmer.

Instruction formats for the IOAG are shown in Figure 2.2.1.7.2.2-4. All instructions are 64 bits long. In the LOAD instruction the R field specifies the register of an Input Station or a Scan Station. The last 48 bits of the instruction are loaded into this specified register. For an Input Station these 48 bits correspond to the description of a vector to be accessed. For a Scan Station they contain information regarding the Input Stations to be scanned. The LOAD BIT-REVERSE instruction is identical with the LOAD instruction except that the vector specified for access is always assumed to be stored in contiguous locations. The twelve-bit INC field of the vector description which usually specifies the displacement between adjacent vector elements contains information regarding the number of bits to be reversed. From the block diagram of the IOAG it can be seen that it is possible to load an address from the memory directly into the FIFOs through the 6th port.

The ISSUE instruction issues go-ahead signals to the Input Stations and specifies the registers that are effective for their address computations. The 10-bit IN field selects the effective registers. The WAIT instruction halts any further access of instructions until all the active Input Stations are finished with their address computations. COMT is used to communicate with the LSI microprocessor and usually signifies the end of address computation specified by the instructions stored in the VSP Store unit.

An example is given to illustrate the programming aspects of the IOAG. We consider a 32 element decimation-in-time and in-place FFT computation. In the first stage,

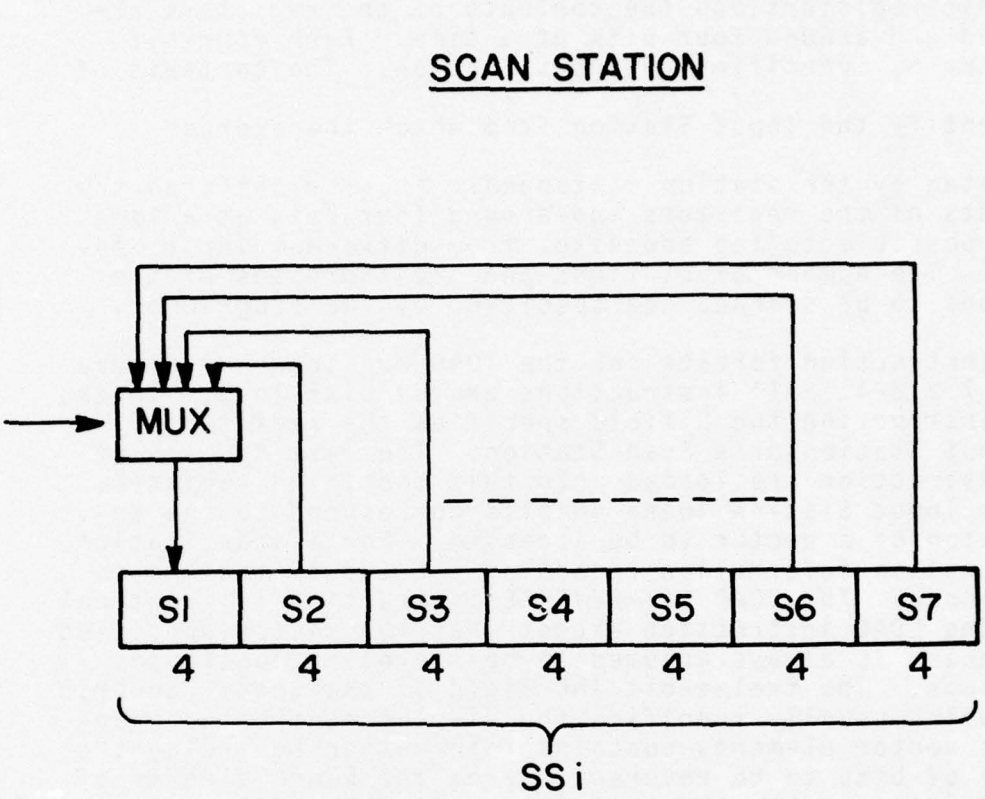


Figure 2.2.1.7.2.2-3

BLOCK DIAGRAM OF IOAG (continued)

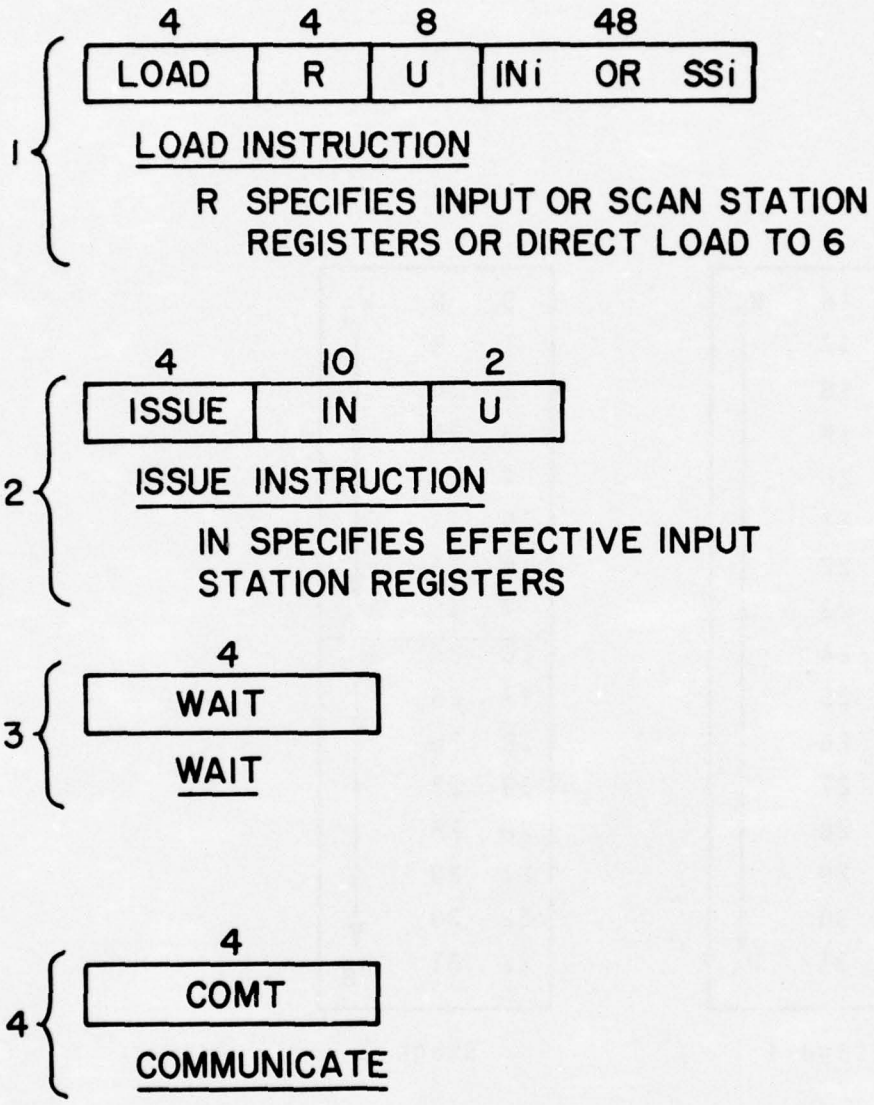
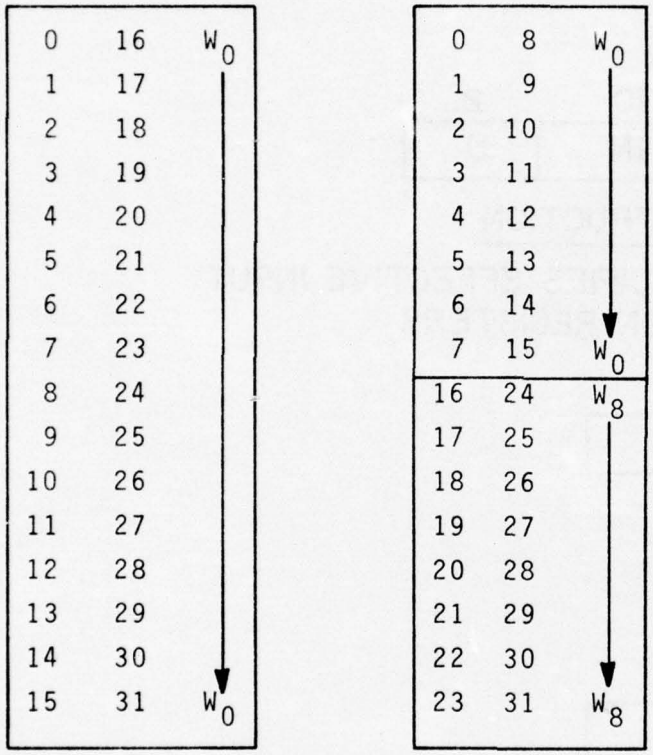


Figure 2.2.1.7.2.2-4
IOAG INSTRUCTION FORMAT



Stage 1 Stage 2

Table 2.2.1.7.2.2-1 - Assignment of twiddle factors, W_i, to operands in each of the five computational stages of a 32-point FFT.

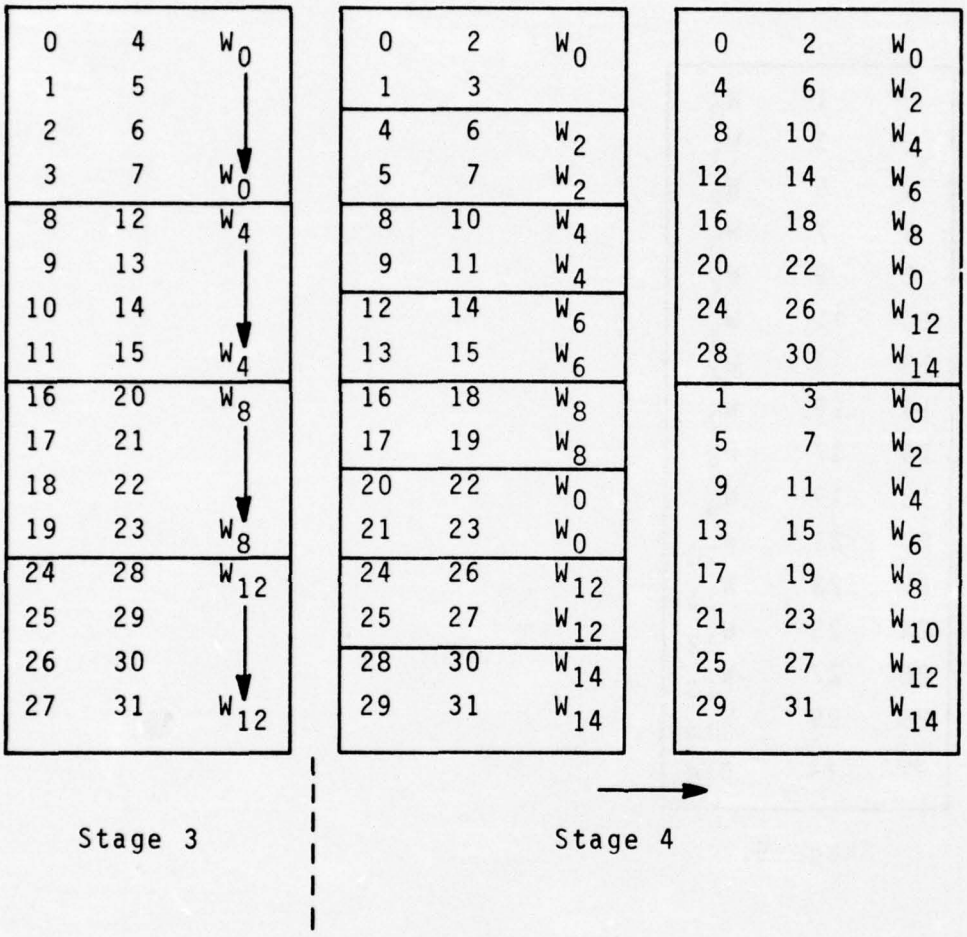


Table 2.2.1.7.2.2-1 (continued)

0	1	W_0
2	3	W_1
4	5	W_2
6	7	W_3
8	9	W_4
10	11	W_5
12	13	W_6
14	15	W_7
16	17	W_8
18	19	W_9
20	21	W_{10}
22	23	W_{11}
24	25	W_{12}
26	27	W_{13}
28	29	W_{14}
30	31	W_{15}

Stage 5

Table 2.2.1.7.2.2-1 (continued)

element 0 is paired with element 16, element 1 with 17 and so on. The twiddle factor is W_0 throughout the first stage of computation. In the second stage elements are paired that are eight apart and there are two twiddle factors W_0 and W_8. For stage 4 we reorder the vector elements as shown in the table so that accessed vectors are relatively long and not fragmented. Similar comments apply to Stage 5.

We use the following notation to indicate the instructions:

```
LOAD INi, SA 1 16
```

Load the input register INi with the starting address SA, INC = 1, LEN = 16.

```
LOADB INi, ........
```

Load the input register INi.....

Addresses are generated with bit reversal

```
ISSUE 1010 ....
```

A ready signal is issued specifying that IN1, IN3, ..., are effective.

```
LOAD SSI, 1 3 2
```

Input Scan Station scans Stations 1, 3 and 2 sequentially for addresses.

With this notation the 32 element FFT program can be written as follows. Here Si and Wi indicate the addresses of the ith element and ith twiddle factor respectively.

```
LOAD  IN1, S0 1 16
LOAD  IN3, S16 1 16
LOAD  IN5, W0 0 16
LOAD  SSI, 1 3 2
LOAD  SS0, 1 2
ISSUE 1010100000
LOAD  IN2, S0 1 8
LOAD  IN4, S8 1 8
WAIT
```

```
ISSUE 0101100000
LOAD  IN1, S16 1 8
LOAD  IN3, S24 1 8
LOAD  IN6, W8  0 8
WAIT
ISSUE 1010010000
LOAD  IN2, S0  1 4
LOAD  IN4, S4  1 4
WAIT
ISSUE 0101100000
LOAD  IN1, S8  1 4
LOAD  IN3, S12 1 4
LOAD  IN6, W4  0 4
WAIT
ISSUE 1010010000
WAIT
LOAD  IN2, S16 1 4
LOAD  IN4, S20 1 4
LOAD  IN5, W8  0 4
WAIT
ISSUE 0101100000
LOAD  IN1, S24 1 4
LOAD  IN3, S28 1 4
LOAD  IN6, W12 0 4
WAIT
ISSUE 1010010000
LOAD  IN2, S0  4 8
LOAD  IN4, S2  4 8
LOAD  IN5, W0  2 8
WAIT
ISSUE 0101100000
LOAD  IN1, S1  4 8
LOAD  IN3, S3  4 8
WAIT
ISSUE 1010100000
LOAD  IN2, S0  2 16
LOAD  IN4, S1  2 16
LOAD  IN6, W0  1 16
LOAD  IN7, S0  1 32
WAIT
LOAD  SSØ, 4
ISSUE 0101011000
COMT
```

2.2.1.7.3 AP FIFO's

The AP operates in a pipelined fashion to obtain extremely high throughputs, particularly for array-type operations. The different parts of this pipeline are buffered by FIRST-IN-FIRST-OUT memories (FIFOs) that help to smooth out the effects of non-uniform delays in different parts of the pipeline.

There are 4 such FIFOs in the AP:

(i) Input Address FIFO (IAF)
(ii) Input Data FIFO (IDF)
(iii) Output Address FIFO (OAF)
(iv) Output Data FIFO (ODF)

The two address FIFO's (IAF and OAF) are 26 bits wide - 24 bits for address and 2 bits for the access mode (8, 16, 32 or 64 bits). The two data FIFOs (IDF and ODF) are 64 bits wide. The lengths of the FIFOs are in multiples of 8 words. A length of 8 words is assumed in the FIFO design described here. The FIFOs are implemented using dual-ported parallel-access TTL RAMs and associated control logic.

2.2.1.7.3.1 FIFO Operation

A block diagram of the FIFO is shown in Figure 2.2.1.7.3.1-1. The Input Control and Output Control provide the interfaces to the FIFO.

During an input cycle, the requesting device puts data on the Input data lines and raises IREQ. The FIFO control initiates a write cycle if FIFO FULL is false, sets Write Busy and writes the device data into the RAM at the address given by WCTR. At the end of the write cycle (45 nanoseconds) WCTR is incremented, setting the RAM address for the next data location. It then waits to allow for the delays of the WCTR and ALU and sends a PL signal to the external device to indicate the end of the write cycle. A typical write cycle takes 87 nanoseconds.

An output cycle starts when FIFO-EMPTY is false; the output control sets the output request FF and initiates a RAM read operation. After a delay to account for delays of the Read Counter (RCTR) and the ALU, the RDY FF is set. The external device can load the FIFO output data whenever RDY is true; it sends a RST signal back to the AFO to reinitiate a new cycle.

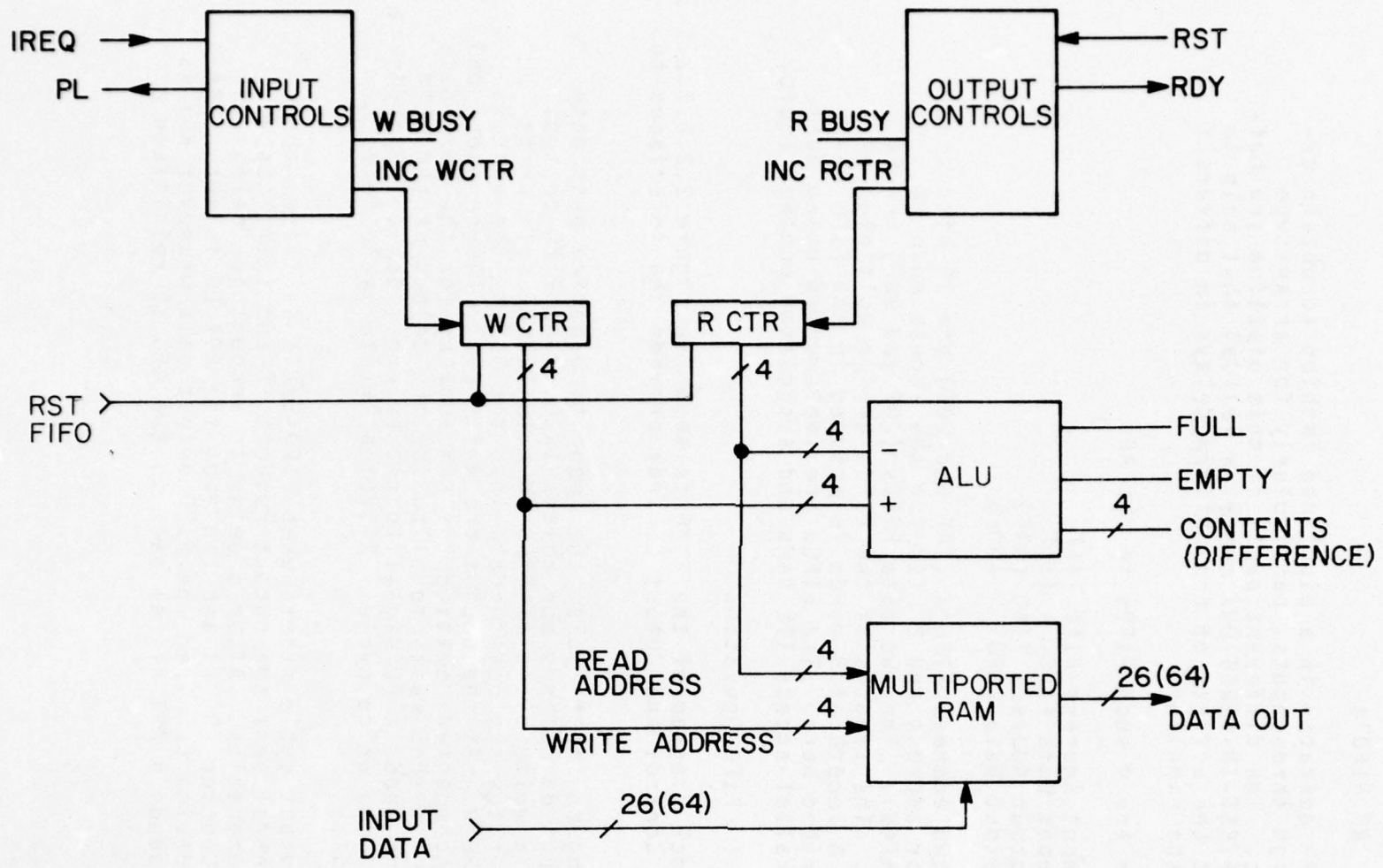


FIGURE 2.2.1.7.3.1-1 FIFO BLOCK DIAGRAM
4047-760227-15 A

The following status signals are available to the device at the input and output of the FIFO and to the AP controller (Section 2.1.7.6).

(i) FULL. This is set when the difference between the WCTR and RCTR is -1. In the case of the IDF only, the Full bit is set when there is still room for two words to account for data in the AP bus pipeline.

(ii) EMPTY. This is set when WCTR = RCTR.

(iii) CONTENTS. The number of words in the FIFO is given by the difference between the WCTR and RCTR, available at the output of the ALU. It is available to the microprocessor through the AP Controller. It is useful for obtaining statistics of FIFO usage.

2.2.1.7.4 Data Fetch Unit and Data Store Unit

The Data Fetch Unit (DFU) and Data Store Unit (DSU) handle the input-output operations for the Central Arithmetic Processing Unit (CAPU) of the AP. They represent two stages in the AP pipeline shown in Figure 2.2.1.7-1 and operate concurrently with other modules within the AP. Figure 2.2.1.7.4-1 contains a block diagram of the AP-to-memory interface.

The DFU receives input addresses from the Input Data FIFO or directly from the CAPU, requests memory read cycles through the AP Bus Control and sends the memory data to the Input Data FIFO or directly to the CAPU. The DSU receives output addresses from the Output Address FIFO and output data from the Output Data FIFO. It requests memory write cycles through the AP Bus Control and writes the data in memory.

2.2.1.7.4.1 DFU Operation

The DFU is shown in Figure 2.2.1.7.4-1. The DFU starts its cycle when the Input Address FIFO has available addresses as indicated by its Ready line, and the Input Data FIFO has room for data as indicated by its FULL line being low, provided the DFU is not busy with a previous cycle. The DFU latches the address into the DFU Address Buffer and sends a request for a memory cycle, AREQ3, to the arbiter within the AP Bus Control. The FULL line of the Input Data FIFO actually goes True when the FIFO still has room for two more words to account for the two words that can already be "in the pipeline" in the DFU and the AP Bus Control. The CAPU also has a direct path into the DFU; when it requests direct data, a request is also sent to the AP Bus Control Arbiter by asserting AREQ1.

The AP Bus Control arbitrates the two requests for the AP Bus from the DFU and requests from the DSU; and executes the memory cycle. After the AP Bus Control reads in the address, it sends back a signal (RST3 or RST1 respectively) so that the DFU can proceed with the next cycle. When the AP Bus Control completes a requested memory cycle, it returns the data (64 data bits and 8 parity bits) as well as mode information and the destination of the data: the Input Data FIFO or direct CAPU input. The DFU routes the data through the Read part of the AP Multi-mode Shifter, so that the required part of the data word is shifted right-justified. See Section 2.2.1.7.7 for the operation of the Multi-mode Shifter. The data is then deposited either in the Input Data FIFO or the CAPU read buffer to be read in directly by the CAPU.

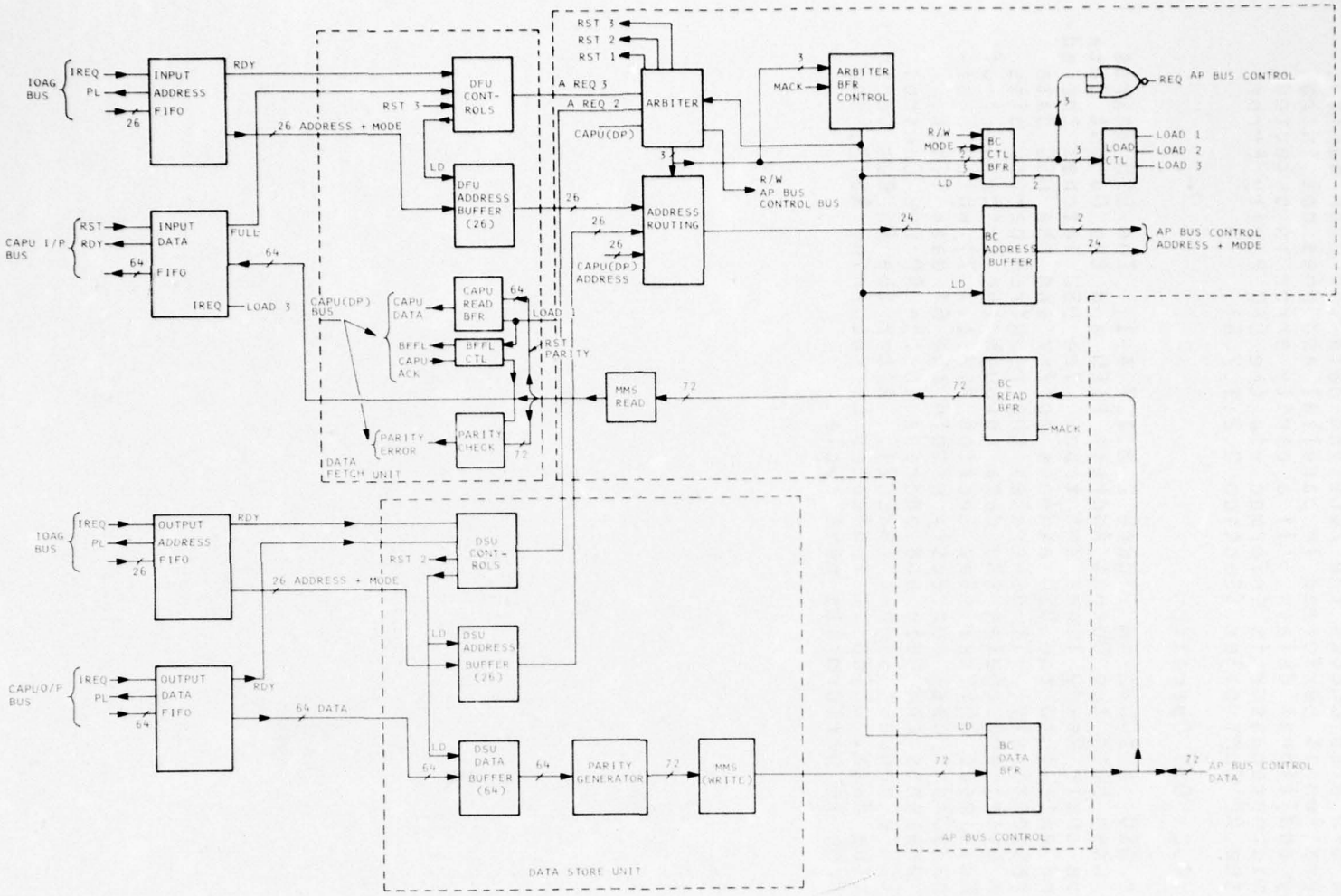


Figure 2.2.1.7.4-1. Block Diagram of AP Bus to Memory Interface

The DFU also checks the received data for even parity. This operation is performed in parallel and does not introduce any additional delays. If a parity error is detected, the PE microprocessor is informed via the DFU Parity Error bit of the AP Controller (Section 2.2.1.7.6).

2.2.1.7.4.2 DSU Operation

The DSU is shown in Figure 2.2.1.7.4-1. The DSU starts a cycle when both the Output Address FIFO and the Output Data FIFO have their Ready lines set true. The DSU latches the address and data into the DSU address buffer and the DSU data buffer respectively. It generates the required parity bits (one per byte) and routes the data through the write part of the AP Multimode Shifter (See Section 2.2.1.7.7) which positions the actual data correctly within the 64 data lines. It then presents the data and address to the AP Bus Control along with a Request signal (AREQ2). After the AP Bus Control accepts the data, which it indicates by returning RST2, the DSU is free to perform its next cycle.

2.2.1.7.5 AP Bus Control

The AP Bus Control handles the memory transactions for the Data Fetch Unit and the Data Store Unit. It receives requests for memory cycles over three paths: read cycle requests from the DFU for the Input Data FIFO or the CAPU direct input or write cycle requests from the DSU. In the event of simultaneous requests, one is selected by an arbitration circuit and a memory cycle is requested over the AP bus, which provides a path to each of the PELS banks and to the CWS through the PE Bus Control and the DREA. See Figure 2.2.1.7.4-1 for details.

2.2.1.7.5.1 Operation of AP Bus Control

The AP Bus Control Arbiter receives requests for AP Bus cycles over REQ1, REQ2, and REQ3. When a request comes on one of these lines, all three are latched into three priority flip-flops and in the event of simultaneous requests arbitration logic ensures that only one priority flip-flop is set. The assignment of priorities is as follows:

(i) Direct CAPU input (highest priority)
(ii) Data Store Unit
(iii) Input Address FIFO, via DSU

The AP Bus Control arbiter ignores subsequent requests until the cycle is completed. The three requesting sources also provide 24-bit addresses and 2-bit mode information. In addition, the DSU also provides 72 bits of data and parity to be written out. Once the arbitration is done and the priority flip-flops are set, they are used to get the address and mode information from the selected source. After a delay to ensure that the multiplexed data has settled it is loaded into the Bus Control Buffer, provided the buffer is not being used for the previous cycle. This is indicated by the Buffer Free flip-flop (BFFF). The contents of the priority flip-flops is also loaded into the BC CTL Buffer to be used as the ID of the source when the data is received from the memory. Once the BC Buffer has been loaded, the arbiter is free to repeat its cycle and look at the request lines. The selected device is also sent a signal (RST1, RST2 or RST3) so that it can make another request. At the same time, the Buffer Free flip-flop is cleared and a request is sent over the AP bus for a memory cycle along with the address, control information, and data (in the case of a DSU cycle) from the Bus Control Buffer. When the memory cycle is completed and MACK is received over the bus, the BFFF is set so that the next cycle can begin. In the event of a read cycle, the received data is loaded into the BC Read Buffer along with the relevant

information such as mode, and the ID bits. The data in the Read buffer is sent to the Input Data FIFO or the CAPU read buffer according to the setting of the ID bits.

2.2.1.7.6 AP Controller

The AP Controller is the interface between the PE microprocessor and the Arithmetic Processor (AP) which consists of the CAPU, IOAG, and the four FIFOs. It consists of a set of registers on the microprocessor bus and the logic necessary to control the AP functions and to provide status indications to the microprocessor.

Using these registers, the microprocessor can start programs, enable interrupts, and check for error conditions. It can also operate the CAPU and IOAG in single-step mode for diagnostic purposes.

The interfaces with the CAPU, IOAG, and the FIFOs are described separately in the following sections.

2.2.1.7.6.1 CAPU Control

The AP Controller contains four Control and Status Registers for the CAPU. These registers are illustrated in Figure 2.2.1.7.6.1-1.

The CAPU Control Register contains only four bits. The RUN bit (bit Ø) controls the CAPU clock. When the RUN bit is set to 1, the CAPU clock starts and continues to run as long as the RUN bit remains 1 and no other halt conditions are present. The RUN bit must be set to Ø to enable programs to be loaded into the CAPU memory by the Channel.

The CAPU can be made to halt on error conditions by setting bit 3 of the control word to 1. Note that this is independent of the interrupt on error function, which is controlled by the Mask Register. If the CAPU halts because of an error condition, it resets the RUN bit to Ø. The microprocessor can then determine the type of error from the Status Register, take any appropriate action, and restart the CAPU using the RUN bit.

For diagnostic purposes, the microprocessor can operate the CAPU in single-step mode. To do this, the RUN bit must be set to Ø. Then the CAPU will execute a single instruction each time bit 2 is set to 1.

Bit 1, the Direct Mode bit, is used to give the microprocessor direct access to the CAPU inputs and outputs. Setting bit 1 to 1 causes the CAPU inputs and outputs to be removed from the FIFOs and switched to a set of registers directly accessible to the microprocessor (see Figure 2.2.1.7.6.1-2). Direct Mode is useful for diagnostics, and may also be used for

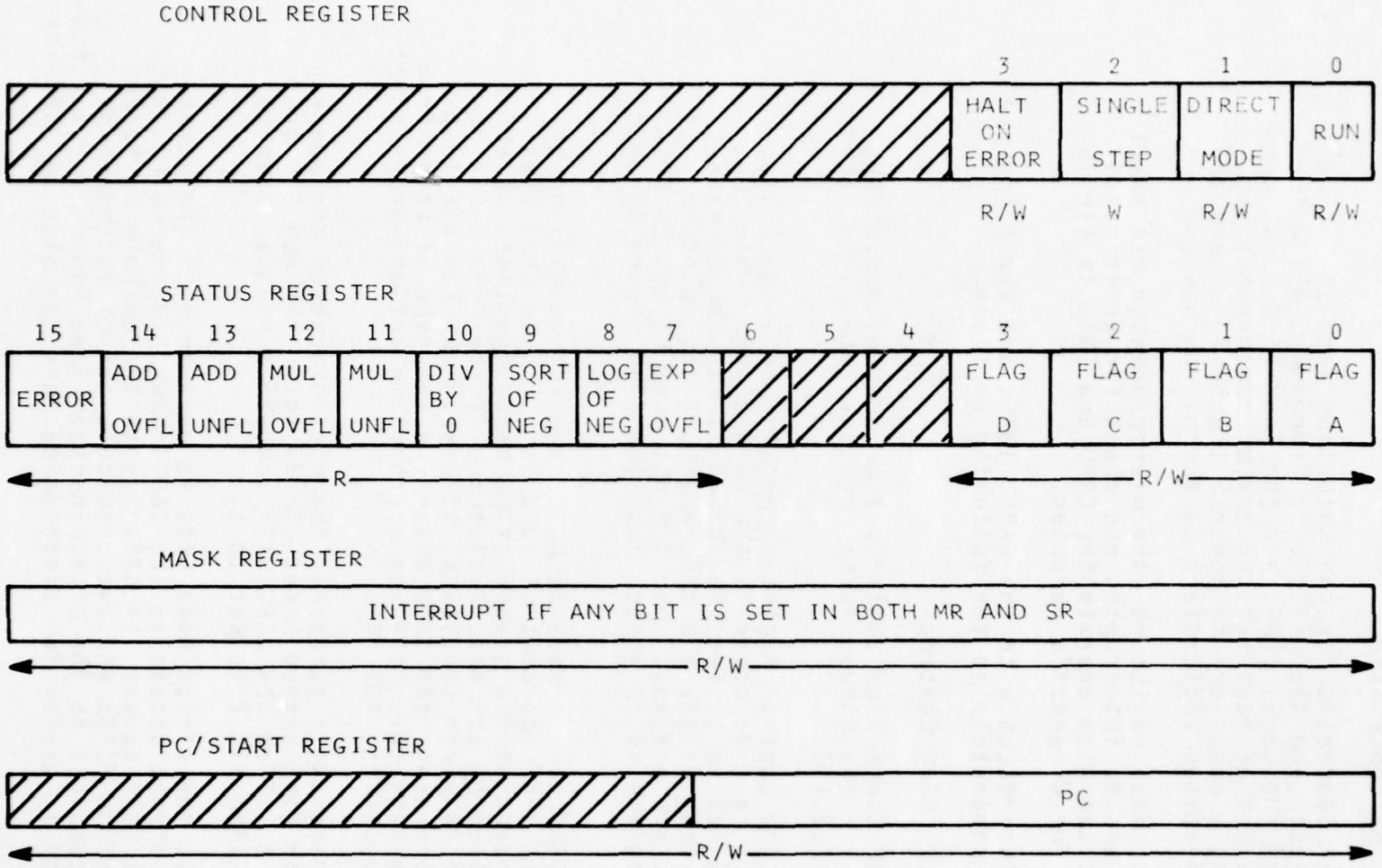


Figure 2.2.1.7.6.1-1. CAPU Control and Status Registers

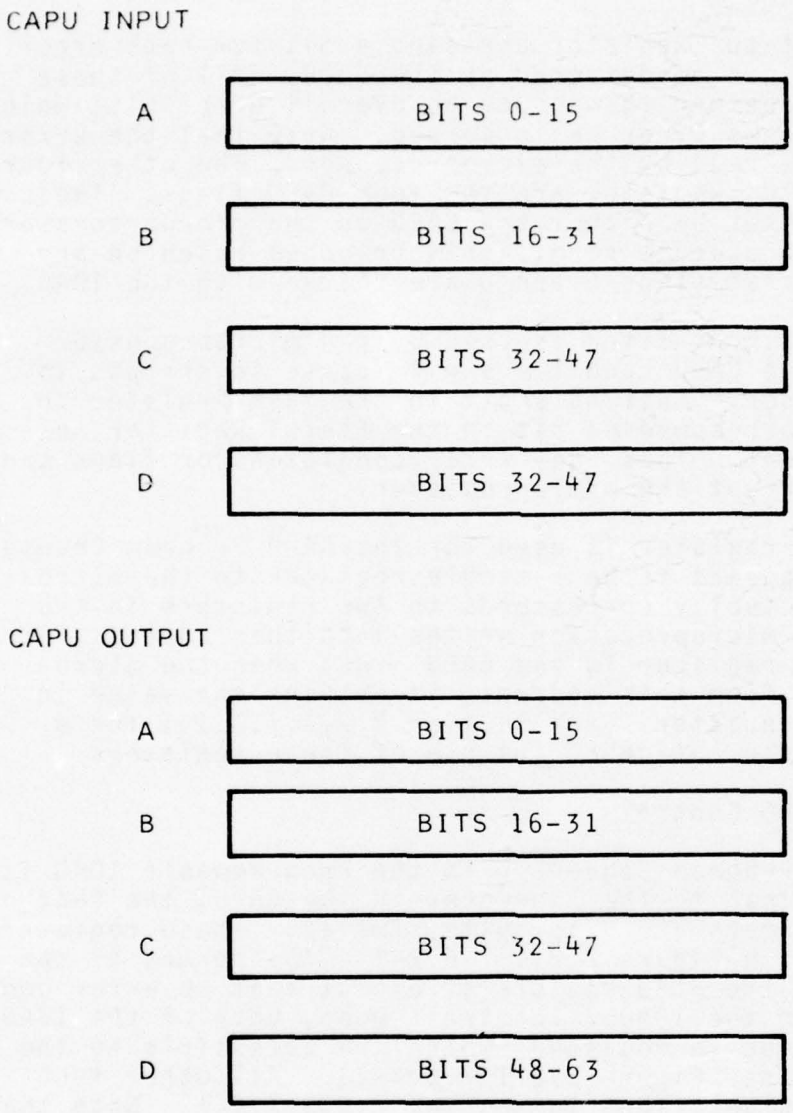


Figure 2.2.1.7.6.1-2. CAPU Direct Mode Registers

scalar floating point operations.

The CAPU Status Register contains a bit for each error condition which can be detected by the CAPU. All of these bits are ORed together to produce an overall error bit, which indicates that some error has occurred. Note that the error bits can only be read by the microprocessor. The other four bits in the Status Register are the four CAPU flags. These can be set or reset by either the CAPU or the microprocessor, and the CAPU can execute conditional branches based on any of them. Note that flags C and D are shared with the IOAG.

The CAPU Mask Register is used by the microprocessor to determine what CAPU conditions will cause interrupts to the microprocessor. Setting a bit in the Mask Register to 1 enables the corresponding bit in the Status Register to cause an interrupt. Thus, any error conditions or flags can be used to interrupt the microprocessor.

The fourth register is used for the CAPU Program Counter. Although this appears to be a single register to the microprocessor, it actually corresponds to two registers in the CAPU. When the microprocessor writes into this address, it loads the START register in the CAPU. But when the microprocessor reads from this address, it obtains the value in the CURRENT PC register. See section 2.2.1.7.1.1.1 for a more detailed description of the use of these registers.

2.2.1.7.6.2 IOAG Control

Since the Program Sequencer in the Programmable IOAG is virtually identical to the Sequencer in the CAPU, the Control and Status Registers are also quite similar. These registers are illustrated in Figure 2.2.1.7.6.2-1. The format is the same as that of the CAPU registers, except that no error conditions exist in the IOAG. In direct mode, both of the IOAG outputs are placed in registers which are accessible to the microprocessor (see Figure 2.2.1.7.6.2-2). All other functions operate as described in section 2.2.1.7.6.1. Note that two of the flags, C and D, are shared with the CAPU.

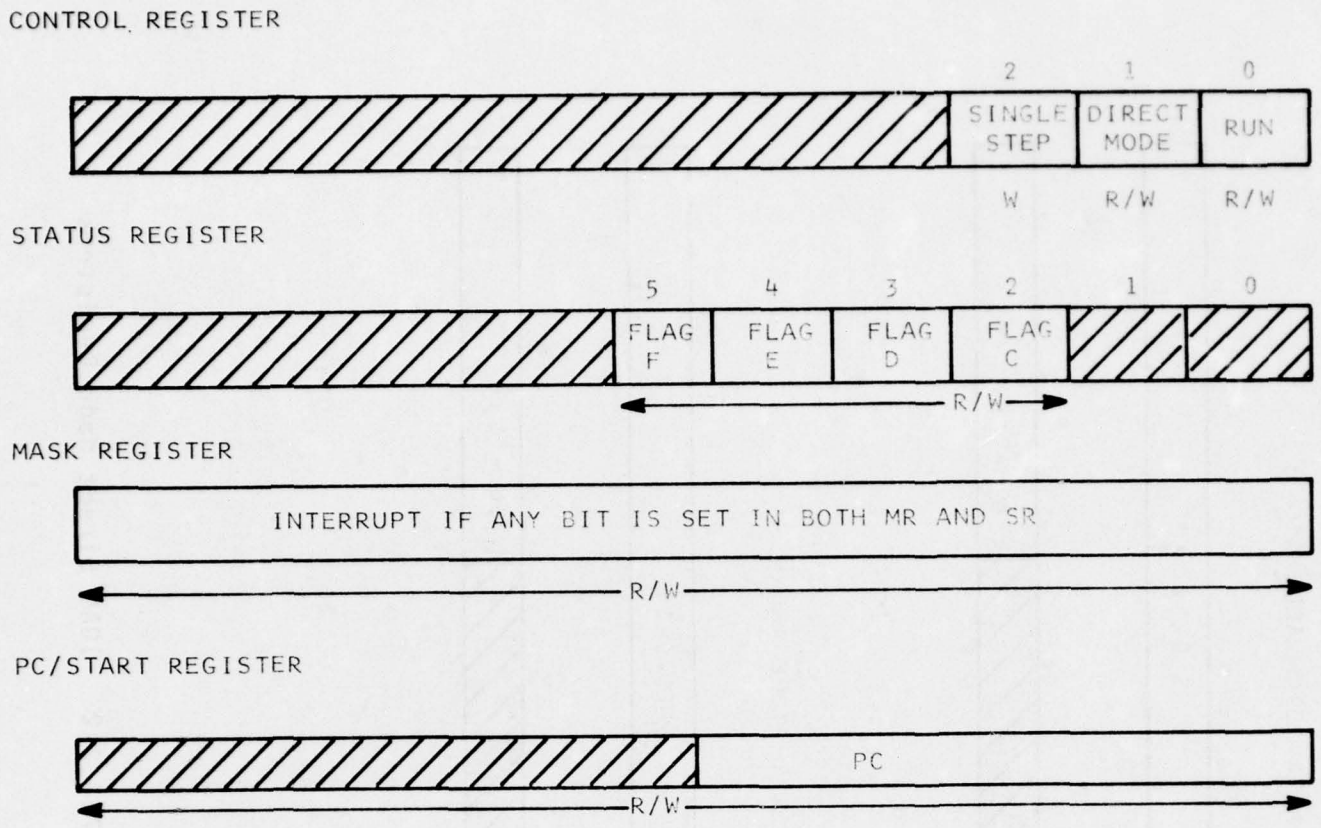


Figure 2.2.1.7.6.2-1. IOAG Control and Status Registers

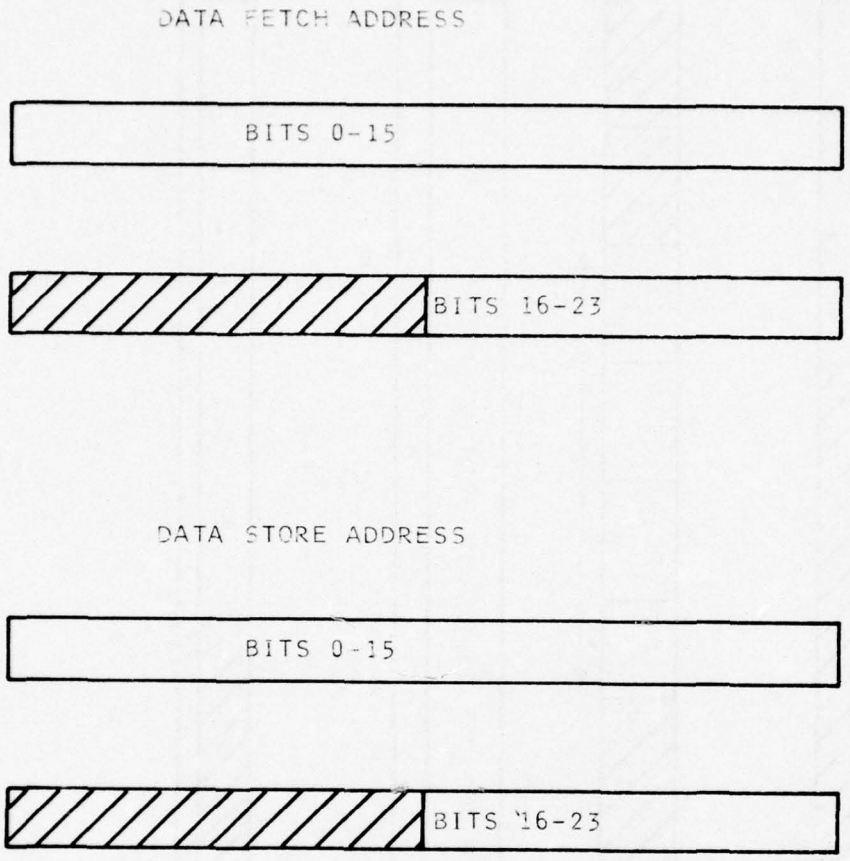


Figure 2.2.1.7.6.2-2. IOAG Direct Mode Registers

2.2.1.7.6.3 FIFO Status

Five registers are provided to enable the microprocessor to control and monitor the four AP FIFOs and the Data Fetch Unit/Data Store Unit. These registers are illustrated in Figure 2.2.1.7.6.3-1.

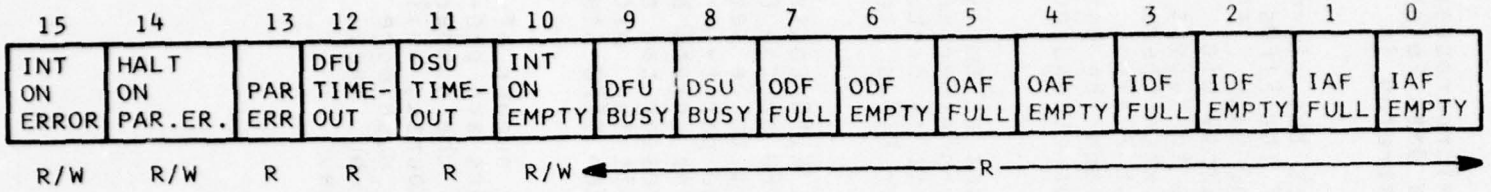
The FIFO Control and Status Register enables the microprocessor to determine the status of all four of the FIFOs and the Data Fetch Unit and Data Store Unit. For each FIFO, the Status Register contains a Full bit and an Empty bit. The Full bit will be 1 when the FIFO is not able to accept any additional inputs. The Empty bit will be 1 when there is no data stored in the FIFO. The Status Register also contains Busy bits for both the Data Fetch Unit and the Data Store Unit. Each of these bits is 1 when the corresponding unit is waiting for a response from a memory.

Bit 10 of the FIFO Status Register, Interrupt on Empty, is used to enable the AP to interrupt the microprocessor when it has completed all operations in the programs it is executing. The interrupt will take place when all four FIFOs are empty and both the DFU and the DSU are not busy.

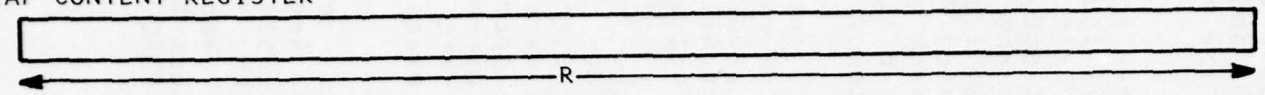
Three error conditions are indicated in the FIFO Control and Status Register. A time-out by either the DFU or the DSU will cause the AP to halt and the appropriate bit to be set. A parity error detected by the DFU will cause the parity error bit to be set, but the AP will continue operations unless bit 14, Halt on Parity Error, is set. The microprocessor can enable interrupts on error conditions by setting bit 15. After the interrupt, the microprocessor must examine the FIFO Status Register to determine the type of error.

The microprocessor can determine the actual number of words stored in each FIFO. Four Content Registers are provided for this purpose. Each register indicates the number of words stored in the corresponding FIFO. Of course, while the AP is running, these numbers may be changing much more rapidly than the microprocessor can access them, but they can be sampled to obtain statistics on FIFO usage.

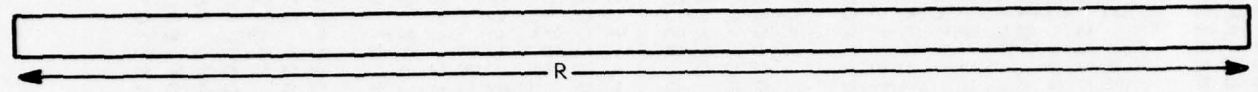
FIFO CONTROL & STATUS REGISTER



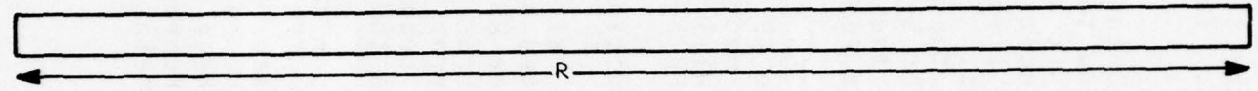
IAF CONTENT REGISTER



IDF CONTENT REGISTER



OAF CONTENT REGISTER



ODF CONTENT REGISTER

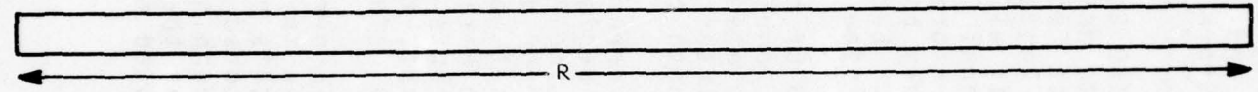


Figure 2.2.1.7.6.3-1. FIFO Registers

2.2.1.7.7 Multimode Shifter

The memories in the G-471 system, the CWS and the PELS in particular, are addressed by byte (8 bits). However, they are organized so that up to 8 bytes (64 bits) can be accessed in parallel (parity bits are ignored in this description).

The major system buses (PE bus, AP bus, channel bus and extended microprocessor bus) provide 64 bit wide data paths, but the devices in command of these buses can communicate with memory over them in units of 8, 16, 32 or 64 bits. Two mode bits are included to specify the path-width as follows:

Mode	Path-Width (bits)
00	64 (double word)
01	32 (word)
10	16 (half-word)
11	8 (byte)

At the device end of the bus, the data is right-justified. The function of the multimode shifter is to shift the data to the correct position. During a read cycle, the memory places the whole 64-bit word on the bus and the multimode shifter shifts the part actually specified to the right end of the word. During a write cycle, the device presents the data right-justified and the multi-mode shifter shifts it left to the correct part of the 64-bit data word. The memory writes only the bytes to be modified. The Multimode Shifter determines the direction and amount of shift and the bytes to be shifted from the read/write bit, the mode bits and the three low-order address bits. This is illustrated in the tables below, which give the amount of shift in bytes and the direction (L = left, R = right) for the different cases.

Read Cycle

			Mode	ØØ	Ø1	1Ø	11
A3	A2	A1		(64)	(32)	(16)	(8)
0	0	0		0	0	0	0
0	0	1		0	0	0	1R
0	1	0		0	0	2R	2R
0	1	1		0	0	2R	3R
1	0	0		0	4R	4R	4R
1	0	1		0	4R	4R	5R
1	1	0		0	4R	6R	6R
1	1	1		0	4R	6R	7R

Write Cycle

A3	A2	A1	Mode	00 (64)	01 (32)	10 (16)	11 (8)
0	0	0		0	0	0	0
0	0	1		0	0	0	1L
0	1	0		0	0	2L	2L
0	1	1		0	0	2L	3L
1	0	0		0	4L	4L	4L
1	0	1		0	4L	4L	5L
1	0	0		0	4L	6L	6L
1	1	1		0	4L	6L	7L

The shifting function is implemented using multiplexers. For each byte position, all the bytes that may have to be shifted to it are multiplexed. The worst case delays introduced are 9 and 23 nanoseconds for the read and write parts respectively.

The following tables show the multiplexing for the various cases. Each row corresponds to a byte position after the shifting and the entries show the bytes from which the data comes for the different modes:

Read

Byte	Mode	0 (64 bits)	1 (32 bits)	2 (16 bits)	3 (8 bits)
0		0	0,4	0,2,4,6	0,1,2,3,4,5,6,7
1		1	1,5	1,3,5,7	-
2		2	2,6	-	-
3		3	3,7	-	-
4		4	-	-	-
5		5	-	-	-
6		6	-	-	-
7		7	-	-	-

Write

0	0	0	0	0
1	1	1	1	0
2	2	2	0	0
3	3	3	1	0
4	4	0	0	0
5	5	1	1	0
6	6	2	0	0
7	7	3	1	0

The Multimode Shifter described here is located within the AP Bus Control. An identical shifter is utilized in the PE Channel.

2.3 The Central Working Storage (CWS)

The CWS is a system wide storage facility with a capacity of up to 16 Mbytes. It can be accessed directly by all the PEs, the Control Computer, the mass storage units and real time I/O channels. It is used to hold the data being processed by the system processors as well as for communication between them. It is also used for buffering by the Mass Storage Units and the real-time I/O channels.

The CWS is divided into up to 32 banks that can be accessed in parallel, through the Data Routing Element Array (DREA) by any of up to 32 devices.

The CWS is organized to access up to 64 bits in parallel, but is byte-addressed. It can be accessed in units of 8, 16, 32 or 64 bits as specified by two mode bits (MB1 and MB2) as follows:

MB1	MB2	Mode
0	0	0 double word - 64 bits (+8 parity bits)
0	1	1 word - 32 bits (+4 parity bits)
1	0	2 half-word - 16 bits (+2 parity bits)
1	1	3 byte - 8 bits (+1 parity bit)

Within a double-word, word or half-word, the bytes are numbered right to left, and the address is the same as the address of the right most byte. Note that this convention is the same as in the PDP-11 but the opposite of the IBM 360 and 370. The CWS addressing scheme is illustrated in Figure 2.3-1.

The CWS has a cycle time of 395 nsec. Read and write cycles, however, are overlapped with the DREA delay by sending MACK before cycle is finished. In the case of read, MACK is set when the data is available (279 nsec.). While in the case of write operation, MACK is set as soon as data and address are latched in the memory module (49 nsec). The DREA requests read or write cycles by setting Read Initiate L or Write Initiate L respectively.

2.3.1 Organization.

The CWS is divided into up to 32 independent banks. Each bank has up to 576 kbytes (including parity bits), which is further subdivided into 16 modules. Total capacity of each module is 36 kbytes which is organized as 4k words of 72 bits.

2.3.2 Layout of CWS Bank

Each bank consists of a memory control module, and sixteen memory modules. One module is activated at a time. Each

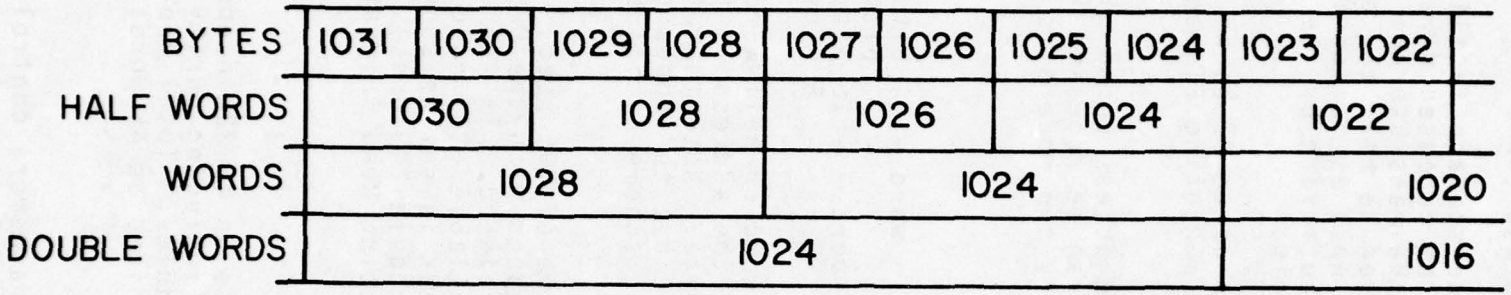


FIGURE 2.3-1 CWS ADDRESSING

module is further arranged in 8 groups, each containing 4k x 9 bits, called memory stack. Thus each bank has 128 stacks, which are realized by sixteen memory modules. The CWS layout is shown in Figure 2.3.2-1.

2.3.3 CWS Operation

The CWS operation can be best explained by describing the CWS read and write cycles.

Read Cycle

The DREA sends address and mode information to the control module. Then it activates the Read Initiate L signal. The control module sees this signal and latches address and mode information when the current cycle is over. It then waits for the following to happen before it starts the memory cycle:

1. Mode bits 0 and 1 are decoded to establish mode of access.

2. Group select Ø through 7 are activated depending upon the mode of access. This decides which group(s) are selected for read operation. The following table describes the relation between group number and corresponding bits:

Group Ø	Bits Ø - 7
1	9 -17
2	18 -26
3	27 -35
4	36 -44
5	45 -53
6	54 -62
7	63 -71

3. Address bits A15 through A18 select one of sixteen memory modules. This decode is performed on each card. Jumper provisions are made to assign each memory module any address 00 through 15.

The Cycle FF initiates a memory cycle. At the end of the access time (225 nsec), data is latched into the read buffer. Selected group buffers present data on the bus to the DREA. Unselected group outputs are put into the high impedance state. The output remains on the bus until the Read Initiate L is active.

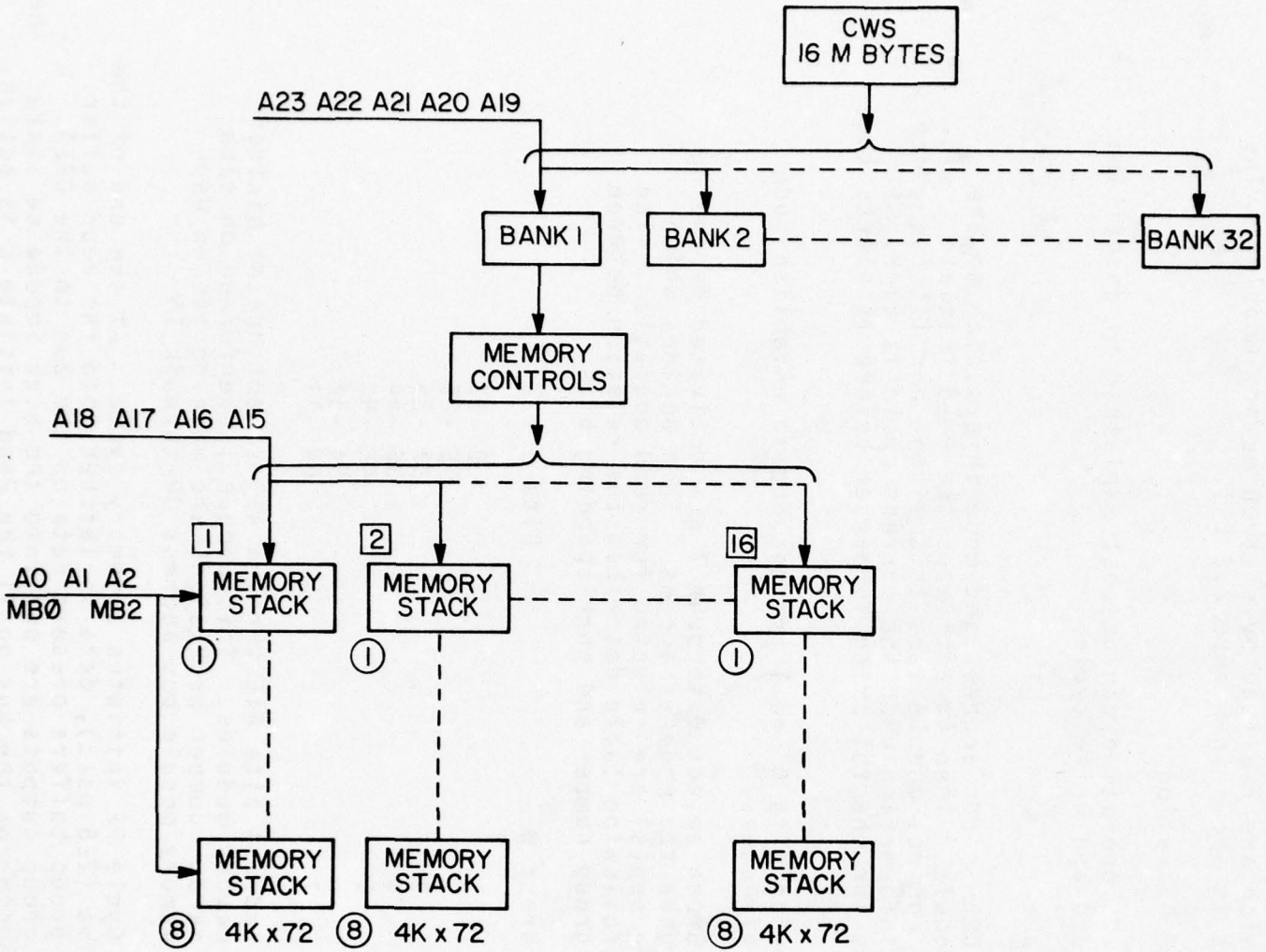


FIGURE 2.3.2-1 CWS BLOCK DIAGRAM
4047-760227-06A

Write Cycle:

The Write cycle is similar to Read cycle with the following exceptions:

1. Write Initiate L also latches the write data presented by DREA.

2. MACK is issued as soon as data and address are loaded and settled down (49 nsec).

Notice that Read and Write operations are performed only on the selected group or groups. Any unselected group and remaining subtanks remain unchanged.

2.3.4 DREA - CWS Interface Signals

1. Read Initiate DREA → CWS

2. Write Initiate DREA → CWS

3. Acknowledge CWS → DREA

Read Cycle

The DREA requests a Read operation by presenting the address to the CWS and lowering Read Initiate. CWS waits until any previous operation is finished, and then accepts the address and begins a read cycle. When the data is available, CWS sends the acknowledge signal to the DREA. Data should remain available on the CWS output lines until the Read Initiate ends.

Write Cycle

The DREA requests a Write operation by presenting the address and data to the CWS and lowering Write Initate. CWS waits until any previous operation is finished, and then accepts the address and data. When CWS has latched in the address and data, it sends the acknowledge signal to the DREA.

2.4 Data Routing Element Array

The purpose of the Data Routing Element Array (DREA) is to provide multiple access to all of the memory blocks used in the Central Working Storage (CWS). The array is designed to permit any of 32 devices (PEs, disk controllers, etc.) to access any of 32 memory banks, but the number of memory banks can easily be increased to 64. Increasing the number of access devices to 64 would require an extension to the arbiters, which would increase the delay in the system, but no other difficulties would result from such an increase. The array may be any size up to 64 by 64 provided that each dimension is a multiple of 4.

As shown in Figure 2.4.0-1, the Data Routing Element Array consists of three major sections: decoders, arbiters, and the switch array. Associated with each access device is an address decoder, which compares the logical address received from the device with all of the memory bank addresses assigned by the control computer. If the address is found, a REQUEST signal is sent to the arbiter for that memory bank.

Associated with each memory bank is an arbiter, which determines which device is to be granted access to the memory in the event that more than one device requests access at a time. The output from an arbiter is a level on one of its SELECT lines to one point in the crossbar switch.

The actual connection of a memory bank to the device granted access takes place in an array of crossbar switch modules. Each module can connect 4 devices to 4 memory banks. The address and control lines are unidirectional, and the data lines bidirectional, with the direction being chosen by the READ or WRITE line from the device.

More detailed descriptions of the operation of the Data Routing Element Array are given in the following sections, together with information on the number of parts required and the delay introduced by the system.

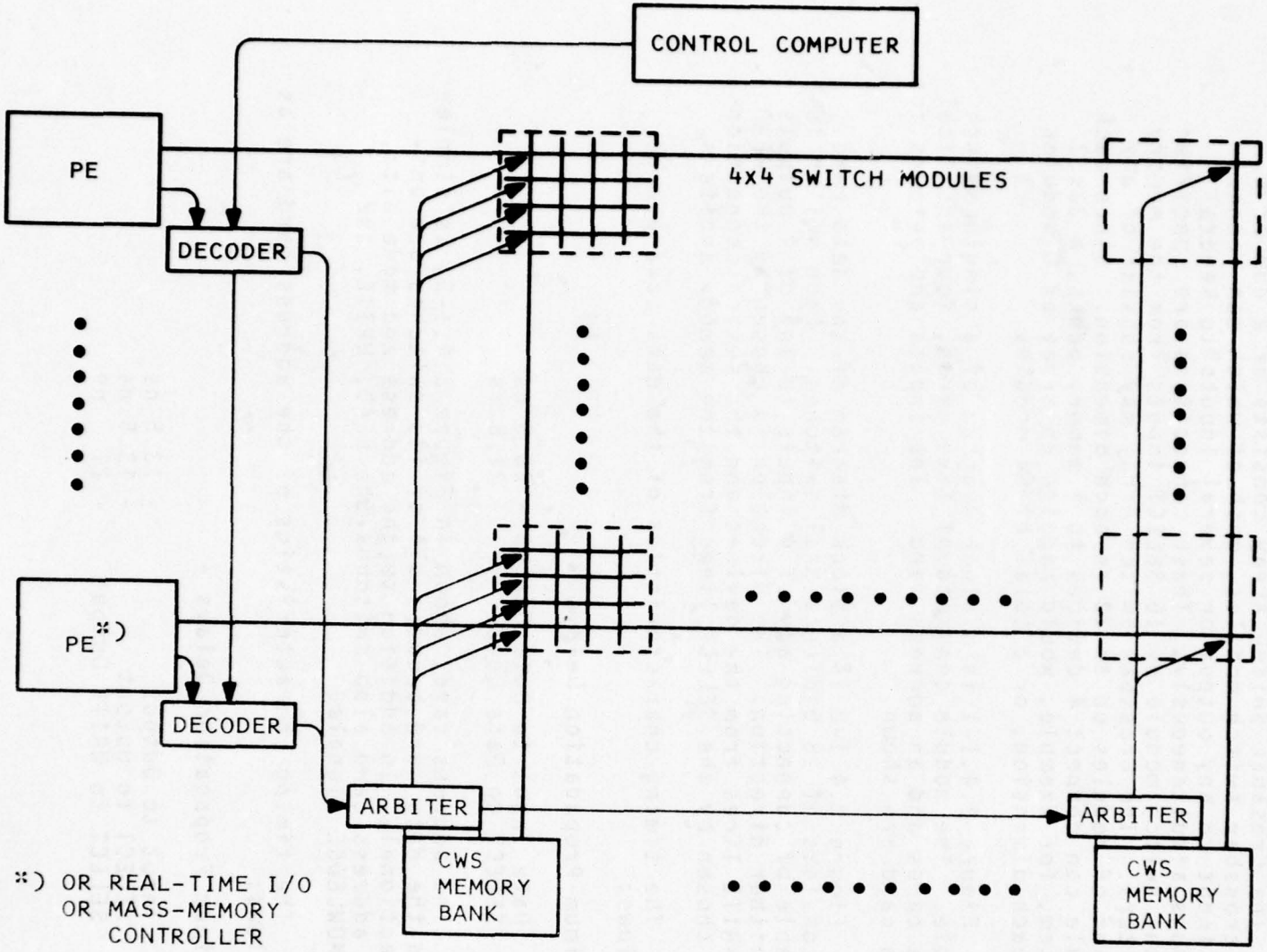


Figure 2.4.0-1. Block Diagram of Data-Routing Element Array

2.4.1 Crossbar Switch Array

The Crossbar Switch Array consists of a number of 4x4 crossbar switch modules, each of which can connect any input to any output or several inputs to several outputs simultaneously. These connections are specified to the switch module by 16 SELECT inputs from the memory arbiters. The Crossbar Switch Array may consist of any number of modules up to 20 in each dimension. Since each module can connect 4 devices to 4 memory banks, a 32x32 system, for example, would require an array of 8 modules in each dimension, or a total of 64 modules.

Figure 2.4.1-1 is a block diagram of a single switch module. The module consists of five cards, four identical data cards and an address card. The inputs and outputs for each card are shown.

Figure 2.4.1-2 is a block diagram of the data card. It consists of 18 bidirectional switches. Each switch is capable of connecting any of 4 inputs to any of 4 outputs in either direction. The direction is chosen by the READ or WRITE lines from the devices and the switch connections are chosen by the SELECT lines from the memory arbiters.

The timing characteristics of the data card are as follows:

Maximum Propagation Delays -

Data Input to Data Output	- 28.5 ns
SELECT to Data Output	- 21.5 ns

The address card, shown in Figure 2.4.1-3, is simpler than the data card because all of its switches are unidirectional. In addition to the address and mode bits, the address card also switches the READ, WRITE, and ACKNOWLEDGE signals.

The timing characteristics of the address card are as follows:

Maximum Propagation Delays -

Input to Output	- 13.5 ns
SELECT to Output	- 13.5 ns
SELECT to WRITE Output	- 21 ns

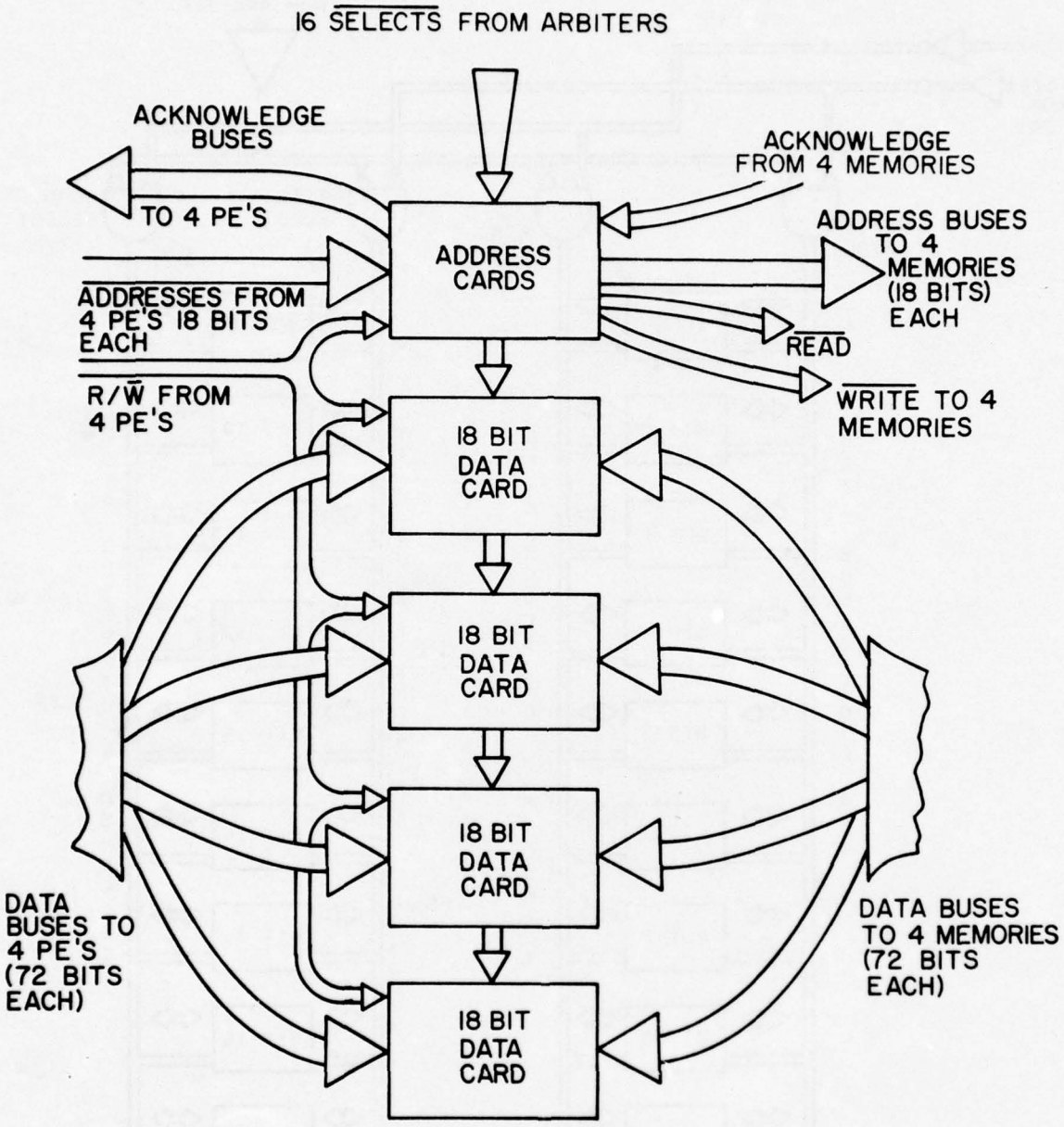


FIGURE 2.4.1-1 DREA 4x4 SWITCH MODULE

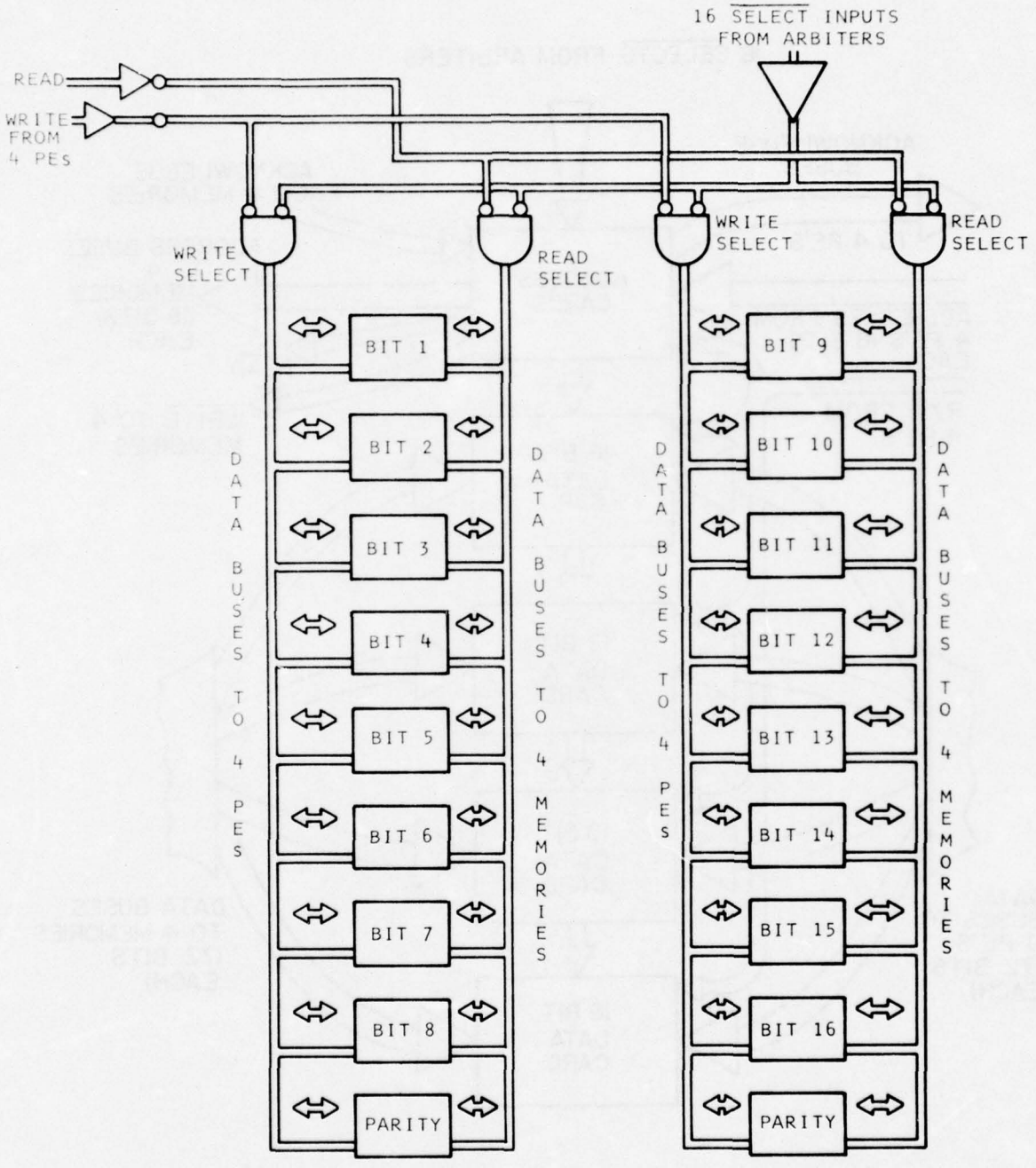


FIGURE 2.4.1-2
DATA CARD

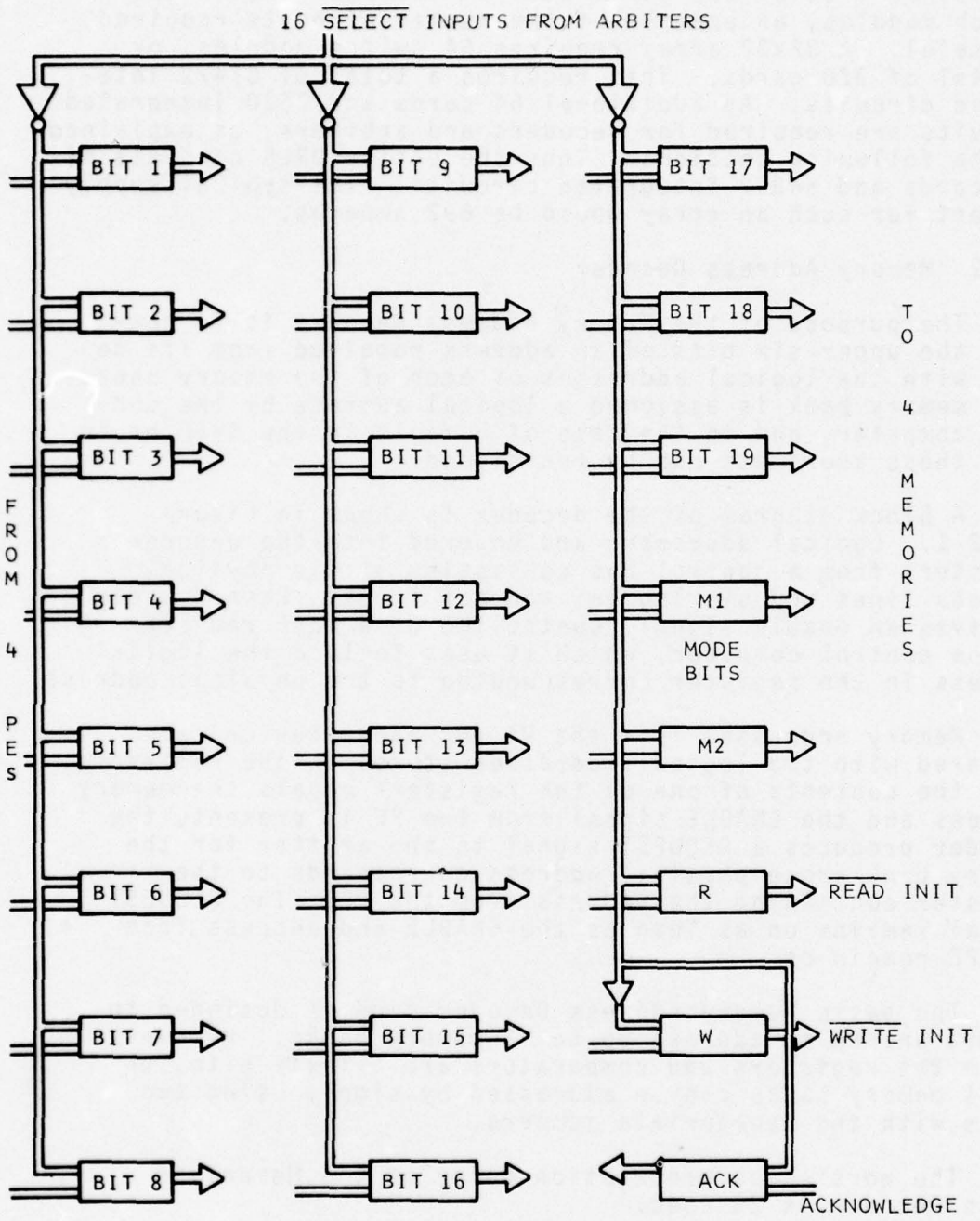


Figure 2.4.1-3. Address Card

Since most of the parts used in the DREA are in the switch modules, an example of the number of parts required is useful. A 32x32 array requires 64 switch modules, or a total of 320 cards. This requires a total of 57472 integrated circuits. An additional 64 cards and 7520 integrated circuits are required for decoders and arbiters, as explained in the following sections. Thus the entire DREA consists of 384 cards and 64992 integrated circuits. The typical supply current for such an array would be 692 amperes.

2.4.2 Memory Address Decoder

The purpose of the Memory Address Decoder is to compare the upper six bits of an address received from its device with the logical addresses of each of the memory banks. Each memory bank is assigned a logical address by the control computer, and in the case of a fault in the DREA or in CWS, these addresses can be reassigned.

A block diagram of the decoder is shown in Figure 2.4.2-1. Logical addresses are entered into the decoder's registers from a control bus consisting of six physical address lines and six logical address lines. Each decoder receives an enable signal, controlled by a mask register in the control computer, which it uses to load the logical address in the register corresponding to the physical address.

Memory addresses from the PE (or other device) are compared with the logical addresses stored in the registers. When the contents of one of the registers equals the memory address and the ENABLE signal from the PE is present, the decoder produces a REQUEST signal to the arbiter for the memory bank whose physical address corresponds to the register containing the address from the PE. The REQUEST signal remains on as long as the ENABLE and address from the PE remain on.

The basic Memory Address Decoder card is designed to permit one PE to address up to 32 memory banks. However, since the registers and comparators are all six bits, up to 64 memory banks can be addressed by simply using two cards with the appropriate jumpers.

The worst-case propagation delay of the Memory Address Decoder is 24 nsec.

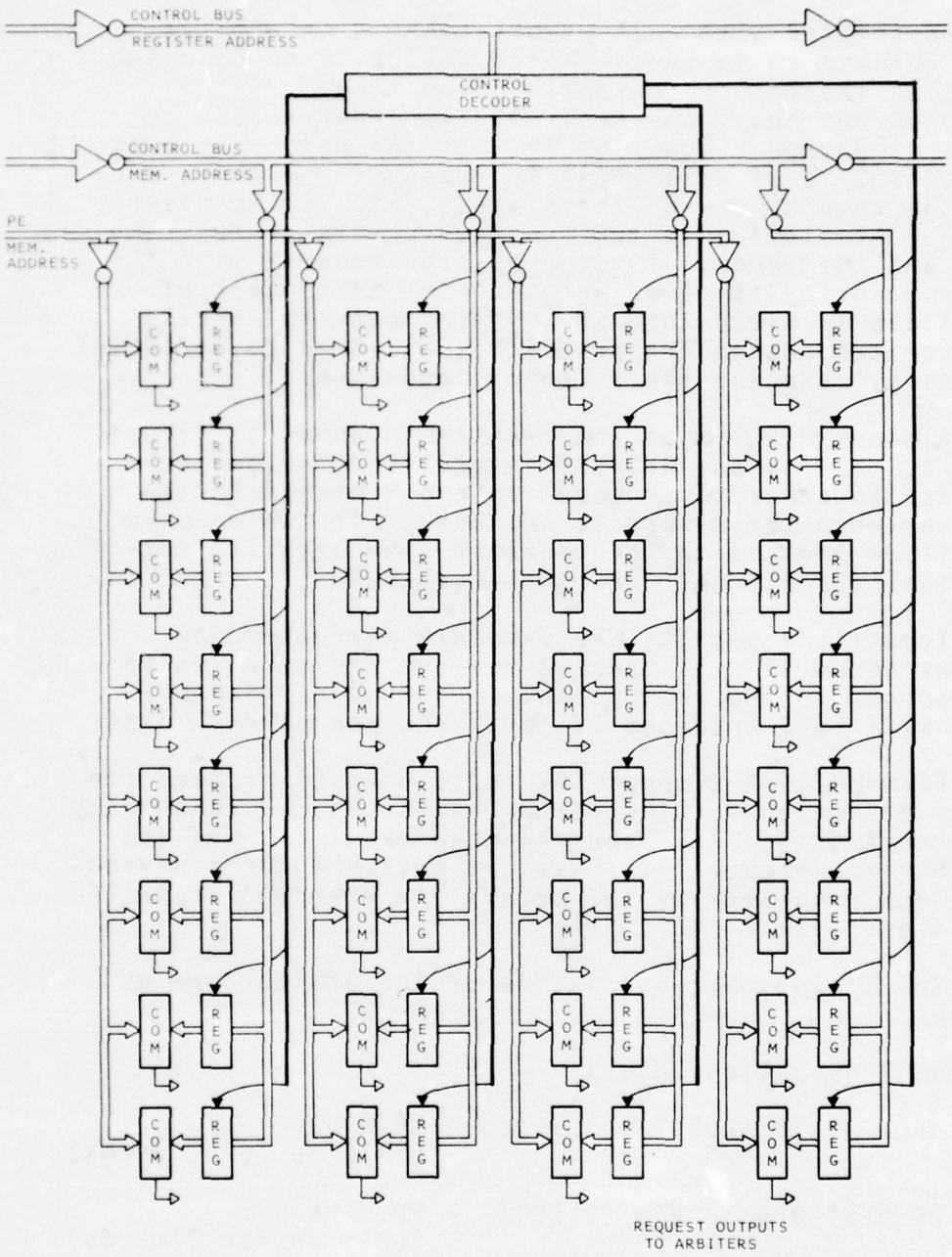


Figure 2.4.2-1. Memory Address Decoder

2.4.3 Arbiter

Associated with each memory bank is an Arbiter, whose purpose is to choose which device is to be given access to the memory bank at a given time. If the memory is not busy, the Arbiter will grant access to the first device requesting it. If the memory is busy, all requests must wait until it ceases to be busy; and then the request having the highest priority is given access. Similarly, if two requests arrive at the same time, and the memory is not busy, the request with the higher priority is given access. The assignment of priorities is wired into the system, but the control computer can change the priorities assigned to different devices by changing their logical addresses.

A block diagram of the Arbiter is shown in Figure 2.4.3-1. All of the REQUEST inputs from the decoders are ORed together to produce the clock signal to the register when the memory is not busy. If two or more flip-flops are set in the register, the priority network inhibits all but the right-most output.

Inputs and outputs are provided for cascading Arbiter cards if more than 32 devices are required to have access to the memory bank. However, additional delay will be introduced if more than one card is used.

Figure 2.4.3-2 shows the logic used to produce the Memory BUSY signal. The memory is considered to be busy whenever a device has been selected and still has its REQUEST line high. This makes it possible for a device to maintain control of the memory for READ/MODIFY/WRITE operations, etc.

The timing characteristics of the Arbiter are as follows:

Maximum Propagation Delays -

REQUEST TO SELECT	- 44.5 ns (65.5 ns for 2 cards)
REQUEST OFF TO NEW SELECT	- 50.5 ns (62.5 ns for 2 cards)

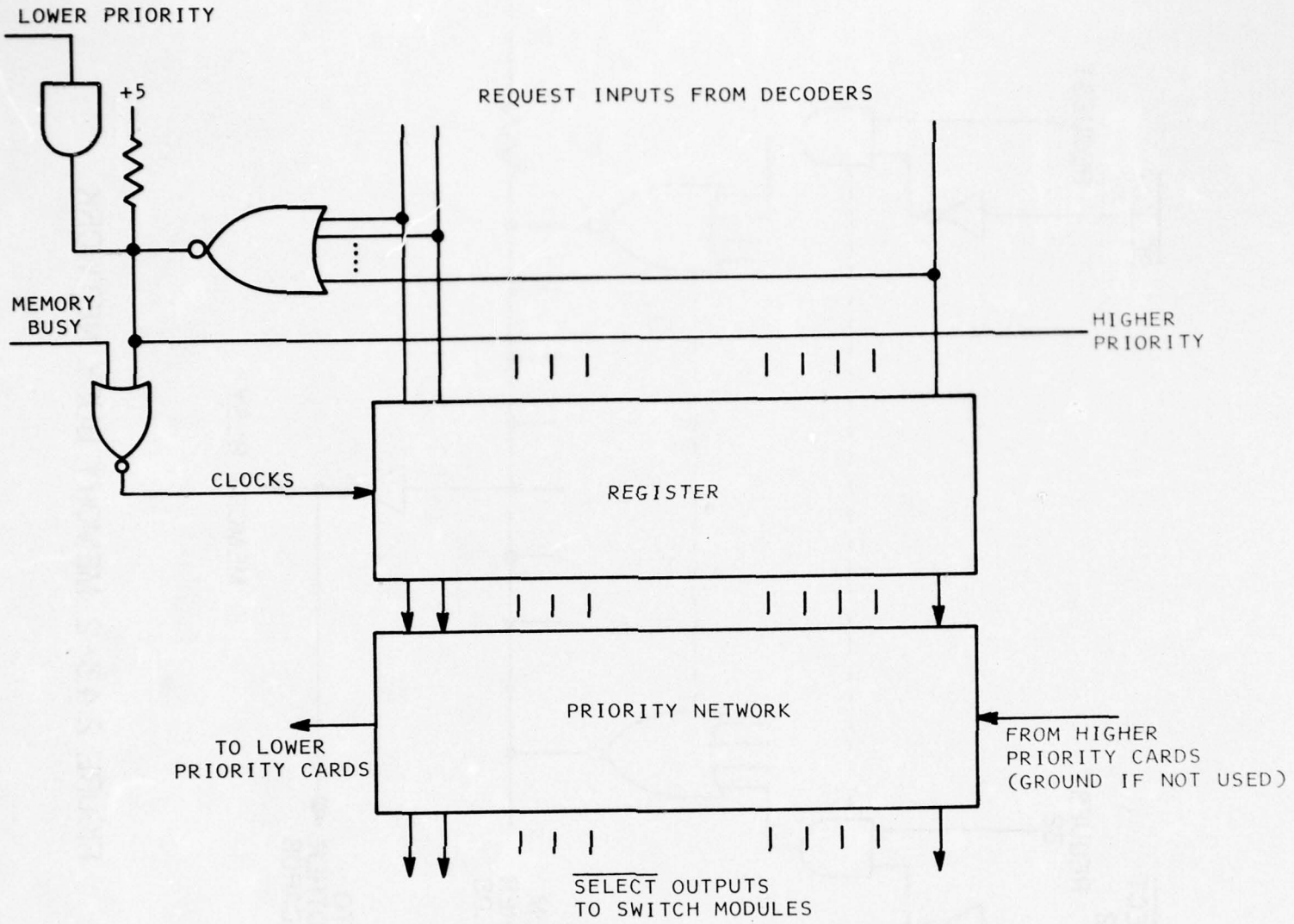


FIGURE 2.4.3-1 DREA ARBITER

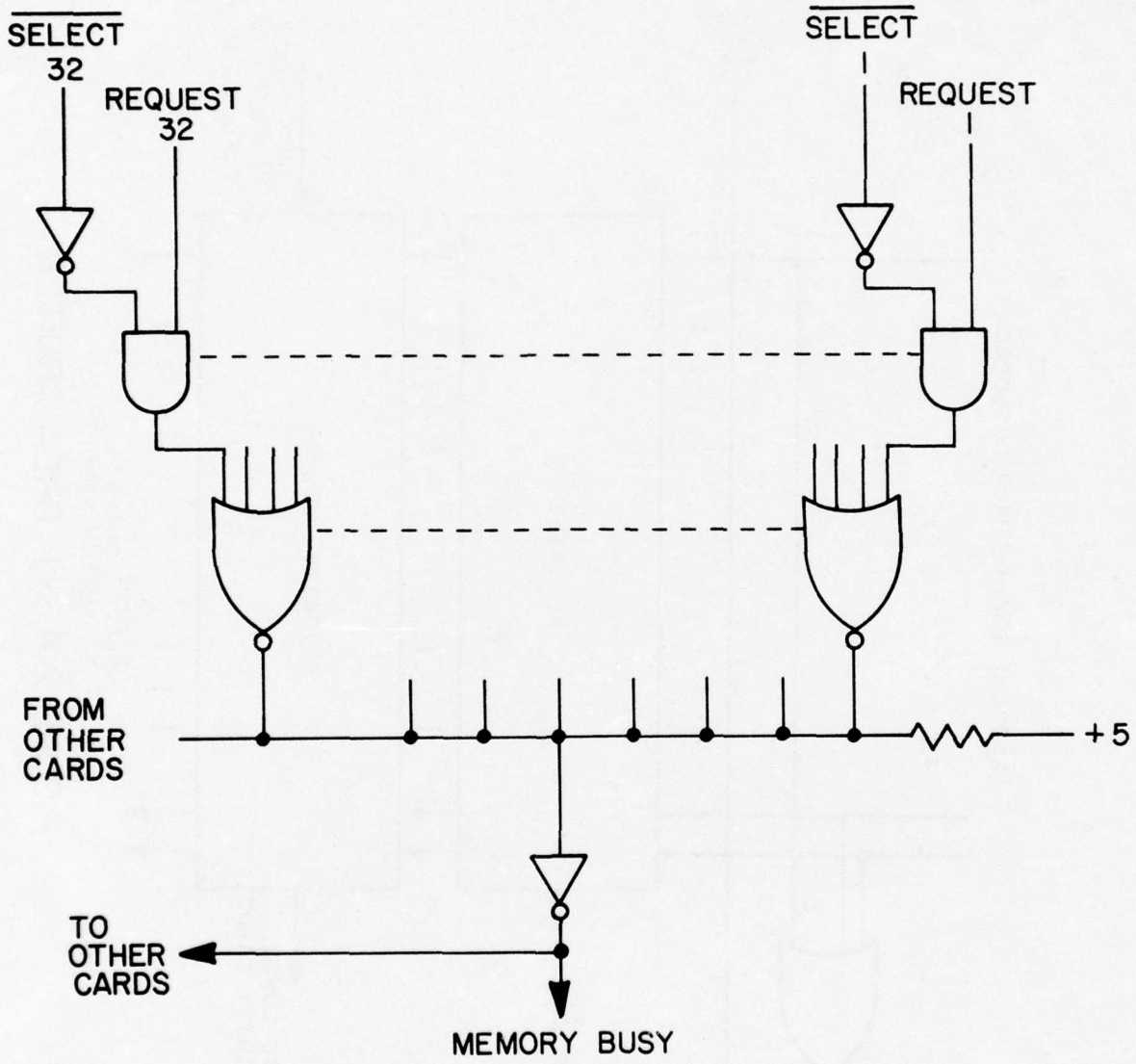


FIGURE 2.4.3-2 MEMORY BUSY NETWORK

2.4.4 DREA System Timing

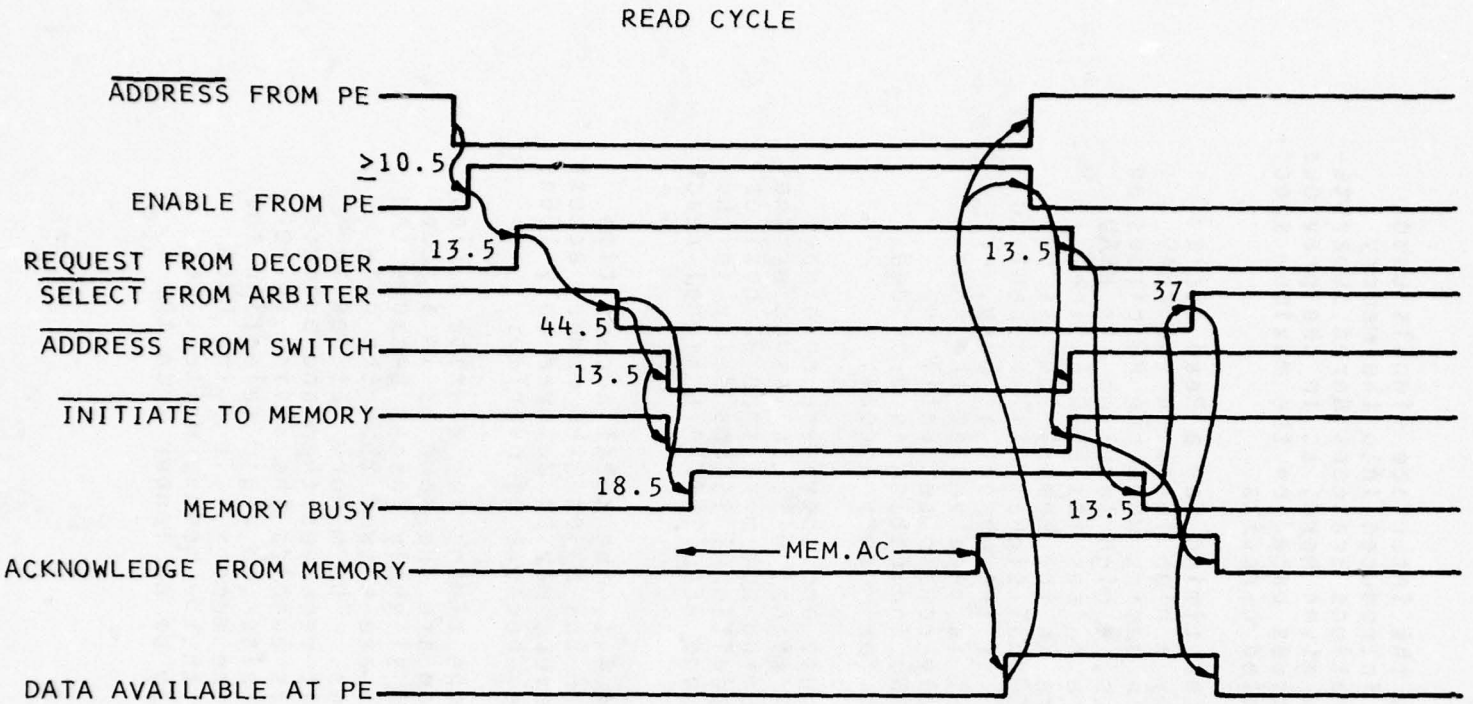
This section describes the interface signals used by the DREA and the delays introduced into the memory cycle. Read and Write operations are considered separately. All of the delay times given here, as in the previous sections, are worst case values based on the maximum specified delays for the integrated circuits.

Figure 2.4.4-1 shows the timing for a Read cycle. A Read cycle is specified by a high level on the READ line from the PE. The cycle begins when the PE places an address on its address lines, a high level on its READ line, and after 10ns or more raises its ENABLE line. This enables the comparators in the Memory Address Decoder. Within 13.5ns, a REQUEST signal will be sent to the proper memory Arbiter. If the memory is busy, or if a higher priority REQUEST is received at the same time, the REQUEST will simply remain set until the memory becomes available. No indication is provided to the PE that it must wait for memory access.

If the memory is not busy and there are no higher priority REQUESTs, a SELECT signal will be issued by the Arbiter within 54.5ns (assuming not more than 32 devices in the system). After an additional 13.5ns delay in the switch module, the address and READ INITIATE signal reach the memory.

As shown in Figure 2.4.4-1, the total access time is delayed an additional 110.5ns beyond the memory access time. The cycle time, however, may be delayed only 106ns over the memory access time, because of overlap.

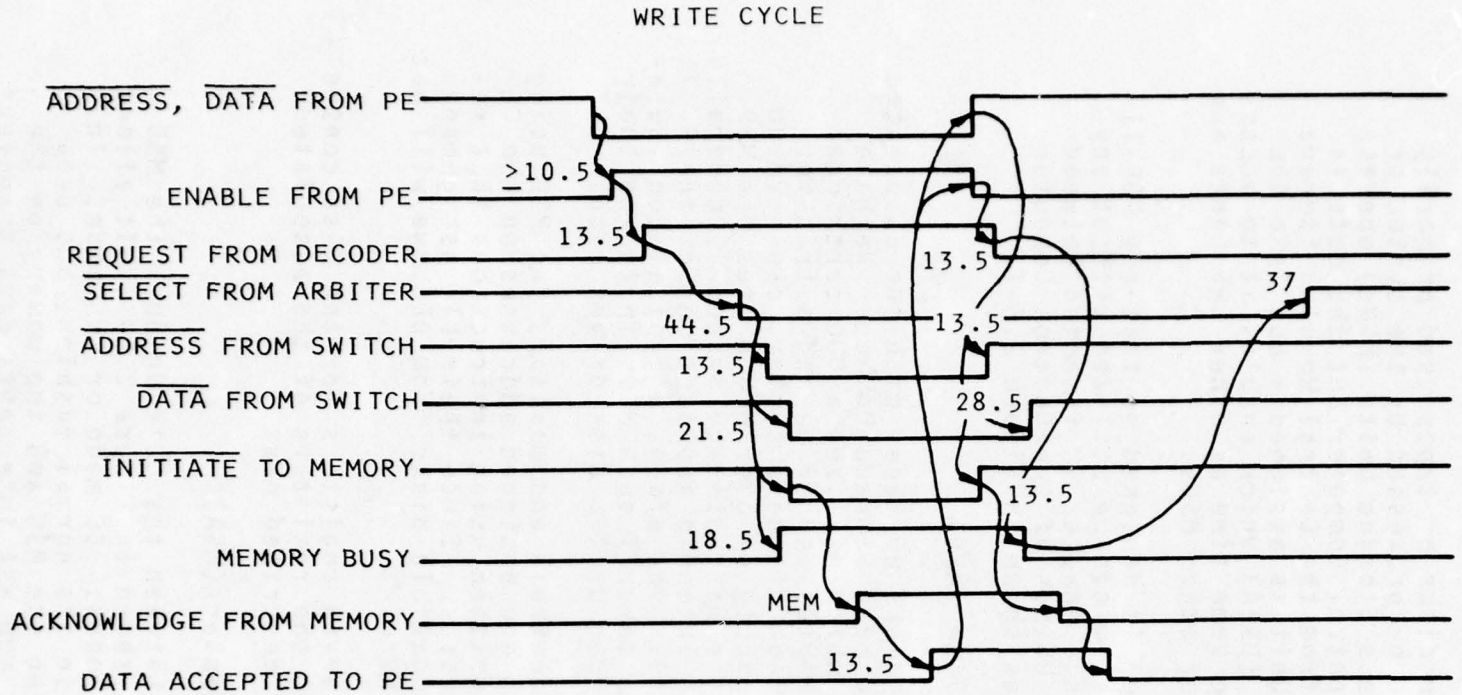
Figure 2.4.4-2 shows the timing for a write cycle. The decoding and arbitration are the same as in a read cycle. The switch delay is slightly longer because the bidirectional data switches are slower than the unidirectional address switches. The memory latches in the data and address when it receives them and produces an ACKNOWLEDGE signal. This enables the device to remove its REQUEST and the Arbiter to begin selecting the next device for access to the memory. Thus the DREA delays can be overlapped with the memory write time, and the total cycle time may be no longer than the memory cycle.



PE ADDRESS	→ REQUEST	24 ns	
REQUEST	→ SELECT	44.5 ns	
SELECT	→ MEMORY ADDRESS	13.5 ns	
MEMORY ADDRESS	→ DATA AVAILABLE	M	MEMORY ACCESS TIME
DATA AVAILABLE	→ DATA AT PE	28.5 ns	
	TOTAL	110.5 ns + M	

CYCLE = 106 ns + MEMORY ACCESS

Figure 2.4.4-1. Read Cycle Timing



PE ADDRESS	→ REQUEST	24	ns
REQUEST	→ SELECT	44.5	ns
SELECT	→ DATA AT MEMORY	21.5	ns
	TOTAL	90	ns

CYCLE = MEMORY CYCLE

Figure 2.4.4-2. Write Cycle Timing

2.5 Mass Storage System

Mass Storage refers to disks or tapes used primarily to hold large data files to be processed by the system or produced by the system. Mass Storage Units (MSUs) operate under the control of the Control Computer but the data is usually transferred to or from the Central Working Storage (CWS). Each Mass Storage Unit is assigned a bus into the Data Routing Element Array (DREA) which enables it to access any of the CWS banks at the same time as other CWS banks are being accessed by the PEs or other MSUs.

The Mass Storage system is designed so that any PDP-11 compatible disk or tape system can be utilized without any modification. A Mass Storage Adapter is inserted between the device controller, the Unibus of the Control Computer and the bus into the DREA as shown in Figure 2.5-1.

2.5.1 The Mass Storage Device (MSD)

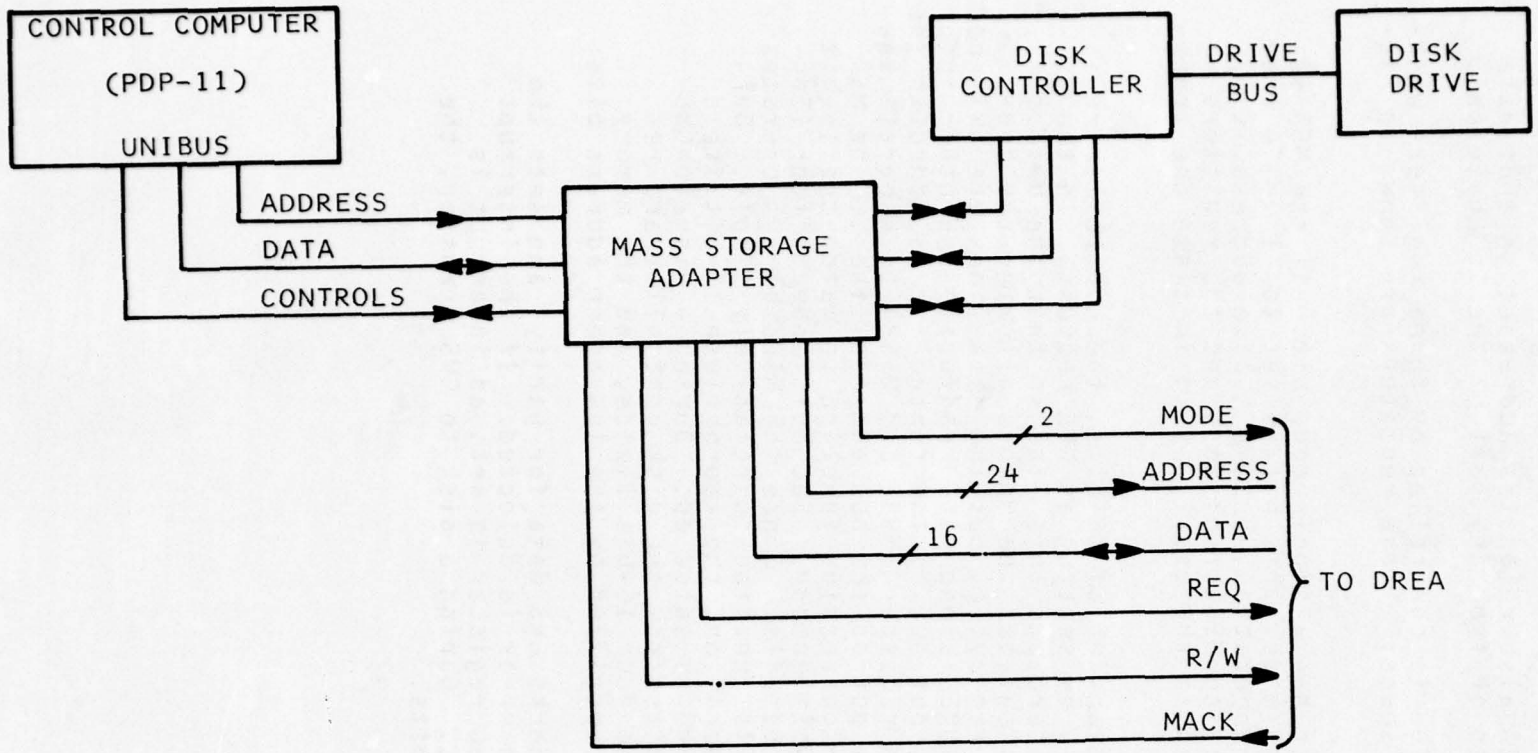
Any PDP-11 compatible disks or tapes with the associated controller can be used as a Mass Storage Device. We have assumed a Diva DD54 system which utilizes a CDC cartridge. It has a capacity of 82.1 Mbytes per drive and a transfer rate of 1209 kbytes/sec. Each of these systems can have up to 8 drives for a total capacity of over 656 Mbytes per MSD. However, the bandwidth of the Mass-Storage System is generally the limiting factor for the intended applications of the G-471, rather than storage capacity: Therefore, a typical configuration is more likely to have several such MSDs including their disk controllers with just one or two disk drives each.

The Control Computer controls and monitors the MSD using seven device registers that are assigned addresses on the CC Unibus and can be read or written using instructions that refer to these register addresses. Since their bit assignments are the same as for the compatible disks from DEC, we will not describe them any further here.

In addition to these seven registers for the disk controller, the CC also has to set the registers of the associated Mass Storage Adapter to be described next.

2.5.2 The Mass Storage Adapter (MSA)

The MSA is connected between the CC's Unibus, the MSD controller and the DREA as shown in Figure 2.5-1. It allows the MSD to operate in two modes: CC mode or CWS mode. In CC mode, the MSD operates in its normal fashion, and data transfers take place between the MSD and the memory on the CC's Unibus. In CWS mode, the MSA intercepts data transfers and redirects them so that data is read from or written to



MASS STORAGE SYSTEM - BLOCK DIAGRAM
FIGURE 2.5-1

CWS instead. The MSA also relocates addresses so that data can be transferred to or from any part of the 16 Mbyte CWS.

The MSA operation is controlled by three registers within the CC's address space. These registers are shown in Figure 2.5.2-1.

The MSC register has a single mode bit. If the MSD is to be utilized in CWS mode, this bit is set to '1'. To execute a data-transfer operation, a 24-bit base address is loaded into the MSB1 and MSB2 registers and the registers of the MSD are loaded in the usual way to initiate the transfer.

For each data transfer of 16 bits, the controller requests bus mastership by setting an NPR request. In the CWS mode, the MSA intercepts this line so that the Unibus is unaffected. It simulates the Unibus allowing the controller to complete its cycle, but the data lines are switched to the bus into the DREA along with read/write and other information. The address put out by the controller is added to the contents of the base register (MSB1 and MSB2) to generate the CWS address which is put on the bus address lines. The MSA also performs a mode conversion function to match the 16-bit disk word to the 64-bit memory. See Multimode Shifter (Section 2.2.1.7.7) for details. Since the MSDs always operates in a 16-bit mode, this function is relatively simple. During a CWS-to-disk operation, the appropriate 16-bit field of the 64-bit CWS word is selected. During a disk-to-CWS cycle, the 16 data bits from the disk controller are repeated in each of the four 16-bit fields, and the memory clocks in the bytes specified by the low order address bits.

The MSA also checks CWS data for parity, and sets the CWS parity bit if an error is detected. If the interrupt enable bit of the MSC register is set, an interrupt is generated for the CC. During a disk to CWS transfer, the MSA inserts parity bits.

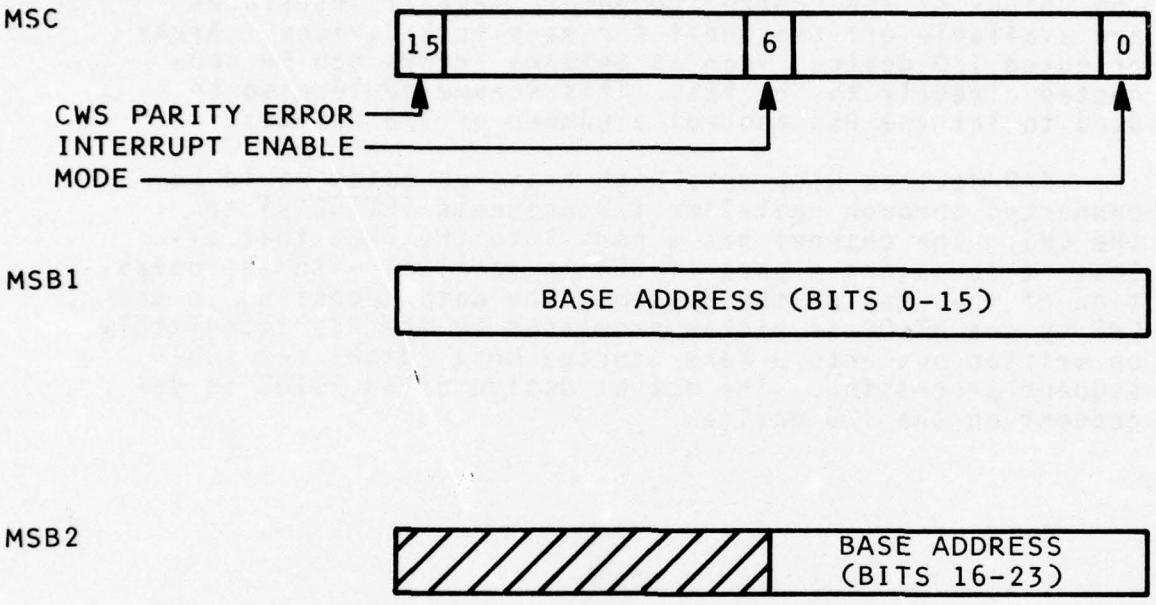


FIGURE 2.5.2-1
MASS STORAGE ADAPTER
CONTROL REGISTERS

2.6 Real Time Input-Output

The G-471 system with its enormous processing power is particularly useful for real-time applications.

Real time devices can be connected to the system in several ways. Slow speed devices are best connected to the Unibus of the Control Computer. PDP-11 interfaces are available off-the-shelf for many such devices. Array-oriented I/O devices such as antenna arrays can be connected directly to the PEs. This scheme could also be used to let the PEs control a number of I/O devices.

I/O devices with very high transfer rates would be connected through Real-Time I/O channels (RTIOC's) to the CWS. The channel has a port into the DREA that allows it to access a part of CWS in parallel with the operation of the rest of the system. The data deposited in the CWS by the RTIOC is either processed by the PEs immediately or written out onto a Mass Storage Unit (disk) for subsequent processing. The actual design of an RTIOC is dependent on the I/O device.

2.7 Packaging, Cost and Power Consumption

This section describes the physical arrangement of the system, cost and power estimates for each of the major elements, and summarizes the total packaged hardware cost for given computational requirements. The computational requirements of a given application can be estimated as described in Chapter 4.

2.7.1 Packaging

Most components of the system are mounted on 16"x16" printed circuit boards. The boards are inserted into standard edge connector blocks, which provide 216 pins per board (6 sections of 36 pins each, 0.125" spacing). In the few cases which require additional pins, cable connectors are provided on the opposite end of the boards.

The boards are mounted vertically in the racks, with the connectors forming a backplane (see Figure 2.7.1-1). With 24" racks, a total of 44 boards are mounted on each of the four levels. Thus, up to 176 boards may be mounted in each rack. With 19" racks, this number would be reduced to 136.

2.7.2 Cost and Power Consumption

In this section estimates of cost and power consumption are obtained as a function of the size of the system. For this purpose the system will be divided into the following blocks:

1) Control Computer Complex
2) Processing Element (without PELS)
3) PELS bank
4) CWS bank
5) Data Routing Element Array
6) Mass Memory
7) Real Time I/O

For a given number of Processing Elements, Memory banks, and Real Time I/O channels, estimates of total cost and power consumption can be obtained by adding the individual figures given in the following sections. Section 2.7.3 summarizes the cost calculations.

The power estimates in the following sections are obtained by adding the typical power-supply current specifications for the integrated circuits. These estimates are used to specify the power supplies required, and are also used in the cost estimates.

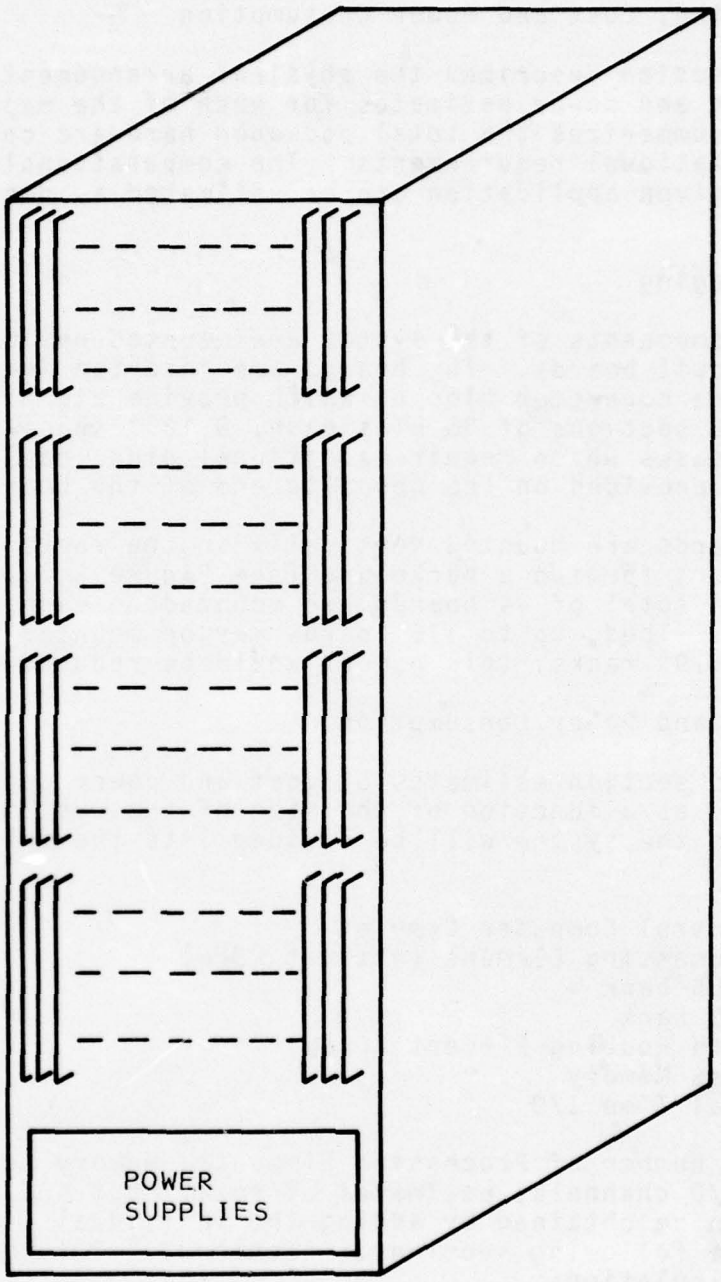


Figure 2.7.1-1. Arrangement of Circuit Boards

Estimates of total cost are obtained as follows: First, the actual costs of the integrated circuits are added, in most cases using 1000 quantity prices. Then, an additional $0.85 per IC is added to cover hardware and assembly costs. This includes boards, connectors, mounting hardware, racks, cables, etc. and assembly of the boards and the backplane. Another $0.50 per IC is added for test costs. (Test costs are reduced to $0.10 for circuits which are extremely simple). In addition to these costs, a power supply cost of $5 per ampere is added to obtain the total cost for each block.

2.7.2.1 Control Computer Complex

The Control Computer is normally a PDP-11/40 minicomputer. Peripherals included in the Control Computer Complex will typically be one or two disk units, a magnetic tape drive, a CRT console, and a printer. All of these are standard DEC units and could be replaced by any similar computer system which the customer may have. In addition the Control Computer Complex includes a PE Array Controller which functions as an interface with the rest of the system. The estimated cost of a typical Control Computer Complex, including the above peripherals and a PE Array Controller, is $54,000.

2.7.2.2 Processing Element

Each Processing Element consists of 19 boards plus 2 boards for each bank of PELS. Table 2.7.2.2-1 lists the boards and the total number of integrated circuits, power, and cost of the integrated circuits for each board.

The total cost of the integrated circuits for a PE is $7045, as shown. Assembly and testing of the boards and the cost of miscellaneous hardware add up to $3510; and the average power supply cost is estimated to be $568. Thus, the total cost for each PE is $11,123, excluding PELS.

2.7.2.3 PELS

Each PELS memory bank consists of two boards, containing a total of 231 integrated circuits. The power required is 13 amperes, and the integrated circuit cost is $1,068. Thus, the total cost of a PELS bank is $1,445.

TABLE 2.7.2.2-1

PROCESSING ELEMENT

SECTION	BOARDS	ICs	POWER	IC COST
CAPU	5	678	33.5 amps	$1,762.
IOAG	2	282	13.7	720.
ADDER	1 1/3	308	10.9	504.
MULTIPLIER	2	560	23.0	1,352.
PE BUS CONTROL	3	203	3.7	207.
ADDRESS MAPPER	1	80	3.6	112.
PE CHANNEL	1	192	9.5	295.
AP BUS CONTROL	3	297	13.8	1,493.
MICROPROCESSOR	2/3	-	1.8	600.
TOTALS	19	2600	113.5 amps	$7,045.

NOTES: IOAG includes the AP Controller

PE Bus Control includes the CC COM Port.

2.7.2.4 CWS

Each bank of Central Working Storage consists of a control board and a set of memory boards. Up to 8 memory boards, each containing 64 kbytes of RAM, can be included in a bank. The cost and power estimates for both types of CWS boards are shown in Table 2.7.2.4-1.

TABLE 2.7.2.4-1

CWS BOARDS

	ICs	POWER	IC COST	TOTAL COST
Control Board	35	2 amps	$ 82.	$ 139.
Memory Board	200	5.5 amps	1,281.	1,499.

Examples:

Control Board and 1 Memory Board - $ 1,638. Total Cost

Control Board and 8 Memory Boards- $12,131. Total Cost

2.7.2.5 Data Routing Element Array

The Data Routing Element Array consists of a set of 4x4 switch modules and decoder and arbiter boards. The size of the DREA is specified by two constants, M and N, defined as follows

$$M = \left\lceil \frac{\text{Number of CWS Memory Banks}}{4} \right\rceil$$

$$N = \left\lceil \frac{\text{Number of PEs, Disks, IO Channels} + 1}{4} \right\rceil$$

N includes all devices which require access to CWS, including the Control Computer.

For given values of M and N, the DREA consists of 5MN switch boards, 2N decoder boards and 2M arbiter boards. Total cost and power estimates are given in Table 2.7.2.5-1, in terms of M and N, and for two typical arrays.

TABLE 2.7.2.5-1

DATA ROUTING ELEMENT ARRAY

	FUNCTION OF M,N	20x20 SYSTEM	32x32 SYSTEM
NUMBER OF BOARDS	5MN +2N +2M	145	352
NUMBER OF INTEGRATED CIRCUITS	898MN +668N +272M	27150	64992
COST OF INTEGRATED CIRCUITS	$244MN +$719N +$187M	$10630	$22864
POWER SUPPLY CURRENT	6.1 MN +24.4 N + 3.7 M amps	293 amps	616 amps
TOTAL COST	$1127 MN +$1475 N +$ 463 M	$37865	$87632

2.7.2.6 Mass Storage

Various options for Mass Storage are listed in Table 2.7.2.6-1, with their costs. The total amount of storage and the transfer rate required may vary greatly with the application. In considering the cost, it should be remembered that each Mass Memory controller requires a port on the DREA.

2.7.2.7 Real Time I/O

If it is necessary to process data in real time at a very high rate, the input data may have to be placed directly in CWS. For this purpose, a Real Time I/O Channel, with an estimated cost of $1000, may be added to the system. As with Mass Memory controllers, each Real Time I/O Channel requires a port on the DREA.

TYPE	KEY CHARACTERISTICS		COST/
	TRANSFER RATE, MBYTES/SEC	TOTAL STORAGE, MBYTES	
DISK	1.2	320	50
		160	30
		80	20
DISK	.806	92	36
DISK	.312	40	21
DISK	.180	2.4	13.7
DISK, FLOPPY	.01	.256	3.9
CCD	2+	ANY	1.6¢/ BYTE

Table 2.7.2.6-1. Cost Estimates for Mass Storage

2.7.3 Calculation of Total Cost

The total cost of the packaged hardware can be calculated simply by adding the costs described in the previous sections. Thus the total cost is given by the following formula.

$$C_{TOT} = C_{CCC} + C_{PE} + C_{DREA} + C_{CWS} + C_{MSS} + C_{RTIO}$$

where

C_{CCC} = \$54K (Control Computer Complex)

C_{PE} = (\$11K + \$1.45K x N_{PELS}) N_{PE}

For N_{PELS} banks of local storage per PE

C_{DREA} = \$1.13K MN + \$1.47K N + \$.46K M
(see Section 2.7.2.5)

C_{CWS} = (\$.14K + \$1.5K x N_{RAM}) x N_{BANKS}

For N_{RAM} cards of 64kbytes per bank.

C_{MSS} = \$10K per controller
+ \$10K per disk drive (up to 4 per controller)

C_{RTIO} = \$1K per real time I/O Channel

2.7.3.1 Cost of a Typical System

As an example of the cost calculations, a typical large system is considered in this section. The system includes 20 PEs, each containing 3 PELS banks, a 4 Mbyte CWS divided into 32 banks, and 2 disk drives with separate controllers.

Since a total of 23 devices require access to CWS, the DREA parameters are N = 6 and M = 8. Therefore C_{DREA} = \$1.13Kx MN + \$1.47KxN + \$.46KxM = \$66.74K. The cost of the PEs is C_{PE} = (\$11K + \$1.45K x N_{PELS}) x N_{PE} = \$307K. The cost of CWS is C_{CWS} = (\$.14K + \$1.5K x N_{RAM}) x N_{BANKS} = \$100.48K for 32 banks of 2 RAM cards each.

Using the above values, the total cost is

$C_{TOT} = C_{CCC} + C_{PE} + C_{DREA} + C_{CWS} + C_{MSS} + C_{RTIO}$

= \$54K + \$307K + \$66.74K + \$100.48K + \$40K + 0
= \$568K.

3.0 Software

3.1 The Multi-Processor Operating System (MPOS/471)

The successful operation of a new computer system is heavily dependent upon the support software available to the applications programmer. It is becoming increasingly evident that the Operating System (here we include support programs such as compilers, utilities and subroutine libraries in addition to the Operating System proper) has at least as important a role to play in determining the effectiveness of a system as the architecture of the hardware.

The primary objective of the system software proposed here is to provide a hospitable environment for the user who wants to utilize the system to solve his problems taking advantage of the capabilities inherent in the hardware.

The following are some of the desirable characteristics of the system:

1) A user familiar with programming in FORTRAN on a typical sequential computer system can write programs on the G-471 without a lengthy learning period. Most array-oriented operations, especially those frequently involved in signal processing, are available as library subroutines that can be called from FORTRAN programs. The user does not have to be particularly aware of programming a multi-processor system. The library subroutine farms out the task to a set of PEs and also takes care of such details as the program in the PE microprocessor, Arithmetic Processor and IOAG, movement of data between the Central Working Storage and the local memories in the PEs (PELS) and communication and synchronization among the various hardware units operating simultaneously. This mode of operation is particularly suitable for problems that involve heavy use of standard number-crunching operations such as FFTs and other array operations.

2) A sophisticated user can program the system at a much lower level, if he chooses to do so in order to fully utilize all the processing power that is provided by the hardware. This might be desirable for production-type programs that are run very frequently, especially if speed is a very critical factor. An example would be a system for real-time processing of high-resolution pictures; here it would be worthwhile putting in a greater amount

of programming effort to achieve the highest possible throughput. A system programmer can also write additional routines to add to the standard subroutine library. Extensive support is provided for this kind of low-level programming, since the G-471 system is intended to be used frequently in the manner described above.

The UNIX system provides for a very convenient and flexible environment for program development in general. In addition, several programs are included to facilitate the development of programs for the specific hardware environment of the G-471. The controlling programs running in the CC as well as the programs to be executed by the PE microprocessors can be written in assembly language or in FORTRAN or C (or in any of several higher-level languages for which compilers are available to run on a PDP-11). Microprograms for the AP and the IOAG are written in their specific assembly languages, and processed by cross-assemblers on the CC.

Run-time services are provided by system programs such as the PECCP which handles synchronization and inter-PE communication, loading of PE routines, etc., and the PE resident monitor which handles the various hardware modules within the PE such as the AP, the IOAG, the channel and the mapper. All these system programs have FORTRAN-compatible interfaces; i.e. they can be invoked from FORTRAN programs by means of subroutine or function calls.

A PE simulator makes it convenient to check out programs on the CC by creating "virtual" PEs and CWS, and providing an interactive debugging environment. This facilitates program development on the CC, or even on a different PDP-11 system, while the system is running production jobs.

3) Several independent programs can run at the same time each utilizing a subset of the available PEs. This is especially significant for program development. In addition, it is possible to have purely sequential programs running simultaneously on the CC. It is easy to write programs that will determine the number of PEs to use at run time, depending upon the number available, or as instructed by the user. At the same time, one can write a program that will utilize certain specific PEs, as may be required if some PEs have additional hardware, or different amounts of local memory (PELS). Simultaneous programs are protected against each other while being able to share resources and communicate with each other.

3.2 System Modules

In this section, we describe the functional modules which comprise the MPOS/471 operating environment. The first seven modules (Section 3.2.1 through 3.2.7) are, in fact, already available in the underlying UNIX system, and these will be mentioned very briefly. If an operating system other than UNIX is selected as the base, the basic functions of these modules should be provided by it but the details could differ.

The remaining modules described in Sections 3.2.8 through 3.2.18 are, for the most part, in the form of libraries of subroutines to be linked with a program to provide it with additional services. Some of these which are used most frequently, may be included in the main operating system. The services of these modules are invoked by executing FUNCTION calls from FORTRAN. The functions and all parameters are of type integer (16-bit) unless otherwise noted. From an assembler language program, the corresponding calling sequence can be used. For resident monitor functions (such as those described in Section 3.2.14 and any other modules to be made part of the operating system proper), traps can also be used by assembly language programs for greater efficiency. In either case, appropriate macros are included in the system macro library to make all such calls uniform.

Some of these modules, such as the CAPU and IOAG cross assemblers, the Multi-Processor Linker (MPL), etc. would usually be executed directly as system commands or programs by the user from a terminal. However, in the interest of flexibility and uniformity, all modules of MPOS/471 are in the form of subroutines, and appropriate driver programs are provided to interface between a user at a terminal and the modules.

3.2.1 The Scheduler and Dispatcher

The function of the Scheduler and Dispatcher is to allocate the CC C.P.U. to the several processes being executed simultaneously. This includes maintaining queues and assignment of priorities (scheduling) and actually switching the CPU between processes (dispatching).

The operating environment envisaged for MPOS/471 includes multiprogramming on the CC. The PEs in the processing element array can be subdivided into subsets working on different computations. One program on the CC sequences and controls the PEs involved in each computation. Even if the normal mode of operation of the system consisted of a single computation utilizing all the PEs, it would be desirable to be able to operate in the mode described above for purposes of program development. In addition, multiprogramming on the CC makes it convenient to utilize the CC to execute sequential support programs of various types (such as I/O spooling, compilations, etc.) during the time it is not actively involved in the main computation.

It is also desirable, though not essential, to have facilities for a process to spawn multiple processes that operate logically in parallel. (Multi-tasking).

This makes it possible to have separate processes on the CC handle subsets of the PEs working simultaneously on different parts of the same problem. While it may always be possible for a single CC process to be used for this purpose, the program logic would be greatly simplified in many cases by the ability to use multiple processes for a single computation.

The UNIX operating system includes facilities of the type mentioned above. See the UNIX Reference Manual for details.

3.2.2 The Process Manager

The Process Manager is responsible for creating and deleting processes and for keeping track of the status of all processes through the maintenance of process tables. This function, again, is performed by the base system (UNIX).

3.2.3 CC Disk Handler and File Manager

This functional module of the system is responsible for all interactions with the disk, queueing and buffering, and file management including the maintenance of a directory

structure and providing protection facilities. The UNIX system provides very powerful and flexible facilities for this purpose. (See the UNIX Reference Manual for details).

3.2.4 CC Memory Manager

This module is responsible for allocation of the CC's main storage to different processes and for maintaining the segment registers in the memory management hardware. This function is also performed within UNIX and will not be described further.

3.2.5 Drivers for Various Peripherals

UNIX already includes drivers for most standard peripherals. These peripherals are accessed through "special files" so that there is a fair degree of uniformity in the I/O operations for disk files and all other peripherals.

Compatible drivers (special files) would be added for non-standard peripherals such as displays and image-generation equipment that may be required for an image-processing environment.

3.2.6 *Command Processor*

The Command Processor interfaces between the user at a terminal and the rest of the system, and interprets his commands, calling upon system modules to execute these as required. This function is performed by the SHELL in UNIX.

3.2.7 Terminal Handler

The function of the Terminal Handler is to manage the interfaces to the various terminals from which the system can be accessed. Its functions include buffering, multiplexing/demultiplexing, handling individual terminal characteristics, *line protocols, etc. This function is also in*cluded in the underlying UNIX operating system.

3.2.8 PE Array Control and Communication Package (PEACCP)

The PEACCP provides the facilities for managing, controlling and communicating with the array of processing elements.

Two paths are available for interaction between the Control Computer (CC) and the PEs:

(i) The Processing Element Array Controller (PEAC) provides a direct communication path between the CC and any PE, including interrupt capability in either direction. In addition, the CC can force any PE to halt or disconnect it from the system. See Section 2.1.2 for further details.

(ii) The Central Working Storage (CWS) is directly accessible by the CC and all the PEs and provides a path for highly efficient communication. Special hardware and software is provided to take care of ambiguities and other problems associated with simultaneous access by more than one processor.

The modules comprising the PEACCP are described next.

3.2.8.1 The Processing Element Allocation Manager (PEAM)

The PEAM manages the Processing Elements as a system resource, allocating them to users upon request. It maintains allocation and other information about the PE in the Processing Element Status Table (PEST). System reconfiguration programs can call the PEAM to remove disabled PEs or add PEs to the pool of available PEs.

3.2.8.1.1 PEAM Routines

The following routines are available to system and user programs to invoke PEAM functions. Like all the other MPOS/471 functions, these can be called as FORTRAN FUNCTIONS. In the following, PELST refers to a vector of P.E. numbers ending with -1.

3.2.8.1.1.1 INPEAM (PELST)

This routine is executed only during system initialization to provide the PEAM with a list of P.E.s available for allocation.

3.2.8.1.1.2 ALPE (N, PELST)

N PEs are allocated to the owner of the calling process, if available. The number of PEs actually allocated is returned by the function, and a list of the I.D.s of the allocated PEs is placed in PELST.

If ALPE is called with N=0, the actual PEs in PELST are allocated, if they are available. Otherwise, the function returns a -1.

If called with N=-1, the maximum number of PEs currently available is allocated to the owner of the calling process. As in the normal case, the number of PEs actually allocated is returned as the value of the function, and the corresponding PE numbers are returned in PELST.

3.2.8.1.1.3 DEALPE (PELST)

The PEs in PELST are deallocated and returned to the pool of available PEs. If any of these were not allocated to the caller already, a -1 is returned.

If DEALPE is called with an empty PELST, (i.e., containing just -1), all the PEs allocated to the owner of the calling process are freed.

3.2.8.1.1.4 CATPEA (LIST)

This function returns the allocation status of all the PEs in the 32-word vector LIST (for a configuration with 32 or less PEs). The ith element of LIST, if positive, gives the UID of the user to whom PE number i is allocated. LIST(i) is zero if PE number i is free (available), -1 if it is disabled and -2 if there is no PE number i in the system.

3.2.8.1.1.5 DISAB (PEN)

This routine is used by a system reconfiguration program to remove a disabled PE from the system. PE number PEN is declared as disabled and made unavailable for allocation.

3.2.8.1.1.6 ENAB (PEN)

PE number PEN is added to the pool of available PEs.

3.2.8.2 The Processing Element Task Dispatcher (PETD)

The PETD provides a convenient mechanism for programs running on the CC to farm out tasks to a set of processing elements. When a computation requires a number of operations to be performed, and the logic of the algorithm permits these to be executed in parallel, the facilities of the PETD can be utilized to request the execution of these operations or tasks by a group of processing elements.

In this mode of operation, called the "Dispatched Mode", the tasks are placed in a PE Task List from where they are picked up and executed by the PEs as they become available. The advantages of this scheme are:

(i) The number of PEs to be used for performing a set of operations need not be specified explicitly. It becomes very simple to write programs that are independent of the number of processing elements that are available. Thus if some processing elements are disabled or allocated to another process, the program can still run unchanged.

(ii) If a PE gets left behind, the other PEs automatically pick up a larger share of the tasks instead of having them all wait for the slower PE This kind of load-balancing will work especially well if the problem is broken up into a large number of small tasks to be executed by the PEs. A PE could get left behind the others for one of the following reasons:

(a) the hardware configuration of different PEs could be different;

(b) the discovery of an error may require some computations to be repeated;

(c) excessive memory conflicts in accessing the CWS could delay some PEs more than others. This should be an infrequent situation because of the parallel paths to the different CWS banks, and the provision of considerable amounts of local buffer storage within the PEs (PELS);

(d) different tasks may require different amounts of computation; e.g., iterative programs where the number of iterations is data-dependent.

(iii) There is very little overhead involved in the PE picking up its next task.

(iv) Programming for multiprogramming is greatly simplified.

3.2.8.2.1 Processing Element Task List (PETL)

The PETD sets up a PE Task List (PETL) for each set of PEs executing tasks in parallel for a program on the CC. A PETL (Figure 3.2.8.2.1-1) is basically a ring buffer: each element consists of two 16-bit fields specifying a routine to be executed (RTN) and the address of a parameter list (PAR). (Note that all addresses here are 16-bit virtual addresses within the address space of a PE). A pointer (PTLPTR), which is initially set to PTLB (the beginning of the PETL), is used by the PEs to fetch tasks from the PETL. The hardware of the DREA and the CWS is designed so that a PE has exclusive control of a CWS memory bank during the execution of an instruction involving a Read-Modify-Write cycle. This ensures that a PE can increment the PETL pointer (PTLPTR) in an unambiguous way even though other PEs may attempt the same operation simultaneously. Further, the value of the pointer before incrementing is available in the LIR register located in the PE's I/O register area, which always contains the last input to the PE microprocessor through the mapper. Without this, another PE could increment PTLPTR between the time it is accessed as a pointer and the time that it is incremented, with the result that the two PEs could fetch the same task while another task may be skipped.

See Section 3.2.14.2.6 for a more detailed description of the operation of the PE when fetching tasks from the PETL.

After fetching the value of the RTN field, it is reset to point to the routine RETRY. If the PEs empty out the ring buffer before the program in the CC has had time to add new entries, any PE fetching another task will execute RETRY. This consists in examining RTN until it is different from the address of RETRY, and then executing the routine it specifies in the normal manner.

The routine to execute the task is itself a subroutine using standard linkage conventions, and could be a FORTRAN subroutine.

The PETL ring-buffer is followed by an entry at PTLO with RTN=LUPBAK and PAR=PTLB. The first PE that reaches the bottom of the ring buffer executes the LUPBAK routine which consists

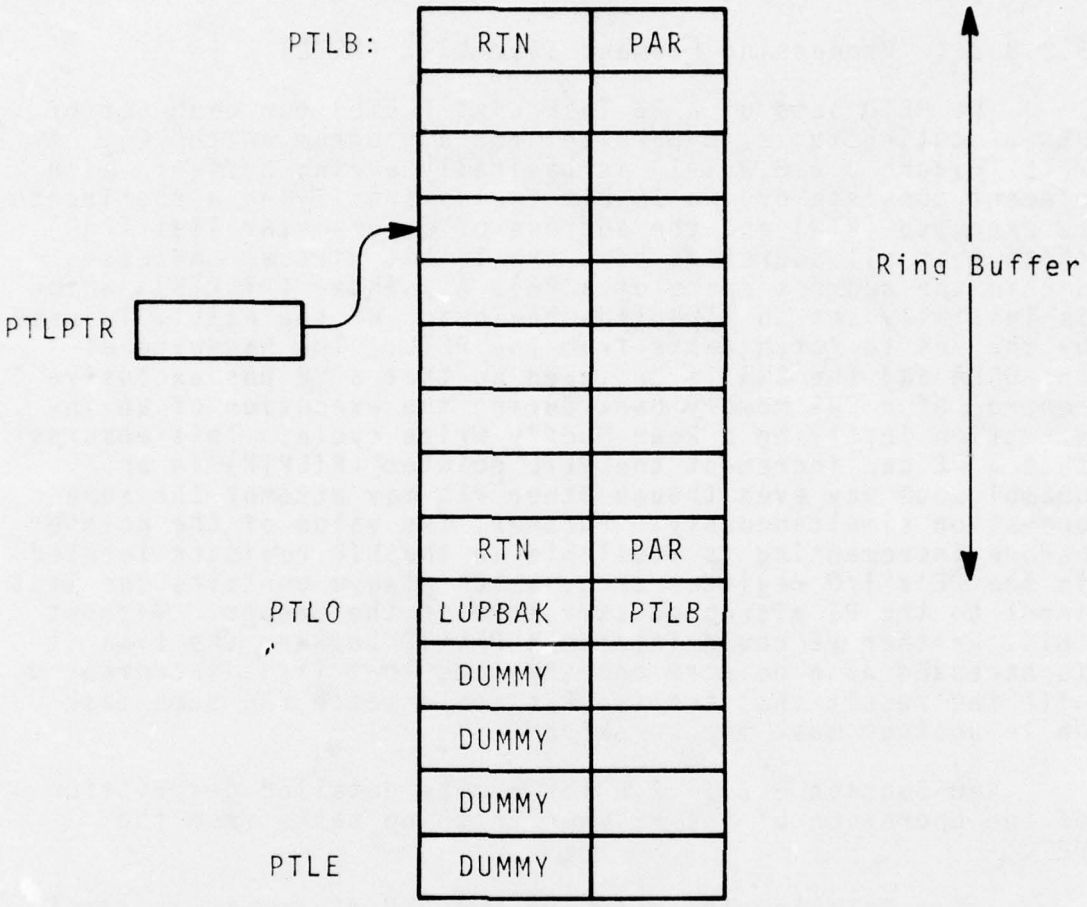


Figure 3.2.8.2.1-1 - Processing Element Task List

simply of setting the pointer back to PTLB, the top of the ring buffer area.

This entry is followed by (NPE-1) dummy entries where NPE is the number of PEs fetching their tasks from the PETL. Each of these has RTN=DUMMY; the routine DUMMY consists of just a subroutine return.

If all the PEs get to the bottom of the PETL at exactly the same time, one of them will execute LUPBAK, resetting the pointer PTLPTR. Meanwhile some of the others will execute the DUMMY routine. Subsequently, all of them will fetch valid tasks from the top of the list.

3.2.8.2.2 PETD Routines

The following FORTRAN-callable subroutines are available to invoke the PETD functions described.

3.2.8.2.2.1 PETLNO = INPETL (PELST, BUFAD, BUFLEN)

This routine sets up a PE task list at BUFAD for the PEs specified by PELST. The number of usable entries in the PETL ring buffer is BUFLEN/4 - #(PELST). The RTN fields of the PETL ring-buffer are set to RETRY, and the overflow area is set as described in section 3.2.8.2.1. The PETLNO returned by the PETD is used in subsequent calls. BUFAD must be in CWS.

3.2.8.2.2.2 ADDENT (PETLNO, RTN, PAR)

An entry is added to the PETL identified by PETLNO. RTN, PAR are the addresses of the routine and the parameter list in the address space of the PEs

3.2.8.2.2.3 DISPATCH (PELST, PETLNO, PTLPTR)

The PEs specified in PELST are directed to begin executing tasks from the PE task list identified by PETLNO. The word PTLPTR in CWS is used as a read-pointer into the PETL. It is initialized to point to the beginning of the PETL.

The PETD does this via the PEAC by executing a

EXECBC (PELST, PDISP)

followed by a

SNMSBC (PELST, MESBUF, 1)

where MESBUF contains the address of PTLPTR. (See sections 3.2.8.3 and 3.2.14.2 for descriptions of EXECBC, SNMSBC and PDISP). The effect is to force the PEs in PELST to execute the routine PDISP and send it the parameter PTLPTR. PDISP is a routine in the PE resident monitor that executes tasks from the PETL using PTLPTR.

3.2.8.2.2.4 RELPTL (PETLNO)

The PETL identified by PETLNO is deallocated.

3.2.8.2.2.5 PETDJOIN(PETLNO,CTR)

This routine is used when all the tasks added to the PETL

so far must be completed before any subsequent tasks are started. For example, when computing a 2-dimensional FFT, all the rows have to be processed before starting with the columns.

The variable CTR, which must be in CWS, is set to NPE, where NPE is the number of PEs in PELST initially specified in the INPETL call. NPE entries of the form (PEJOIN, CTR) are added to the PETL. PEJOIN is a routine in the PE resident monitor. Each of the PEs executing PEJOIN decrements CTR and waits until CTR is zero before continuing with subsequent tasks.

3.2.8.2.2.6 PETDWAIT (PETLNO, CTR)

The PEs stop after the PETL is exhausted and control then returns to the calling CC program.

(NPE-1) entries of the form (DECHLT, CTR) followed by one entry of (DECSIG, CTR) are added to the PETL. The process is then blocked until a "DONE" signal is received from one of the PEs in PELST over the PEAC.

The (NPE-1) PEs executing DECHLT decrement CTR and HALT. The PE executing DECSIG decrements CTR, waits until it is zero, sends a "DONE" signal to the CC and HALTS too.

3.2.8.2.2.7 PETDDNSG (PETLNO, CTR)

The effect of this is the same as PETDWAIT except that control returns immediately and an "INTRPT" signal is sent to the process when the "DONE" is finally received from a PE. This permits the CC program to continue to execute in parallel with the PEs under its control.

3.2.8.3 The PEAC Driver

The PEAC driver is a package of subroutines that allows FORTRAN programs to interface with the PE Array Controller (PEAC). See section 2.1.2 for a detailed description of the operation of the PEAC. The PEAC driver allows the user to program the PEAC at the lowest level. Normally, however, user-written programs would call higher-level system programs such as the Inter-PE Communication and Synchronization Monitor (Section 3.2.8.5), the Multi-Processor Loader (Section 3.2.8.4) and the PE Task Dispatcher (Section 3.2.8.2) which in turn call the PEAC driver.

3.2.8.3.1 PEAC Driver Routines

The following is a description of the PEAC driver routines. All parameters are 16-bit integers unless noted otherwise.

PEN refers to a PE number; PELST is a list of PEN's ending with -1. The PEAC driver routines check by a call to the PE allocation manager whether all the PEs specified are allocated to the user. Otherwise an error is indicated by returning a non-zero value.

3.2.8.3.1.1 DISCON (PEN)

PE number "PEN" is physically disconnected from the rest of the system. This routine is utilized by diagnostic/reconfiguration programs upon detecting a malfunctioning PE.

3.2.8.3.1.2 CON (PEN)

PE number "PEN" is connected to the rest of the system.

3.2.8.3.1.3 RUN (PEN, ADR)

PE number "PEN" is put into the 'RUN' mode, and directed to begin execution at address ADR.

3.2.8.3.1.4 HALT (PEN)

PE number "PEN" is put in the 'HALT' mode.

The PE suspends execution of its current programs and enters a mode where it is executing ODT (On-line Debugging Technique - see LSI-11 processer handbook, section 7.3, ODT/ Console Microcode). By transmitting appropriate strings of ASCII characters tc the PE, a program running on the CC can examine and modify any memory location or device register in the address space of the LSI-11, or any of the internal registers.

3.2.8.3.1.5 SEND (PEN, BUFAD, BUFLEN)

"BUFLEN" words starting at "BUFAD" are sent to PE number "PEN". The PE must accept each word within a certain "time-out" period by reading out of its CCID register. This will normally be done by the ODT microcode (if the PE is in HALT mode) or by the C.C. Communication handler within the PE resident monitor. If the resident monitor gets clobbered, or the PE microprocesser is stuck in a loop with interrupter masked off it can fail to respond. In this case an error is returned.

Control is returned to the calling routine after all the data has been transmitted.

3.2.8.3.1.6 BCAST (PELST, BUFAD, BUFLEN)

The operation of this is the same as for the SEND except that the data is broadcast to all the PEs in "PELST".

3.2.8.3.1.7 RECV (PEN, BUFAD, BUFLEN)

BUFLEN words from PE number PEN are returned into the area starting at BUFAD.

3.2.8.3.1.8 RECWDR (PEN, NAVAIL)

Returns one word from PE number PEN if available. Otherwise NAVAIL is set to 1. The routine returns immediately in either case.

3.2.8.3.1.9 PESIG (PEN, PROC)

An "INTRPT" signal is sent to the process PROC when an input is received from PE number PEN. (See the UNIX Programmer's Manual - SIG(II) and KILL(I), for a description of "signals").

3.2.8.3.1.10 LOADPE (PEN, BUFAD, BUFLEN, RELOC)

Loads BUFLEN words of data beginning at BUFAD into the memory of PE number PEN. The data must be in a format compatible with the PDP-11 absolute loader. This routine is useful for initial loading of a PE microprocessor. RELOC is an optional relocation value.

3.2.8.3.1.11 LOADBC (PELST, BUFAD, BUFLEN, RELOC)

The operation of this is the same as LOADPE except that all the PEs in PELST are loaded in broadcast mode.

3.2.8.3.1.12 RECLIN (PEN, BUFAD, BUFLEN, BRKCH, RCNT)

This routine is used to receive a variable length message from PE number PEN, as when communicating with ODT. The words up to the break character BRKCH are returned in the buffer beginning at BUFAD. The number of words actually received is given in RCNT. If more than BUFLEN words are received, the extra words are discarded and an error indication is returned. The calling program receives control only after a break character has been received.

3.2.8.4 Inter-PE Communication and Synchronization Monitor (IPECSM)

The Inter-PE Communication and Synchronization monitor provides facilities for flexible communication between the processing elements and the control computer and for synchronization among the routines running on the PEs and the CC.

The Central Working Storage (CWS), which is directly accessible to all the PEs, provides a generally powerful and efficient means for communications and synchronization. Routines are provided in the resident PE monitor as well as in the PE sub-task dispatcher which facilitate communication and synchronization via the Central Working Storage. These should be adequate for most programs. However, much greater flexibility is available through the Inter-PE Communication and Synchronization monitor which utilizes the direct communication hardware of the Processing Element Array Controller (PEAC). In particular, the PEAC supports interrupts between the CC and the PEs.

3.2.8.4.1 IPECSM Message Format

The format of messages transmitted or received under the control of the IPECSM is shown in Figure 3.2.8.4.1. The 6-bit fields "SRC" and "DEST" specify the processor generating the message and the target processor respectively. The values 0-31 refer to the corresponding Processing Element while a value of 63 refers to the Control Computer. For a message that includes data (format A), the 4-bit "type" field is zero. The next word specifies the length of the data and the message is followed by a checksum.

An alternative one-word format (format B) is utilized for transmitting one of 15 possible "signals" as specified by the non-zero "type" field. This format is normally used for synchronization-type applications where no data is involved. Some of the standard signals used by system routines are:

1. WAIT
2. PROCEED
3. START
4. DONE

The remaining signal numbers can be assigned any appropriate meaning by user programs.

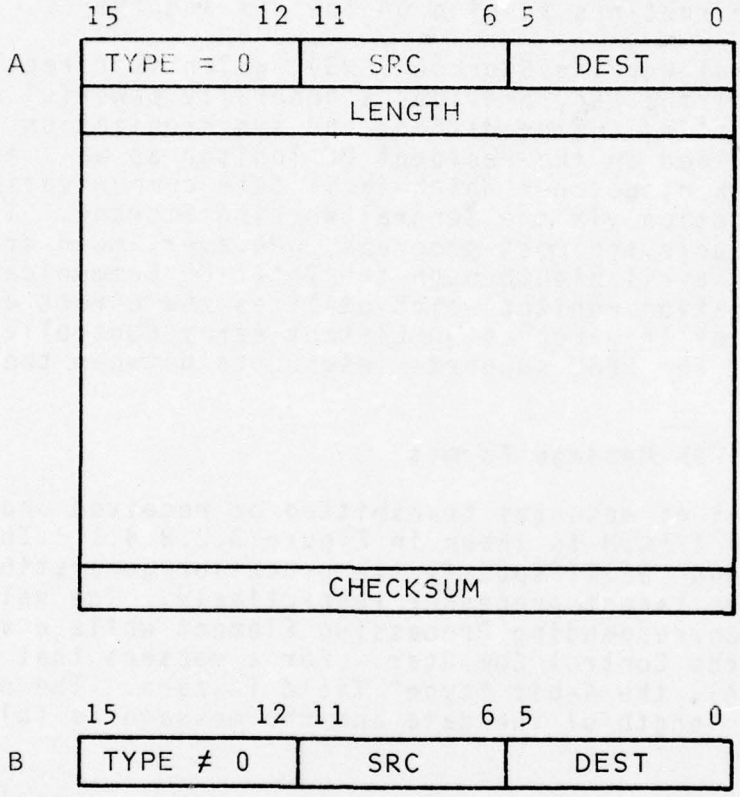


Figure 3.2.8.4-1. IPECSM Message Format

3.2.8.4.2 IPECSM Routines

The following routines of the IPECSM are available to user programs.

3.2.8.4.2.1 SETCOM (PELST)

The PEs specified in PELST are permitted to send messages to each other. The purpose here is to prevent malfunctioning PE hardware or software from sending random messages to other PEs and generating unexpected interrupts. SETCOM effectively creates a "firewall" around a subset of the PEs cooperating in performing a task. When a message header word is received from a PE within a group specified in a SETCOM, and addressed to another PE within the group, the header word and the subsequent words of the message if any are transmitted to the addressed PE by IPECSM.

3.2.8.4.2.2 SENDSIG (PEN, SIGNO)

Signal number SIGNO is sent to PE number PEN.

3.2.8.4.2.3 SENDMES (PEN, MESBUF, MESL)

The message consisting of MESL words beginning at MESBUF is sent to PE number PEN. An appropriate header and trailer (checksum) are also included. Control is returned to the calling program after the whole message has been sent.

SNMSBC (PELST, MESBUF, MESL)

3.2.8.4.2.4 RECSGR (PEN, SIGNO, PROC)

An "INTRPT" signal is sent to the process PROC when a signal is received from PE number PEN, addressed to the CC. The signal number is returned variable SIGNO (see the UNIX Programmer's Manual - SIG(II) and KILL(I)).

3.2.8.4.2.5 RECSIG (PEN, SIGNO)

The process is blocked until a signal is received from PE number PEN. The signal number is then returned in SIGNO.

3.2.8.4.2.6 RECMSR (PEN, BUFAD, LEN, PROC)

When a message from PE number PEN is received, addressed to the CC, it is stored beginning at BUFAD, the length is stored in LEN, and an "INTRPT" is sent to process PROC.

3.2.8.4.2.7 RECMSG (PEN, BUFAD, LEN)

The operation of this is similar to RECMES except that the calling process is blocked until the message is received.

3.2.8.4.2.8 EXEC (PEN, STADDR)

PE number PEN is halted and directed to begin execution at STADDR.

3.2.8.4.2.9 EXECBC (PELST, ADDR)

This is similar to EXEC except that all the PE's in PELST start execution at STADDR.

3.2.9 The Central Working Storage Manager (CWSM)

The CWSM manages the allocation of the CWS as a system-wide resource. It facilitates the concurrent execution of different computations that require space in the CWS and provides protection between them. It can also be used for dynamic allocation of storage within a single computation. The following FORTRAN-compatible functions are available to CC programs to request CWSM services.

3.2.9.1 EXBANK (I, PELST)

CWS memory bank no. I is assigned for exclusive use by the PE's specified by PELST, which is a vector containing PE numbers terminated with a -1. All the PEs in PELST must be allocated to the owner of the calling process.

If I is negative, any available bank is assigned. In either case, the number of the bank actually assigned is returned as the value of the function.

A negative value is returned indicating an error condition if any space in the requested memory bank is currently allocated to a PE not in PELST or if I is negative and no bank is available for assignment.

3.2.9.2 FRBANK (I)

Bank no. I which should have been assigned previously by an EXBANK call, is freed for use by any computation.

3.2.9.3 ALCBLK (SIZE, LOC, BNK)

A block of SIZE contiguous bytes is allocated to the caller, beginning at LOC or in memory bank BNK. Either LOC or BNK should be positive but not both. The address of the block is returned as the function value.

SIZE, LOC and the function ALCBLK are of type INTEGER *4 (32-bit Integer).

If both LOC and BNK are negative, the CWSM selects an available block of the requested size to allocate.

If size is zero, the largest available block satisfying any LOC or BNK specification is allocated.

A negative value is returned if the request made in the

ASSBLK cannot be satisfied.

Note that while a block of memory allocated using ALCBLK is guaranteed not to be allocated to anybody else, this does not prevent another computation from erroneously accessing it unless the memory bank (or banks) containing the block are assigned exclusively by an EXBANK call.

3.2.9.4 DEABLK (ADDR)

The block at ADDR, previously allocated by ALCBLK, is deallocated, and added to the pool of memory available for allocation. ADDR is a 32-bit integer. The function returns a negative value if a block at address ADDR was not assigned to the caller.

3.2.10 Mass Storage Access System (MSAS)

The Mass Storage Access System handles all transfers of data to and from the System Mass Storage. This differs from the CC storage in that it has access to the entire CWS (up to 16 Mbytes) while the normal disks on the CC can only transfer data in and out of the memory on the CC Unibus (up to 256 kbytes with memory management).

While the CC disks are used primarily for storage of programs, the System Mass Storage is used primarily for storage of large data files, which have to be moved in or out of CWS in the most efficient way possible. Thus, whereas the CC file manager, which is a part of the underlying UNIX system, is designed to maximum flexibility and programming convenience, the MSAS design emphasizes efficiency.

The files on the mass storage disks are contiguous and space has to be preallocated for all files. Further, the unit of storage allocation for a file is the track and the MSAS tries to keep a file in the smallest number of cylinders possible to minimize the head-positioning times required on going from one cylinder to the next. Thus, a file smaller than about 100 kbytes would normally be allocated storage within a single cylinder.

The directory system is relatively simple compared to the UNIX directory system. The file name of a file on a mass-storage disk consists of several names separated by slashes as in UNIX. The first name is of the form MSn where n is a digit corresponding to the drive the file is located on. The directory for a disk is stored on the disk itself in the first track, followed by tables listing free blocks of the disk.

3.2.10.1 MSAS Routines

The following functions are available to programs running on the CC to invoke the MSAS.

3.2.10.1.1 MCREAT (NAME, MODE, SIZE)

MCREAT creates a new file on the Mass Storage disk indicated by the prefix of NAME. NAME is a null-terminated ASCII string in the standard file-name format for the MSAS. MODE is the same as for UNIX files (see CHMOD(II), UNIX reference manual).

If the file already exists, it is set to be rewritten. Its mode and owner is unchanged.

SIZE is the space in tracks to be allocated to the new file. A file-descriptor, which is a small integer, is returned as the value of the function. This is used in subsequent operations on the file. A negative file-desriptor indicates an error.

3.2.10.1.2 MOPEN (NAME, MODE)

MOPEN opens the file specified by NAME for reading if mode is Ø, writing if mode is 1 or for both reading and writing if the mode is 2. NAME is a null-terminated string of ASCII characters representing the directory path name of the file.

A file-descriptor is returned, which is saved for use in subsequent operations on the file. A negative value indicates an error in attempting the MOPEN operation.

3.2.10.1.3 MCLOSE (fildes)

MCLOSE closes the file specified by the file-descriptor, which would be one returned previously by a MCREAT or MOPEN.

3.2.10.1.4 MUNLINK (NAME)

NAME is a null-terminated string of ASCII characters representing a directory path name. The corresponding directory entry is deleted. If this entry was the last link to the file, the file is deleted too.

3.2.10.1.5 MREAD (FILDES, BUF, NBYTES)

Fildes is a file descriptor returned by a successful MOPEN call. NBYTES bytes are input from the file into a buffer at BUF in CWS. The number of bytes actually transferred is returned as the value of the function.

A returned value of Ø indicates that the end of the file has been reached. A negative value indicates any other error. The function returns to the caller after the read operation is completed.

BUF, NBYTES and the function MREAD (i.e. the value returned) are of type INTEGER*4 (32-bit integer).

3.2.10.1.6 MREADR (FILDES, BUF NBYTES)

The operation of this function is the same as MREAD except that the function returns control to the caller immediately. The function MWAIT (Sec. 3.2.10.1.11) can be used to determine when the operation is completed.

3.2.10.1.7 MREADF (FILDES, BUF, NBYTES, FUNC)

The operation of this function is the same as MREAD except that the function returns to its caller immediately, and the function FUNC is executed when the operation is completed.

3.2.10.1.8 MWRITE (FILDES, BUF, NBYTES)

FILDES is a file descriptor returned from a successful MOPEN or MCREAT call. NBYTES bytes are written onto the file starting at BUF in CWS. The numbers of bytes actually transferred is returned as the function value. A negative returned value indicates an error. The function returns to the calling program after the write operation is completed.

BUF, NBYTES and the function MWRITE itself (i.e., the returned value) are of type INTEGER*4 (32-bit integer).

3.2.10.1.9 MWRITR (FILDES, BUF, NBYTES)

The operation of this function is the same as MWRITE except that control returns to the caller immediately. The function MWAIT (Sec. 3.2.10.1.11) can be used to determine when the operation is completed.

3.2.10.1.10 MWRITE (FILDES, BUF, NBYTES)

The operation of this function is the same as MWRITE except the function returns to its caller immediately and the function FUNC is executed when the write operation is completed.

3.2.10.1.11 MWAIT (FILDES)

MWAIT suspends execution until all read/write operations on the specified file have been completed. It is used in conjunction with MREADR and MWRITR.

3.2.10.1.12 MSEEK (FILDES, OFFSET, MODE)

FILDES refers to a file opened for reading or writing. The read or write pointer is set as follows:

if MODE is Ø, the pointer is set to OFFSET.

if MODE is 1, the pointer is set to its current location plus OFFSET.

if MODE is 2, the pointer is set to the current end of data in the file plus OFFSET.

The value of the pointer is used as the starting file address for a subsequent read or write.

3.2.10.1.13 MRDC (FILDES, BUF, NBYTES)
MRDR (FILDES, BUF, NBYTES)
MRDF (FILDES, BUF, NBYTES, FUNC)
MWRT (FILDES, BUF, NBYTES)
MWRTR (FILDES, BUF, NBYTES)
MWRTF (FILDES, BUF, NBYTES, FUNC)

The operation of these functions is identical to MREAD, MREADR, MREADF, MWRITE, MWRITR and MWRITF respectively except that the CC memory is used instead of CWS. A Mass Storage disk can transfer data to/from CWS or CC memory depending on the setting of the MODE bit in the Control and Status register (see Sec. 2.5).

3.2.11 PE/CWS Simulator

The PE/CWS simulator creates "virtual PEs" and "virtual CWS" which interface with CC programs in the same way as actual PEs and the CWS. In addition, facilities are provided for setting breakpoints and examining and modifying the contents of virtual PE registers and memories or the virtual CWS.

The simulator makes it very convenient to debug programs on the CC while all the PEs may be busy on a production computation. It is also possible to do program development on any other PDP-11 system with adequate disk storage. Debugging a program on a multiprocessor system can be very much harder than on a single processor. In the G-471, in particular, a PE is itself a multiprocessor with its LSI-11 microprocessor, IOAG, CAPU, DFU, DSU, FIFOs and channel all operating in parallel, which can make the problem much more complex. The simulator makes it possible to freeze the state of all parts of the virtual PEs at a breakpoint and examine the relevant registers, flags or memories. It can, therefore, be a real boon for program debugging.

The simulator maintains tables representing the state (internal registers, etc.) for each piece of hardware to be simulated. Memories are maintained on disk with relevant parts "paged" in and out of buffer areas in memory. The simulator interprets instructions for each processor using the virtual registers maintained in its tables. A separate process is created for each hardware unit being simulated that can operate in parallel. In this way, the multi-tasking facilities of the operating system are utilized to interleave these operations.

3.2.12 The CAPU Cross-Assembler

3.2.12.1 Introduction to CAPU Cross-Assembler

This module translates programs written in assembly language into the machine language of the CAPU. It is intended to be used for programming algorithms not available in the Standard Subroutine Library (SSL), or where it is considered desirable to program at the lowest level in order to get the maximum throughput that a clever programmer can squeeze out of the hardware. The G-471 architecture makes it possible to achieve very high processing rates through parallelism and pipelining at various levels, but it may not always be possible to utilize the hardware in the most optimum manner when using pre-packaged subroutines. The CAPU cross-assembler is also used, of course, to initially prepare the routines for the SSL and to add new routines to it.

The CAPU cross-assembler runs on the CC (or other PDP-11 systems). The input to this module is a standard ASCII file, while the output is a binary file in a format compatible with the PE loader. A listing file is also produced upon request.

3.2.12.2 CAPU Cross-Assembler Input Format

The format of the input file for the CAPU assembler (i.e., the CAPU assembly language) is described next.

The following conventions are used in describing the format of an assembly language statement:

a) Underlined characters - Upper-case alphabetics and other characters that have to appear exactly as shown (corresponds to terminals in BNF)

b) A word in lower case - must be replaced by one of several possible strings (this corresponds to a BNF non-terminal)

c) Square brackets [. . .]. The part within the square brackets is optional

d) Braces { . . . }. The part within the braces may be repeated any number of times (including zero)

e) Stacked items - where several items are stacked vertically between bars, they represent alternatives; any one is to be specified.

Each statement of the source file must be of the form shown below.

```
stmt → [label:] |      instn      | [; comment]
                | vbl   =   instn |
                | .     =   instn |
```

"label" and "vbl" can be any alphanumeric string beginning with an alphabetic character. If the "label" field is present, the current value of the location counter (LC) is assigned to "label". The "comment" is any string between the semicolon and the end of line and is ignored.

The first alternative for the main part of the statement represents a normal instruction that generates a 64-bit word of object code as specified by "instn".

The second alternative computes the value of "instn" and assigns it to "vbl" without generating any code.

The third alternative simply sets the LC to the value of "instn".

The "instn" part consists of one or more "terms" separated by operators from the list shown below.

```
instn → term {op term}

op    → | +  |  (add)
        | -  |  (subtract)
        | *  |  (multiply)
        | /  |  (divide)
        | &  |  (AND)
        | !  |  (OR)
```

All evaluations are done left-to-right on 64-bit signed integers (2's complement).

Each "term" can be one of the following:

a) The current value of the location counter

.

b) A decimal constant

D'n' or n

where "n" is a string of digits Ø through 9

c) A hexadecimal constant

X'n'

where n is a string of digits Ø through 9 or A through F

d) A binary constant

B'n'

where n is a string of Ø's and 1's

e) A label or variable

f) one or more mnemonics separated by commas

mnem {,mnem}

Each mnemonic field "mnem" specifies the bit pattern for some part of the 64-bit word to be generated which will specify the operation of some hardware module within the CAPU at execution time. The assembler checks to see that all the mnemonics within a "term" specify unique parts of the word, (except for certain exceptional cases).

e.g. MUL(M1 M3), ADD(A1 A2), M5 → R3

Mnemonics

The following is a list of valid mnemonics that can be specified in the "mnem" field of the source statement, along with a brief description of the resulting instruction coded. For a more complete description of the CAPU operation and its instruction format, see Section 3.2.14.4.1.

(i) MVA (src dest)

A transfer on bus A from "src" to "dest" is coded in bits 47 to 40.

The valid values of "src" and "dest" are given below along with their meaning and the code generated:

Bus A Sources ("src")

Mnem	Code (Hex) (bits 47-44)	Description
ZER	Ø	Zero - destination is cleared
QIL	1	Queued Input Low - lower order 32 bits of Input Data FIFO
QIH	2	Queued Input high - higher order 32 bits of Input Data FIFO
DML	3	Direct Memory Low - lower order 32 bits of data from DFU
DMH	4	Direct Memory high - higher order 32 bits of data from DFU
SFT	5	Shifter output - inputs to shifter are specified by the RFAA and RFAB fields
ALU	6	Output of ALU - inputs to ALU are same as to shifter (above)
CON	7	Constant Register
M5	8	Output of Multiplier
M6	9	Lower order part of multiplier output
A5	A	Output of Adder

RFA	E	Register File - A output (register number specified by RFAA)
RFB	F	Register File - B output (register number specified by RFAB)
Rn	E or F	A general purpose register; n is a decimal number between Ø and 15. See detailed description below

Bus A Destinations ("dest")

Mnem	Code (Hex) bits 43-40	Description
M1	1	Multiplier Input register 1
M2	2	Multiplier Input register 2
M3	3	Multiplier Input register 3
M4	4	Multiplier Input register 4
DMA	8	Direct Memory Address to Data Fetch Unit
A1	9	Adder Input register 1
A2	A	Adder Input register 2
A3	B	Adder Input register 3
A4	C	Adder Input register 4
QOL	D	Queued Output Low - Lower order 32 bits to Output Data FIFO
QOH	E	Queued Output High - Higher order 32 bits to Output Data FIFO
RFA	F	Register File - even numbered register specified by RFAA
Rn	F	General purpose register; n is an even decimal number between Ø and 14. See detailed description below.

General Purpose Registers with Bus A

The mnemonics RØ through R15 in the "src" or "dest" fields specify the register file. In addition, the register address bits are also set.

When the "dest" is a register (Rn), n must be even. The Register File Address A (RFAA) is set to n and the destination field is set to F for register file. When the "src" is of the form Rn, the source field is set to F (for RFB) and the Register file Address B (RFAB) is set to n. (However, if the RFAB field has already been set by a previous mnemonic, and the "dest" is not a general register, the assembler will try to use the A port of the register file. It will set the source to E (for RFA) and RFAA to n.)

(ii) MVB(src dest)

A transfer on bus B is coded in bits 39 through 32. The operation of the assembler for this mnemonic and the valid sources and destinations are very similar to the MVA case. We will only describe the differences here.

The "src" and "dest" mnemonics are the same as for bus A except that RFA as "dest" corresponds to odd numbered registers.

General Purpose Registers with Bus B

When the "src" or "dest" fields of a Bus B transfer specification are of the form Rn (n = 0 - 15), the operation of the assembler is similar to the Bus A case except that when the dest is Rn, n must be odd.

If transfers are specified on both buses with G-P registers as destinations, then the "dest" fields must be of the form Rn and R(n + 1) for bus A and B respectively where n is even. I.e., it is possible to write into two registers in the same cycle if they are an adjacent pair. This is specially useful for transfering complex numbers in and out of the CAPU.

e.g. MVA(IQL R2), MVB(IQH R3) or
IQL > R2, IQH > R3

(iii) MOV(src dest)
or src > dest

This is treated exactly like an MVA (src dest) if the following are satisfied:

a) a bus A transfer has not been specified already within the term being evaluated either explicitly or implicitly

b) "src" and "dest" are valid for bus A.

If any of the above conditions is not satisfied, the mnemonic is treated like a

MVB (src dest).

e.g. MOV(M5 M2)

M5 > M2

Immediate Data

If the "src" field of a MOV, MVA or MVB contains a "#" followed by a constant or a variable that has previously been defined, its value is inserted into bits Ø to 31 of the generated word, bit 63 is set to indicate an immediate instruction, and the "src" field is set to "7" (for CON).

e.g. MVA(#X'FFFFØØ3F' R3)
#FACTOR > A3

Note: The constant is latched into the CON register and can be reused later by specifying CON as the source of a MOV, MVA or MVB instruction. It is also possible to load the CON register without transferring the constant anywhere in the same cycle by using the following

LC! value

when LC has been set to X'8ØØØ ØØØØ ØØØØ ØØØØ' and "value" is the desired immediate data.

(iv) MUL(Mm Mn) when m = 1,2,5
n = 3,4,5

e.g. MUL(M1 M3)

This generates code to initiate a floating point multiply with Mm and Mn as inputs in bits 39 through 34.

(v) IMUL (Mm Mn)

Same as MUL except that a fixed point multiplication is specified by a 1 in bit 39.

(vi) ADD (Am An) where m = 1,2,5
n = 3,4,5

e.g. ADD(A2 A5)

This generates code to initiate a floating point add with Am, An as inputs in bits 45 through 40.

(vii) SUB(Am An)

Same as ADD except that a floating point subtract is specified by a 1 in bit 41. An is subtracted from Am.

(viii) ALU Instructions.

The ALU (arithmetic and Logic Unit) performs integer arithmetic and bitwise logical operations on the contents of one or two general purpose registers. The data format is 32-bit integer in 2's complement representation. The result of the operation can be stored in one of the input registers or transferred to any other destination on one of the two CAPU buses in the same cycle. The ALU condition flags AZ and AN, which denote zero or negative ALU output, can be set optionally by specifying the mnemonic

CL

The condition flags can be used by a subsequent conditional branch (BC) or subroutine (BCS). Bit 3 of the object code word is set by the "CL" mnemonic.

A mnemonic specifying an ALU operation codes the ALU function in bits 8 through 4. In addition the RFAA field (bits 18-15) is coded for single operand operations while both the RFAA field and the RFAB field (bits 14-11) are coded for double operand operations.

Single Operand Operations

The general format is

op (Rn) [>[dest]] [,CL]

The valid mnemonics "op" and the corresponding functions are

op	code (bits 8-4)	function
INC	ØØØØØ	add 1
DEC	1111Ø	subtract 1
COM	ØØØØ1	complement (bitwise)
ASL	11ØØØ	arithmetic shift left one position

If a destination is specified by including the "> dest", the computed result is stored in "dest". The assembler operates as though the mnemonic MOV(ALU dest) were specified in addition to the ALU operation, and generates code for the corresponding bus transfer. If the "dest" is a general purpose register of the form Rn, it must be identical to the first operand. In this case, it may be omitted but the ">" is required.

examples:

INC(R3), CL

(R3 + 1 is computed and the condition flags are set accordingly. R3 itself is not changed).

DEC(RØ)> RØ or

DEC(RØ)>

(RØ is decremented by 1. The condition flags are not changed).

COM(R12)> A1, CL

(The complement of R12 is loaded into A1, and the condition flags are set corresponding to the new contents of A1. R12 is not modified.)

Note: Since single operand ALU operations code the RFAA field, it is not possible to have a transfer into a G.P. register simultaneously but it is possible to transfer out of a G.P. register in the same instruction.

e.g. DEC(RØ) > RØ, R3 > M3

Double Operand Operations

The general format is

op(Rm Rn) [>[dest]] [,CL]

The valid mnemonics "op" and the corresponding operations are

op	code (bits 8-4)	function	
IADD	10010	integer addition	(Rm + Rn)
ISUB	01100	integer subtraction	(Rm - Rn)
AND	10111	bitwise AND	(Rm Rn)
OR	11101	bitwise OR	(Rm v Rn)
XOR	01101	bitwise exclusive OR	(Rm + Rn)
NAND	01001	bitwise NAND	(~(Rm Rn))
NOR	00011	bitwise NOR	(~(Rm v Rn))

If a ">" or "> dest" is specified, the operation is the same as for single operand instructions and will not be repeated here.

examples:

ISUB (R1 R2), CL

(computes the difference R1 - R2 and sets the condition flags accordingly. R1, R2 are unchanged.)

IADD (RØ R4) > RØ or

IADD (RØ R4) >

(R4 is added to RØ. R4 and the condition flags are unchanged.)

AND (RØ R15) > QOL

(A bit-by-bit logical AND operation is performed on the contents of RØ and R15 and the result is transferred to the lower-order part of the output Data

FIFO. RØ, R15 and the condition flags are unchanged.)

(ix) Shifting Operations

The general format is

SHIFT(Rm Rn) > [dest]

The contents of the register Rm are shifted to the left. The number of positions to shift is given by the contents of Rn modulo 32. The bits shifted out of the left of Rm are fed back at the right. The shifted result is loaded into the register specified by "dest". If "dest" is omitted, Rm is assumed. Further, if a G.P. register is specified for "dest", it must be Rm.

The assembler operation for a SHIFT consists of simulating a "MOV(SFT dest)" and setting the RFAA and RFAB fields to m and n respectively. If "dest" is omitted or is identical to Rm, then "dest" is considered equivalent to an RFA.

(x) Unconditional Branch

The general format is

BR locn

"locn" can be any valid "instn" but usually it would be either a label or an expression of the form ". + num" or ". - num" where "num" is a decimal or hexadecimal number. An unconditional transfer of control to "locn" takes place after executing the current instruction. The current value of the LC is subtracted from the value computed for "locn" and the result modulo 256 is inserted into the DISP field (bits 55 through 48). In addition, TYPE (bits 62-61) is set to 11 and the COND field (bits 59-56) is cleared.

An error message is produced if ("locn" - current LC) is outside the range (-128 to +127).

(xi) Conditional Branch

The general format is

BC(con) locn

A conditional transfer of control to "locn" takes place after executing the current instruction if the condition specified by "con" is true. Otherwise the next instruction is executed.

The DISP and TYPE fields are set as for a BR. The COND field (bits 59-56) is set according to "con". The valid values for "con" are:

"con"	code (hex) (bits 59-56)	condition
AZ	1	flag AZ is set (ALU output = Ø)
ANZ	9	flag AZ is clear (ALU output ≠ Ø)
AP	2	flag AN is clear (ALU output is positive)
AN	A	flag AN is set (ALU output is negative)
A5P	3	content of A5 is positive
A5N	B	content of A5 is negative
FLA	4	flag A is set
FLB	5	flag B is set
FLC	6	flag C is set
FLD	7	flag D is set

(xii) Subroutines

The mnemonic "PUSH" used along with an appropriate branch mnemonic will result in pushing the current program counter onto the subroutine stack when the instruction is executed. The assembler codes a 1 in the PUSH field (bit 60). The mnemonics BRS and BCS used in place of BR and BC respectively produce the same result.

e.g. BR SUB1, PUSH; execute subroutine at SUB1

BRS SUB1; execute subroutine at SUB1

BCS (FLA) SUB2; execute subroutine SUB2 if flag A is set

A return from a subroutine is specified by the mnemonic

POP

which sets bit 60 (the POP field) in the generated object word.

(xiii) SET(flags)

"flags" is a string of one or more letters from A, B, C, D. The assembler codes B'0101' in bits 43 to 40 and the flags specified are coded in bits 47 to 44. Since MVA also sets bits 47 to 40, transfers on bus A cannot be specified in the same instruction.

e.g. SET(AC); set flags A and C.

(xiv) CLR(flags)

"flags" is a string of one or more letters from A, B, C, D. The assembler codes B'0101' in bits 35 to 32 and the flags are specified in bits 39 to 36. Since MVB also sets bits 39 to 32, transfers on bus B cannot be specified in the same instruction.

e.g. CLR(BC); clear flags B and C

(xv) NOP

This generates a no-op consisting of all Ø's.

SAMPLE CAPU PROGRAM

```
;  PROGRAM TO PERFORM BUTTERFLY OPERATION FOR FFT
;
;  THIS PROGRAM PERFORMS THE BUTTERFLY OPERATION:
;
;            C = A + B ⁜ W
;            D = A - B ⁜ W
;
;  WHERE A, B, W, C, D ARE COMPLEX FLOATING POINT NUMBERS,
;  8 BYTES EACH.  THE COUNT N IS A 4 BYTE INTEGER
;
;  INPUT SEQUENCE:  N, A(1), B(1), W(1), A(2), B(2), W(2),
;  A(3), B(3), W(3) ..... A(N), B(N), W(N)
;  OUTPUT SEQUENCE:  C(1), D(1), C(2), D(2) ... C(N), D(N)
;  TWO EXTRA SETS OF DUMMY INPUTS ARE ALSO REQUIRED:  A(N+1),
;  B(N+1) ... B(N+2), W(N+2)
;  EACH BUTTERFLY TAKES THREE CYCLES OF THE
;  MAIN LOOP (BEGINNING AT LOOP) TO COMPUTE.  IN
;  CYCLE A, THE INPUTS A, B AND W ARE READ IN.  IN
;  CYCLE B THE FOUR REAL PRODUCTS REQUIRED FOR
;  B ⁜ W ARE COMPUTED AS WELL AS THE ADDITIONS
;  TO COMPUTE THE REAL PARTS OF C & D.  IN
;  CYCLE C, THE IMAGING PARTS OF C AND D ARE COMPUTED
;  AND THE OUTPUTS C & D ARE SENT TO THE
;  OUTPUT DATA FIFO.  THE COMMENTS WITHIN THE
;  MAIN LOOP ARE PREFIXED WITH A:, B: OR C:
;  TO IDENTIFY THE CYCLE BEING REFERRED TO.
;  REGISTER ASSIGNMENTS:
;  RØ - LOOP COUNTER; COUNT DOWN FROM N
;  R1 - TEMPORARY STORAGE FOR B I (IMAGINARY PART OF B)
;  R2 - TEMPORARY STORAGE FOR D R
;  R3 - TEMPORARY STORAGE FOR D I
;  R4 - TEMPORARY STORAGE FOR A R
;  R5 - TEMPORARY STORAGE FOR A I
;  FIRST PERFORM CYCLE A & B FOR FIRST SET OF INPUTS
;  AND CYCLE A FOR SECOND SET OF INPUTS
```

```
        IQL > RØ                                        ;  INPUT N
        IQL > A2, IQH > A4                              ;  INPUT A(1)
        IQL > M1, IQH > M2                              ;  INPUT B(1)
        IQL > M3, IQH > M4                              ;  INPUT W(1)
        MUL(M1 M2)                                      ;  BR(1) * WR(1)
        NOP
        NOP
        MUL(M2 M4), M5 > A1                             ;  BI(1) * WI(1), A1=BR * WR
        NOP
        NOP
        MUL(M1 M4), M5 > A3                             ;  BR(1)*WI(1), A3=BI*WI
        SUB(A1 A3)                                      ;  BR*WR - BI*WI
        IQL > R4, IQH > R5                              ;  INPUT A(2)
        MUL(M2 M3), ADD (A2 A5), MS > A1, AS > A3;  A1 = BR*WI
                                                        ;  A3=BR*WR - BI X WI
        IQL > R4, IQH > R1                              ;  INPUT B(2)
        IQL > M3, IQH > M4, SUB(A2 A3)                  ;  INPUT W(2), COMPUTE AR-(BR*CR-BI*CI)
;  NOW ENTER MAIN PART OF LOOP.
;
LOOP;   MUL(M2 M3), M5 > A3, A5 > R2                    ;  B: MULT BR & WR/C: A3 = BI*WR, R2=CR
        ADD (A1 A3), A5 > R3                            ;  C: ADD BR*WI & BI*WR, RI=DR
        R1 > M2, R4 > A2                                ;  B: M2=BI, A2=AR
        MUL(M2 M4), ADD (A4 A5), M5>A1, A5>A3           ;  B:MULT BI & WI, A1=BR*WR/C:
                                                        ;  ADD AI & (BR*WI + BI*WR),
                                                        ;  A3 = BR*WI + BI*WR
        DEC(RØ) > RØ, CL                                ;  C: COUNT DOWN FOR LOOP
        SUB (A4 A1), R2 > OQL, A5 > OOH                 ;  C:COMPUTE AI-(BR*WI + BI*WR),
                                                        ;  OUTPUT C FROM R2, A5
        MUL (M1 M4), M5>A3, R5>A4                       ;  B: COMPUTE BR*WI, A3=BI*WI, A4=AI
        SUB (A1 A3), R3>OQL, A5>OOH                     ;  B: COMPUTE B$*WR-BI*WR C: OUTPUT D
                                                        ;  FROM R3, A5
        IQL > R4, IQH > R5                              ;  A: INPUT A INTO R4, R5
        MUL (M2 M3), ADD(A2 A5), M5>A1, A5>A3           ;  B: MULT BI & WR, COMPUTE AR + (BR*WR -
                                                        ;  BI*CI), A1=BR*CI, A3=BR*WR - BI*WI
        IQL > M1, IQH > R1                              ;  A: INPUT B INTO M1, R1
        IQL > M3, IQH > M4, SUB(A2 A3), BC(ANZ) LOOP;
                                                        ;  A: INPUT W INTO M3, M4/B: COMPUTE AR -
                                                        ;  (BR*CR - BI*CI)/C: REPEAT LOOP UNTIL
                                                        ;  RØ IS ZERO
```

```
;  LOOP EXECUTED N TIMES.  SET FLAG TO SIGNAL
;  MICROPROCESSOR AND WAIT

   SET (A)                                          ;  SET FLAG A
WAIT:     WAIT IN LOOP
```

3.2.13 The IOAG Cross Assembler

3.2.13.1 Introduction to IOAG Cross Assembler

This module translates programs written in assembly language into the machine language programs of the IOAG. It is intended to be used for programming algorithms not available in the Standard Subroutine Library (SSL), or where it is considered desirable to program at the lowest level to get the maximum possible speed for a particular problem. The G-471 architecture makes it possible to achieve extremely high processing rates through parallelism and pipelining at various levels, but it may not always be possible to fully exploit the potential of the hardware using pre-packaged subroutines. The IOAG cross-assembler is also used to initially prepare the routines for the SSL and to add new routines to it.

The CAPU cross-assembler runs on the CC (or other PDP-11 systems). The input to this module is a standard ASCII file, while the output is a binary file in a format compatible with the PE loader. A listing file is also produced upon request.

3.2.13.2 Input Format

The format of the input file for the IOAG cross assembler is described next. See Section 3.2.12.2 for conventions used in describing the format of an assembly language statement.

Each statement of the source file must be of the form shown below.

stmt → [label:]		instn		[; comment]
	vbl	=	instn	
	.	=	instn	

"label" and "vbl" can be any alphanumeric string beginning with an alphabetic character. If the "label" field is present, the current value of the location counter (LC) is assigned to "label". The "comment" is any string between the semicolon and the end of line and is ignored.

The first alternative for the main part of the statement represents a normal instruction that generates a 64-bit word of object code as specified by "instn".

The second alternative computes the value of "instn" and assigns it to "vbl" without generating any code.

The third alternative simply sets the LC to the value of "instn".

The "instn" part consists of one or more "terms" separated by operators from the list shown below.

```
instn → term {op term}

op    → | + |  (add)
        | - |  (subtract)
        | * |  (multiply)
        | / |  (divide)
        | & |  (AND)
        | ! |  (OR)
```

All evaluations are done left-to-right on 64-bit signed integers (2's complement).

Each "term" can be one of the following:

a) The current value of the location counter

   .

b) A decimal constant

   D'n' or n

   where "n" is a string of digits Ø through 9

c) A hexadecimal constant

   X'n'

   where n is a string of digits Ø through 9 or A through F

d) A binary constant

   B'n'

   where n is a string of Ø's and 1's

e) A label or variable

f) one or more mnemonics separated by commas

mnem {,mnem}

Each mnemonic field "mnem" specifies the bit pattern for some part of the 64-bit word to be generated which will specify the operation of some hardware module within the CAPU at execution time. The assembler checks to see that all the mnemonics within a "term" specify unique parts of the word, (except for certain exceptional cases).

Mnemonics

The following is a list of valid mnemonics that can be specified in the "mnem" field of the source statement, along with a brief description of the resulting instruction coded by the assembler. For a detailed description of the operation of the IOAG and its instruction repertoire, see Section 2.2.1.7.2.1.

(i) MOV(src dest [dest dest])

A bus transfer is coded in bits 47 through 43. One source ("src") and one to three destinations ("dest") are specified. The valid sources and destinations are given below along with their meaning and the code generated.

Sources (bits 47, 46)

Mnem	Code (binary)	Description
ALU	00	Output of the ALU. The inputs are specified by the RFAA and RFAB fields.
CON	01	Constant Register. This holds immediate data.
SFT	10	Output of the shifter. The data and control inputs to the shifter are specified by the RFAA and RFAB fields.
REV	11	Bit Reversal: the contents of the register specified by the RFAA field, with the bits reversed (bit i = bit (23-i)

Destinations (bits 45 - 43)

Mnem	Code (binary)	Description
RFL	100	Register File. The particular register to be loaded is specified by the RFAA field.
IAF	010	Input Address FIFO. The mode bits are specified by the MODE mnemonic.
OAF	001	Output Address FIFO. The mode bits are specified by the MODE mnemonic.

Example:

```
MOV(CON IAF OAF); send constant to both FIFOs.
```

General Purpose Registers with MOV

The mnemonics RØ through R31 refer to general purpose registers in the Register file, and are valid also as "src" or "dest".

If a G.P. register is used as a destination, bit 45 is set as though the destination had been RFL. In addition, the RFAA field is coded to select the appropriate register.

Example:

```
MOV(CON R5); load constant into Reg. 5
```

If a G.P. register is used as a source, the bus source field (bits 47-46) is set as though ALU was specified as the source. In addition, the ALU function field is set so that the ALU produces the B input at its output, and the RFAB field is set to select the specified register. See Figure 3.2.13.2-1 for the fields in the instruction word.

Example:

```
MOV (R1 R5); R5 = R1
```

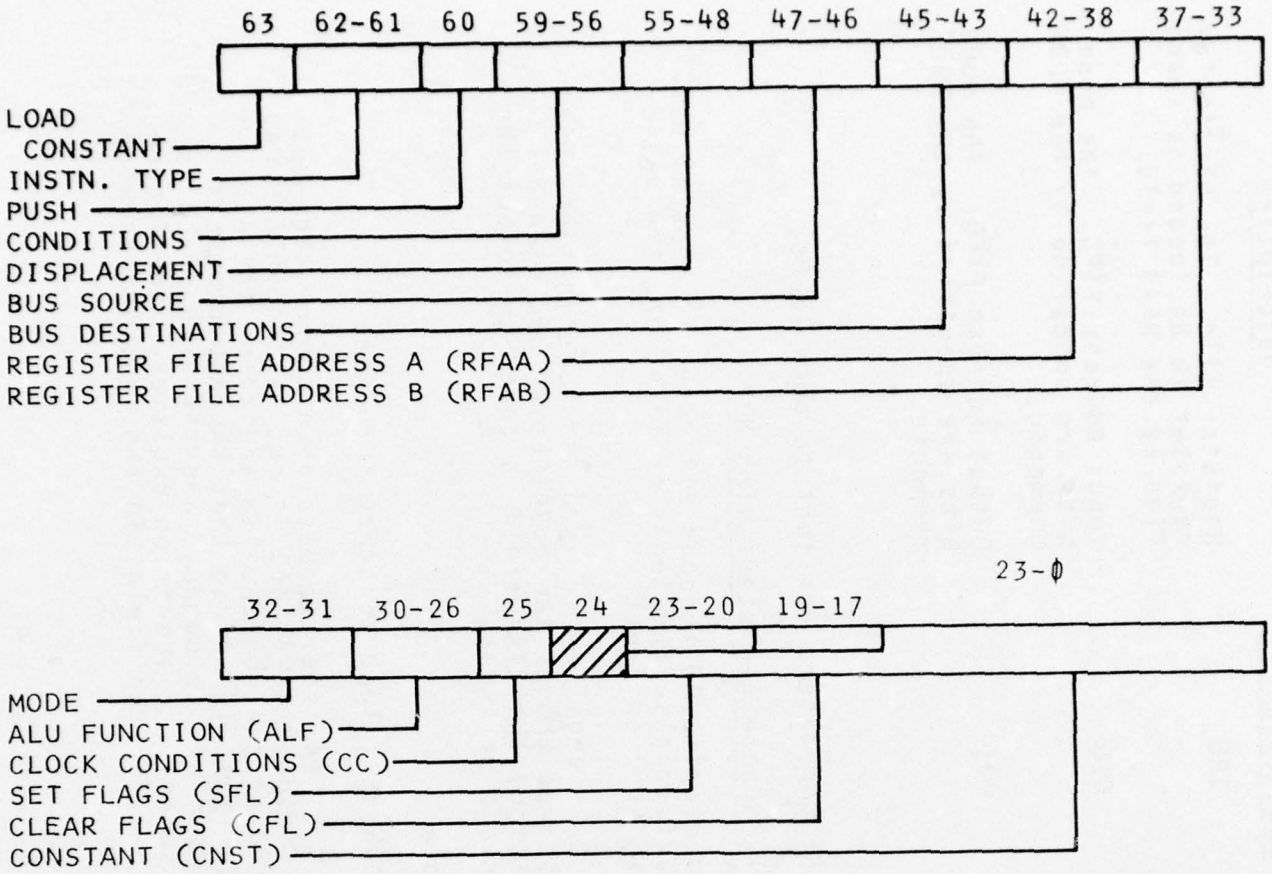


Figure 3.2.13.2-1. IOAG Instruction Format

Immediate Data

If the "src" field of a MOV contains a "#" followed by a constant or a variable that has previously been defined, its value is inserted into bits Ø to 31 of the generated word, bit 63 is set to indicate an immediate instruction, and the "src" field is set to "01" (for CON).

e.g. MVA(#X'FFFFØØ3F' R3)

Note: The constant is latched into the CON register and can be reused later by specifying CON as the source of a MOV instruction. It is also possible to load the CON register without transferring the constant anywhere in the same cycle by using the following

LC! value

when LC has been set to X'8ØØØ ØØØØ ØØØØ ØØØØ' and "value" is the desired immediate data.

(ii) ALU Instructions.

The ALU (arithmetic and Logic Unit) performs integer arithmetic and bitwise logical operations on the contents of one or two general-purpose registers. The data format is 24-bit integer in 2's complement representation. The result of the operation can be stored in one of the input registers or transferred to any other destination on the IOAG bus in the same cycle. The ALU condition flags AZ and AN, which denote zero or negative ALU output, can be set optionally by specifying the mnemonic

CL

The condition flags can be used by a subsequent conditional branch (BC) or subroutine (BCS). Bit 25 of the object code word is set by the "CL" mnemonic.

A mnemonic specifying an ALU operation codes the ALU function in bits 30 through 26. In addition the RFAA field (bits 42-38) is coded for single operand operations while both the RFAA field and the RFAB field (bits 37-33) are coded for double operand operations.

Single Operand Operations

The general format is

op (Rn) [>{dest}] [,CL]

The valid mnemonics "op" and the corresponding functions are

op	code (bits 30-26	function
INC	ØØØØØ	Add 1
DEC	1111Ø	Subtract 1
COM	ØØØØ1	complement (bitwise)
ASL	11ØØØ	arithmetic shift left one position

If one or more destinations are specified by including the "> dest", the computed result is stored in the destinations. The assembler operates as though the mnemonic MOV(ALU dest) were specified in addition to the ALU operation, and generates code for the corresponding bus transfer. If one of the "dest" is a general purpose register of the form Rn, it must be identical to the first operand. In this case, it may be omitted but the ">" is required.

examples:

INC(R3), CL

(R3 + 1 is computed and the condition flags are set accordingly. R3 itself is not changed).

DEC(RØ)> RØ or

DEC(RØ)>

(RØ is decremented by 1. The condition flags are not changed).

COM(R12)> IAF OAF, CL

(The complement of R12 is loaded into both FIFOs and the condition flags are set accordingly. R12 is not modified).

Note: Since single operand ALU operations code the RFAA field, it is not possible to have a transfer into a G.P. register simultaneously but it is possible to transfer out of a G.P. register in the same instruction.

e.g. DEC(RØ) > RØ, MOV (R3 OAF)

Double Operand Operations

The general format is

op(Rm Rn) [>[dest]] [,CL]

The valid mnemonics "op" and the corresponding operations are

op	code (bits 8-4)	function	
IADD	10010	integer addition	(Rm + Rn)
ISUB	01100	integer subtraction	(Rm - Rn)
AND	10111	bitwise AND	(Rm ∧ Rn)
OR	11101	bitwise OR	(Rm v Rn)
XOR	01101	bitwise exclusive OR	(Rm + Rn)
NAND	01001	bitwise NAND	(∼(Rm ∧Rn))
NOR	00011	bitwise NOR	(∼(Rm v Rn))

If a ">" or "> dest" is specified, the operation is the same as for single operand instructions and will not be repeated here.

examples:

ISUB (R1 R2), CL

(computes the difference R1 - R2 and sets the condition flags accordingly. R1, R2 are unchanged.)

IADD (RØ R4) > RØ or

IADD (RØ R4) >

(R4 is addes to RØ. R4 and the condition flags are unchanged.)

AND (RØ R15) > IAF

(A bit-by-bit logical and operation is performed on the contents of RØ and R15 and the result is transferred to the Input Address FIFO. RØ, R15 and the condition flags are unchanged.)

(iii) Shifting Operations

The general format is

SHIFT(Rm Rn) >[dest]

The contents of the register Rm are shifted to the left. The number of positions to shift is given by the contents of Rn modulo 24. The bits shifted out of the left of Rm are fed back at the right. The shifted result is loaded into the register specified by "dest". If "dest" is omitted, Rm is assumed. Further, if a G.P. register is specified for "dest", it must be Rm.

The assembler operation for a SHIFT consists of simulating a "MOV(SFT dest)" and setting the RFAA and RFAB fields to m and n respectively. If "dest" is omitted or is identical to Rm, then "dest" is considered equivalent to an RFA.

(iv) Unconditional Branch

The general format is

BR locn

"locn" can be any valid "instn" but usually it would be either a label or an expression of the form ". + num" or ". - num" where "num" is a decimal or hexadecimal number. An unconditional transfer of control to "locn" takes place after executing the current instruction. The current value of the LC is subtracted from the value computed for "locn" and the result modulo 256 is inserted into the DISP field (bits 55 through 48). In addition, TYPE (bits 62-61) is set to 11 and the COND field (bits 59-56) is cleared.

An error message is produced if ("locn" - current LC) is outside the range (-128 to +127).

(v) Conditional Branch

The general format is

BC(con) locn

A conditional transfer of control to "locn" takes place after executing the current instruction if the condition specified by "con" is true. Other-

wise the next instruction is executed.

The DISP and TYPE fields are set as for a BR. The COND field (bits 59-56) is set according to "con". The valid values for "con" are:

"con"	code (hex) (bits 59-56)	condition
AZ	1	flag AZ is set (ALU output = Ø)
ANZ	9	flag AZ is clear (ALU output ≠ Ø)
AP	2	flag AN is clear (ALU output is positive)
AN	A	flag AN is set (ALU output is negative)
FLC	6	flag C is set
FLD	7	flag D is set
FLE	E	flag E is set
FLF	F	flag F is set

(vi) Subroutines

The mnemonic "PUSH" used along with an appropriate branch mnemonic will result in pushing the current program counter onto the subroutine stack when the instruction is executed. The assembler codes a 1 in the PUSH field (bit 60). The mnemonics BRS and BCS used in place of BR and BC respectively produce the same result.

e.g. BR SUB1, PUSH; execute subroutine at SUB1

BRS SUB1; execute subroutine at SUB1

BCS (FLF) SUB2; execute subroutine SUB2 if flag F is set

A return from a subroutine is specified by the mnemonic

POP

which sets bit 60 (the POP field) in the generated object word.

(vii) SET(flags)

"flags" is a list of one or more letters from C, D, E, F. The assembler codes the SFL field (bits 23-20) according to "flags". Since the same bits are also used for the constant field, the same instruction may not specify immediate data.

e.g. SET(CF)

(viii) CLR(flags)

"flags" is a list of one or more letters from C, D, E, F. The assembler codes the CFL field (bits 19-17) according to "flags". Since the same bits are also used for the constant field, immediate data may not be specified in the same instruction.

e.g. CLR(DEF)

3.2.14 The Processing Element Resident Monitor

The PE Resident Monitor (PERM) is a package of routines that provide several utility functions for PE programs, such as interfacing with the various internal hardware units of the PE, communicating with CC programs, etc. The PERM normally occupies the lowest part of the microprocessor memory, usually the first 1 to 2 K words. Once the system software has stabilized, it may be desirable to put the PERM in a ROM. The various PERM routines to be described in the next few sections can be called as FORTRAN functions.

3.2.14.1 The PE Bootstrap

This routine is used for loading the microprocessor memory over the PEAC bus. Among other things, the rest of the PERM is loaded by it. It is more efficient than the built-in bootstrap of the LSI-11: it handles 16-bit words received from the CC via the PEAC instead of 8-bit bytes, and would otherwise be programmed for higher efficiency as well. It is the first program to be loaded into the PE at system initialization or when a new PE is added to the system or reinitialized. At the CC end, the data is transmitted by the LOADPE or LOADBC routines in the PEAC driver.

3.2.14.2 System Communication Handler (SCH)

The SCH provides a set of routines for handling all interaction with programs running on the CC or other PEs. The SCH also handles the interface to the Control Computer (CC) via the CC Communication port and the PE Array Controller (PEAC). It maintains two ring buffers to hold data received from the CC and data to be transmitted to the CC over this path. The following routines are included in the SCH and can be called as FORTRAN functions.

3.2.14.2.1 PSENDWD (WRD)

The 16-bit word WRD is transmitted to the CC over the PEAC bus.

3.2.14.2.2 PSENDCC (BUFAD, BUFLEN)

BUFLEN words of data beginning at BUFAD are transmitted to the CC over the PEAC bus. The format is compatible with that used by the IPECSM (see Section 3.2.8.4.1 for further details on the message format).

3.2.14.2.3 PSENDPE (BUFAD, BUFLEN, PEN)

The operation of this is the same as for SENDCC except that the message is addressed to PE number PEN.

3.2.14.2.4 PSIGCC (SIGNO)

Signal number SIGNO is sent to the CC. See Section 3.2.8.4.1 for the data format and types of signals.

3.2.14.2.5 PSIGPE (SIGNO, PEN)

Signal number SIGNO is sent to PE number PEN. See Section 3.2.8.4.1 for further details on signals.

3.2.14.2.6 PRECV (BUFAD, BUFLEN, LEN, SRC, FUNC)

A received message is returned in the buffer at BUFAD. If the message is larger than BUFLEN, it is truncated to BUFLEN. The length of the message is returned in LEN, and its source in SRC. The function FUNC is executed when the message has been received. The routine returns to its caller immediately. See Section 3.2.8.4.1 for further details on message formats, etc.

3.2.14.2.7 PRCWDR (WD, AVAIL)

Returns a word of data received from the CC in WD, if available, and sets the AVAIL flag to 1. If no data is available, AVAIL is cleared. The routine returns immediately in either case.

3.2.14.2.8 PINSIG (FUNC)

The function FUNC is executed when input data is received from the CC over the PEAC bus.

3.2.14.2.9 PDISP (PETLPTR)

The PE enters the "Dispatched Mode". In this mode, it executes tasks from a PE Task List (PETL). (See Section 3.2.8.2 for a detailed description of the Dispatched Mode and the PE Task List). PELPTR is used as a pointer into the PETL.

The PETL is basically a ring buffer, each element of which consists of a subroutine address and the address of a parameter list.

The hardware of the Data Routing Element Array (DREA - Section 2.4) and the Central Working Storage (CWS - Section 2.3) is designed such that a PE has exclusive access to a CWS memory bank during the execution of an instruction that results in a read-modify-write cycle. This ensures that a PE can increment the PETL pointer (PTLPTR) in an unambiguous manner even if other PEs attempt to increment it at the same time. The value of PETLPTR just before incrementing is used to fetch the subroutine and parameter-list addresses from the PETL. A special hardware register in the PEs I/O register area, the Last Input Register (LIR) in the PE Mapper holds the last input from CWS. This is utilized to obviate any possible ambiguities if several PEs attempt to access the PETL simultaneously.

The actual instruction sequence for PDISP looks as follows:

```
PDISP:      MOV 2(R5), PTRAD      ; GET ADDRESS OF PTLPTR

DPLOOP:     ADD #4, @ PTRAD       ; STEP PETL POINTER
            MOV @#LIR, RØ         ; GET POINTER VALUE BEFORE STEPPING
            MOV (RØ), R1          ; R1 CONTAINS RTN ADDRESS
            MOV#RETRY, (RØ)+N     ; RESET RTN TO RETRY
            MOV (RØ), R5          ; R5 POINTS TO PARAMETER LIST
            JSR PC, (R1)          ; GO EXECUTE THE SUBROUTINE
            BR DPLOOP             ; FETCH NEXT TASK

PTRAD:      .WORD Ø               ; ADDRESS OF PTLPTR GOES HERE
```

Note that after a task has been fetched, the PETL slot is reset to point to the subroutine RETRY, which will be described below. This ensures that if the PETL gets emptied out, the PEs keep retrying until the PE task dispatcher (PETD) in the CC can insert new entries.

The tasks to be executed are themselves subroutines following standard linkage conventions. Thus, they can be written in FORTRAN and executed either in the Dispatched Mode or directly.

3.2.14.2.10 PDEXIT ( )

The PE exits the Dispatched Mode. The implementation of this consists essentially of unstacking an extra element from the subroutine stack.

3.2.14.2.11 LUPBAK (PTLB)

This routine is used in Dispatched Mode only to reset the ring buffer pointer PTLPTR to the top of the PETL (PTLB). See Section 3.2.8.2.1 for further details.

3.2.14.2.12 RETRY ( )

This routine is used in Dispatched Mode only. When a task is fetched from a PETL, the corresponding element is set to point to RETRY. Thus if the PETL gets emptied out by the PEs before the PETD running on the CC gets around to filling it up, the PEs execute this routine. The element of the PETL pointed to by the PETL pointer is continuously examined by RETRY until it is different from the address of RETRY. At that point, the corresponding task is executed, and the PE continues to fetch and execute tasks from the PETL in the normal manner. See Section 3.2.8.2.1 for further details.

3.2.14.2.13 DUMMY ( )

This consists of just a return. It is used in Dispatched Mode as follows: the PETL ring buffer is followed by several entries of DUMMY. If several PEs come to the bottom of the ring buffer at the same time, all but one will execute DUMMY, and then fetch a valid task. See Section 3.2.8.2.1 for further details.

3.2.14.2.14 PEJOIN (CTR)

This routine is used at synchronization points where a group of PEs is required to wait until they have all com-

pleted all previous computations before proceeding to subsequent ones. CTR is a word in Central Working Storage (CWS) which is initially set to the number of PEs involved. Upon executing PEJOIN, the PE decrements CTR and then waits until CTR is zero before returning. This routine is utilized in Dispatched Mode by the PETD routine PETDJOIN. See Section 3.2.8.2.2.5.

3.2.14.2.15 PWAIT (SEM)

This is used for mutual exclusion and other kinds of signalling between PEs. It is similar in operation to Dijkstra's P. The word SEM in Central Working Storage (CWS) is decremented and the PE waits until the result is non-negative.

3.2.14.2.16 PSIG (SEM)

This is a companion to PWAIT above. It is similar in operation to Dijkstra's V. The effect is essentially to increment SEM.

3.2.14.3 The PE Channel Handler (PECH)

The PECH provides PE programs written in FORTRAN with a capability of programming the PE channel. The PE Channel (see Section 2.2.1.4) can move blocks of data between any two places within its 16 megabyte address space which includes the entire Central Working Storage (CWS), the PE Local Storage (PELS), the CAPU Microprogram Storage and the IOAG Microprogram Storage. The data may be moved in one of four modes - byte, 16-bit word, 32-bit word and 64-bit word. Further, any increment may be specified between successive words at the source or destination. This can be used, for example, to read or write a column of a matrix stored by rows.

The following functions can be called to invoke the PECH. The value returned by a function, if non-zero, indicates an error as follows:

1 - illegal input

2 - parity error

3 - timeout error (normally results from attempts to access non-existent memory)

In the following descriptions, the parameters SRCADR, SRCINC, DSTADR, DSTINC, WRDENT are of type INTEGER*4.

3.2.14.3.1 CHMOVE (SRCADR, SRCINC, DSTADR, DSTINC, WRDCNT, MODE)

The channel is directed to transfer WRDCNT words of the size indicated by MODE. The data starting at SRCADR with steps of SRCINC bytes is transferred to the area DSTADR with steps of size DSTINC. The possible values of MODE are:

MODE = 0 : 64-bit words (8 bytes)

MODE = 1 : 32-bit words (4 bytes)

MODE = 2 : 16-bit words (2 bytes)

MODE = 3 : 8-bit words (bytes)

Note that to specify a contiguous block of memory, the increment (SRCINC, DSTINC) must be equal to the word size in bytes specified by MODE. For example to move 4096 bytes

from vector A to vector B, one could use:

```
CHMOV (A, 8, B, 8, 512, Ø)
```

CHMOV returns immediately to the caller after initiating the transfer. CHWAIT Is used to test for when the operation is completed.

3.2.14.3.2 CHWAIT ( )

This function returns to the calling program when the channel has completed its operation. An error detected during the previous transfer is indicated by the value returned by the function. This function is used in conjunction with CHMOVE.

3.2.14.3.3 CHMOVF (SRCADR, SRCINC, DSTADR, DSTINC, WRDCNT, MODE, FUNC)

The operation of this function is identical to CHMOVE except that when the specified transfer has been completed, the function FUNC is executed.

3.2.14.3.4 CHSTAT (BUF)

This returns the current value of the channel's interface registers in the area BUF as shown in Figure 3.2.14.3.4-1. See Section 2.2.1.4 for a description of these registers.

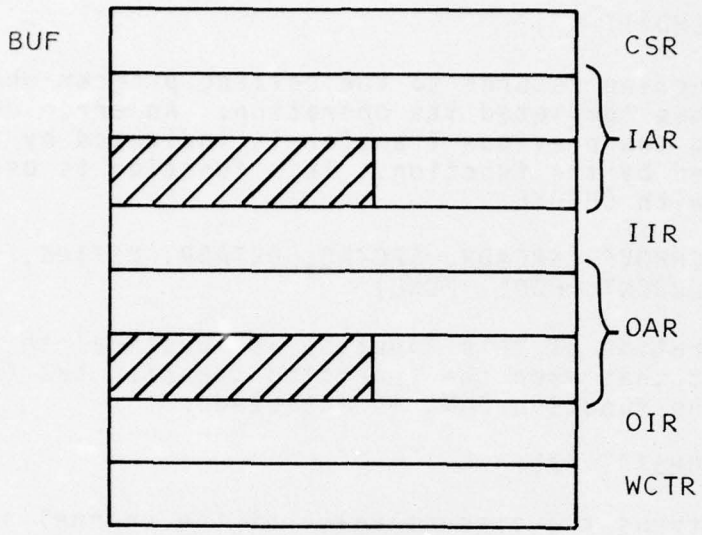


Figure 3.2.14.3.4-1. CHSTAT Format

3.2.14.4 Arithmetic Processor Control System (APCS)

The APCS provides a program running on the PE with the capability of controlling the Arithmetic Processor (AP) fully via the AP Controller. The AP Controller (Section 2.2.1.7.6) consists of the hardware that gives the PE microprocessor complete control over the various modules of the AP: the Central Arithmetic Processing Unit (CAPU), the Input/Output Address Generator (IOAG), and the four AP FIFOs.

3.2.14.4.1 CAPU Control Functions

The following FORTRAN-compatible functions are used for the control of the CAPU. Except where otherwise noted the parameters are 16-bit integers. See Section 2.2.1.7.6 for a detailed description of the CAPU interface registers, flags, etc.

3.2.14.4.1.1 APRUN (STAD)

The starting address STAD is loaded into the start register and the CAPU is put into RUN mode by setting the RUN bit in the Control Register.

3.2.14.4.1.2 APHALT ( )

The CAPU is halted by clearing the RUN bit. The RUN bit also controls access to the CAPU microprogram storage. The channel can load new microprograms only when RUN is cleared; i.e. an APHALT must be issued before the channel is directed to change the CAPU microprograms.

3.2.14.4.1.3 APFLAG (F)

This function returns the value of the flag identified by F. The valid flags are

F	Flag
0	A
1	B
2	C
3	D

3.2.14.4.1.4 APFLSET (F,VAL)

The flag identified by F (see APFLAG) is set to the value VAL (Ø or 1).

3.2.14.4.1.5 APSTAT ( )

The contents of the CAPU Status Register (Fig. 2.2.1.7.6.1-1) is returned.

3.2.14.4.1.6 APINT (MASK, FUNC)

The function FUNC is executed when any of the bits in the Status Register corresponding to 1's in MASK are set. This could include any flags or error conditions.

3.2.14.4.1.7 APDIR ( )

The CAPU is put in Direct Mode. In this mode, the CAPU gets its inputs from the microprocessor via the AP Controller instead of the Input Data FIFO, and similarly sends its outputs to the microprocessor instead of the Output Data FIFO. The functions APSEND and APGET are used for this purpose. This mode is used for diagnostics and for scalar floating point operations.

3.2.14.4.1.8 APSEND (APIN)

This is only used with the CAPU in Direct Mode. The 64-bit quantity at APIN is sent to the CAPU simulating an input from the Input Data FIFO. APIN could be a complex number, a real vector of length two or an integer vector of length four.

3.2.14.4.1.9 APGET (APOUT)

This is only used with the CAPU in Direct Mode, in conjunction with APSEND. The 64-bit output of the CAPU is returned in APOUT, simulating an output to the Output Data FIFO. If no data is available, the function returns a non-zero value.

3.2.14.4.1.10 APPC ( )

Returns the current value of the CAPU program counter.

3.2.14.4.1.11 APSSTEP ( )

The CAPU executes a single instruction each time this function is executed. It has no effect if the CAPU is in RUN mode. This function is included for diagnostic purposes.

3.2.14.4.1.12 HLTERR (M)

The CAPU is directed to halt on errors if M is a 1, or keep running in the event of errors if M is a Ø. Note that the error bits are set in the CAPU status register in either case when an error occurs.

3.2.14.4.2 IOAG Control Functions

The following FORTRAN-compatible functions are used to control and monitor the Input-Output Address Generator (IOAG). See Section 2.2.1.7.6.2 for a detailed description of the IOAG interface registers and flags.

3.2.14.4.2.1 IORUN (STAD)

The starting address STAD is loaded into the start register and the IOAG is put into the RUN mode by setting the RUN bit in the IOAG Control Register.

3.2.14.4.2.2 IOHALT ( )

The IOAG is halted by clearing the RUN bit. The RUN bit also controls access to the IOAG microprogram storage. The channel can load new microprograms only when the IOAG is halted (RUN = Ø); i.e. a IOHALT( ) must be issued before directing the channel to load IOAG microprograms.

3.2.14.4.2.3 IOFLAG (F)

This function returns the value of the IOAG flags identified by F. The valid flags are:

F	Flag
2	C
3	D
4	E
5	F

Note that flags C and D are common to the CAPU and the IOAG. Thus APFLAG(2) and IOFLAG(2) have the same effect.

3.2.14.4.2.4 IOFLSET (F, VAL)

The flag specified by F (see IOFLAG above) is set to VAL which must be Ø or 1.

3.2.14.4.2.5 IOSTAT ( )

The contents of the IOAG Status Register (Fig. 2.2.1.7.6.2-1) is returned.

3.2.14.4.2.6 APINT (MASK, FUNC)

The function FUNC is executed when any of the bits in the IOAG Status Register corresponding to 1's in the IOAG Mark Register are set.

3.2.14.4.2.7 IODIR ( )

The IOAG is put in the Direct Mode (see Section 2.2.1.7.2). In this mode, the microprocessor (via the AP controller) simulates the Input Address FIFO and the Output FIFO and receives the IOAG outputs directly. This mode is used primarily for diagnostic purposes.

3.2.14.4.2.8 IOGETI (II)

The address generated by the IOAG for the Input Address FIFO is returned in II, simulating the FIFO. II is a 32-bit integer. This function is used only in the Direct Mode.

3.2.14.4.2.9 IOGETO (IO)

The address generated by the IOAG for the Output Address FIFO is returned in IO, simulating the FIFO. IO is a 32-bit integer. This function is used only in the Direct Mode.

3.2.14.4.2.10 IOPC ( )

Returns the current value of the IOAG program counter.

3.2.14.4.2.11 IOSSTEP ( )

The IOAG executes a single instruction each time this function is called. It has no effect if the IOAG is in RUN mode. This function is included for diagnostic purposes.

3.2.14.4.3 FIFO Control Functions

The following functions are used to monitor the status of the AP FIFOs and the Data Fetch Unit and Data Store Unit. (See Sections 2.2.1.7.3 and 2.2.1.7.4).

3.2.14.4.3.1 FISTAT ( )

This function returns the contents of the FIFO Status Register (Figure 2.2.1.7.6.2-1) which indicates the status of all four FIFOs (full or empty) and the Data Store Unit (DSU) and Data Fetch Unit (DFU).

3.2.14.4.3.2 FICONT (IF)

This function returns the number of words in the FIFO identified by IF. The valid values of IF are:

IF = 1 - Input Address FIFO

IF = 2 - Input Data FIFO

IF = 3 - Output Address FIFO

IF = 4 - Output Data FIFO

3.2.14.4.3.3 FISIG (FUNC)

The function FUNC is executed when all four FIFOs as well as the DFU and the DSU are empty. This is used to signal the completion of an array computation.

3.2.14.5 The Mapper Handler

The Mapper Handler makes it possible for a program to change the PE Mapper Segment Registers and thereby gain access to any part of the CWS or PELS. See Sec. 2.2.1.2 for a detailed description of the PE Mapper hardware and its segment registers. Normally, for FORTRAN programs, the Mapper registers are set up by the loader and would not be altered dynamically. The Mapper Handler, however, provides the capability for doing so, if desired. A similar Mapper Handler is also available for use by CC programs to handle an identical Mapper between the CC and the CWS. The following functions can be called to invoke the Mapper Handler:

3.2.14.5.1 SEGSET (SEGNO, BASE, LEN)

Segment register number SEGNO is set to provide a window of length LEN bytes starting at BASE. BASE must be a 4-byte integer giving the desired CWS or PELS address. SEGNO must lie between 1 and 6 and specifies the corresponding 8 kbyte slot in the microprocessor 64 kbyte address space.

3.2.14.5.2 SEGACT (SEGNO, VAL)

The bit in the Active Segment Register (ASR) corresponding to SEGNO is set to VAL (1 or Ø). The effect of setting a bit to a "1" in the ASR is to map all addresses in the corresponding 8 kbyte slot according to the contents of the corresponding segment register.

3.2.14.5.3 PHYSAD (AD)

This function returns the physical address (in the extended 24-bit address space of the PE) corresponding to the virtual address AD; i.e., it returns the value that AD would be mapped to by the MAPPER.

3.2.14.5.4 LOGAD (EXTAD)

This function is the converse of PHYSAD. It returns the virtual address that would be mapped to EXTAD by the MAPPER. EXTAD is a 4-byte integer.

3.2.15 The Multiprocessor Linkage Editor (MPLE)

The MPLE is responsible for relocating and linking together relocatable program sections which are to be loaded into the CC memory, local PE memory, and the CWS. The input to the MPLE consists of a number of files containing relocatable object modules produced by system compilers such as FORTRAN and the assembler. The output consists of one or more files containing load modules to be loaded by the Multiprocessor Loader (MPL) into the CC memory, PEs or CWS. The load modules produced by the MPLE also contain information used by the loader to set the MAPPER registers in the CC and the PEs.

The output of the Linkage Editor consists, in general, of a load module for the CC, one or more load modules for the CWS and one or more load modules for PEs. In the usual case, all the PEs involved in a computation are loaded with the same programs so that only one PE load module is required. However, if it is desired to have several sets of PEs working on different parts of the computation in parallel performing different operations, each of these sets of PEs would correspond to a PE load module. The load modules produced by the MPLE may be written on the same or different files.

A Linkage Map is also produced upon request.

The object modules targeted for a single load module are relocated and linked together in the usual manner. A reference to a global name in any load module generates its address within its load module. Thus a CC routine can refer to a PE procedure: the value inserted by the MPLE is the address of the procedure in the PE. (However, if it tried to call the procedure directly, it would transfer control to the corresponding address in the CC instead). This scheme makes it very simple to do things like having the CC direct a PE to execute a particular subroutine. However all Global Names, which include FORTRAN Subroutines, Function and Entry Points, must be unique over the entire group of programs involved in a link-edit.

In addition to object modules explicitly placed in a CWS load module, all CSECTs in common between object modules targeted for different load modules are automatically placed in CWS. Thus, a FORTRAN routine running on the CC and a PE can share variables or arrays simply by declaring them in COMMON areas of the same name in both programs.

When a CC or PE program references CWS, the corresponding area of CWS is assigned to one of the "windows" or segments provided by the MAPPER. The CWS area corresponding to each such CWS load module is assigned to a separate Mapper segment (if the CWS area is larger than 8 kbytes, a number of contiguous segment registers have to be assigned). A total of six "windows" or mapper segments is available for a CC or PE load module. If more are required, it represents a fatal error. The mapper segment registers are loaded by the MPL; the actual values loaded into them depend on the physical location of the corresponding modules in CWS.

3.2.15.1 MPLE Command Format

The operation of the MPLE is controlled by one or more command lines terminated by a blank line. If called as a subroutine, it is passed as a null-terminated ASCII string of the same format.

Each command line corresponds to one load module and has the following format:

dest, ofile[,addr]←ifile {,ifile}

"dest" is one of CC, P.E., CWS and specifies the type of load module to be produced. "file" is the file in which the load module is placed.

The same "ofile" may be used for more than one load module. "addr" is an optional argument specifying the address at which the code is to be located. Each "ifile" is the name of a file containing object modules. If "ifile" is a library file, only object modules required to resolve global references are included.

In addition to one or more command lines of the form shown above, any number of command lines of the format shown below may be used:

CWSX, ofile[,addr]←[csect] {,csect}

These command lines are used in connection with CSECTS (such as FORTRAN COMMONs shared by CC and P.E. Programs) that are assigned by the MPLE automatically to CWS. If omitted, all such areas are put in a CWS load module on the "ofile" specified in the command line specifying the CC "dest".

For each CWSX command, a load module is created on the file specified by "ofile" containing the CSECTS given by each "csect".

This makes it possible to put different COMMON areas in separate load modules so that the loader (MPL) can be directed to load them in separate CWS banks to minimize memory conflicts.

3.2.16 The Multiprocessor Loader

The MPL loads the load modules produced by the Multiprocessor Linkage Editor (MPLE) into the memories of the CC or PEs or into the Central Working Storage (CWS). Facilities are provided for specifying that certain CWS load modules are to be loaded in separate CWS banks to minimize memory conflicts.

The load modules produced by the MPLE contain information regarding the destinations (CC, PE or CWS) and may contain starting addresses, if specified at link time. If no addresses are specified, the loader uses appropriate default values.

The loader also loads the MAPPER segment registers of the CC and the PEs, using information supplied by the MPLE, and the addresses at which the relevant CWS load modules are loaded.

The operation of the MPL is controlled by one or more command lines terminated by a blank line. When the MPL is called as a subroutine, it is passed a null-terminated ASCII string containing the command lines.

Each command line has the following format:

```
[PE(pen{,pen}),]ifile
```

Each "pen" is a PE number between Ø and 31 (for a system with up to 32 PEs). "Ifile" is the name of a file containing one or more load modules produced by the MPLE. If "ifile" contains a PE load module, it is loaded into each of the PEs in the given list, a CC module is loaded into the CC and any CWS modules are loaded into the CWS. If the CWS modules do not specify specific addresses, the CWS manager's facilities are used to assign their locations.

CWS load modules corresponding to different command lines are loaded in separate memory banks where possible. Thus, to load two CSECTs in separate banks to minimize memory conflicts, one directs the MPLE to place them in load modules on separate files and then uses separate command lines for them with the loader.

3.2.17 Programs for Systems Reliability

This package contains programs designed to maximize system reliability and availability. One of the major advantages of a modular multi-processor system like the G-471 is that the system can continue to operate when part of the hardware is malfunctioning. The hardware and software of the G-471 is designed to exploit the potential for high reliability inherent in the basic architecture.

The requirements for achieving this high reliability are:

(i) Any hardware malfunction should be detected as soon as possible.

(ii) It should be possible to reconfigure the system by removing the malfunctioning hardware and switching in spares if necessary.

(iii) The software should make it possible for programs to operate with different hardware configurations though the efficiency may be reduced when disabled modules are switched out (graceful degradation). Where execution speeds are critical, it should be possible to have spare modules that can replace malfunctioning ones (redundancy).

(iv) When a malfunction is detected during a computation, facilities have to be provided to repeat affected parts of the computation.

3.2.17.1 Diagnostic Routines

The system hardware is designed to make it easy to detect malfunctions. Examples of this are:

(i) All memories within the system (CWS, PELS, microprogram memories, etc.) include parity bits (1 bit per byte). The parity is generated and checked at the processor end so that errors in the memory as well as in the data paths (such as the Data Routing Element Array) can be detected.

(ii) Most of the registers within various parts of the PE are made accessible to the LSI-11 microprocessor. This makes it easier to write diagnostic programs running on the PE microprocessor to check out the rest of the PE hardware.

(iii) The IOAG and the CAPU have a "Direct Mode" in which they are disconnected from their FIFOs and the PE microprocessor can simulate the FIFOs and directly control their I/O. Further, a microprocessor program can direct the CAPU or the IOAG to execute a single instruction at a time. Since the microprocessor can also load any program into the IOAG and CAPU microprogram memories, it is possible to write exhaustive test routines for checking out these processors.

The Diagnostics Package (DP) includes routines to check each module of hardware in the system for correct operation.

There is a program called the System Hardware Monitor (SHM) that runs on the CC at a low priority and constantly tests the system hardware. It can be programmed via a control file to execute diagnostic programs from the Diagnostic Package, in any sequence and at any frequency. In addition, it can be programmed to test modules that are not being used by anybody. It gets information about usage of these modules from their resource managers: it finds out which CWS banks are not allocated from the CWS manager and the PE. Allocation Manager provides similar information on PEs. The control file also specifies the action to take upon detection of errors. This could include messages to the operator or calls to the System Configuration Manager to remove the malfunctioning unit from operation.

3.2.17.2 System Reconfiguration

The configuration of the system hardware can be altered dynamically: modules that are malfunctioning or need maintenance can be removed from service, and repaired modules or replacements can be switched in and added to the pool of available system resources. This is facilitated by several features of the G-471 hardware and the MPOS/471 software:

(i) Individual PEs can be disconnected physically from the system by the CC through the PEAC (Section 2.1.2)

(ii) Programs written utilizing the facilities of the PE Array Control and Communication Package (Section 3.2.8) and particularly the PE. Allocation Manager (Section 3.2.8.1) and PE Task Dispatcher (Section 3.2.8.2), can generally run independently of the particular PEs available or even the num-

ber of PEs available.

(iii) Each bank of the Central Working Storage (CWS) can be assigned an address independent of its physical position by loading the Bank-Decoder registers of the Data Routing Element Array (Section 2.1.5). Thus any malfunctioning part of the CWS can be switched out of service and another memory bank assigned the same address; this is completely transparent to programs.

(iv) The Control Computer (CC) itself can be made redundant by adding a spare CPU, and it can be switched into service to keep the system operating if the CC CPU malfunctions.

The hardware configuration of the system is under the control of the System Configuration Manager (SCM). The SCM maintains tables representing the system configuration, and provides the required information to the system resource managers such as the PE. Allocation Manager and the CWS Manager.

The SCM also handles the actual hardware reconfiguration: it disconnects or connects PEs, loads the Bank Decoder registers of the Data Routing Element Array, etc.

The directives to the SCM to remove or add a module usually come from the SHM or from the system operator.

3.2.17.3 Error Handling

A program can specify the actions to be taken when an error is detected, either by the system or by the program itself. It can elect to handle the error itself, or it can request system action which could be about the computation, log an error and continue, or restart the computation at the last checkpoint. Suitable defaults actions can be set when the program does not specify any particular kind of error-handling action.

The functions described above are performed by the Error Handling System (EHS). The EHS also handles checkpoints: a computation calls the EHS to request a checkpoint. All data files accessed since the last checkpoint are dumped, so that in the event of an error, the data can be recovered and the computation restarted at the last checkpoint before the error is detected. A program requesting a checkpoint can also explicitly specify specific files as well as areas of CWS to be dumped.

3.2.18 Standard Subroutine Library (SSL)

The SSL is a library of subroutines that can be used to perform most commonly used functions involved in image-processing and array-processing in general. An SSL subroutine called by a CC program takes care of distributing the requested task among a set of processing elements for processing. Thus the FORTRAN programmer does not get involved in the details of communicating with the PEs or of programming the PE Microprocessor, Input-Output Processor (IOAG), Central Arithmetic Processing Unit (CAPU) or the PE Channel. The SSL permits a programmer to essentially ignore the existence of the various hardware modules working in parallel for his computation and program as though he were programming a normal sequential computer while taking advantage of the parallel architecture to obtain very high execution speeds.

When a CC program with a call to an SSL subroutine is Link-edited, the MPLE automatically includes the called subroutine and other subroutines that it refers to and so on. As a result, the routines required for the PE microprocessor, CAPU and IOAG are automatically included in the load modules by the MPLE, and loaded into the appropriate processors by the MPL.

The rest of this section is a brief description of the members of the SSL. The exact composition of the SSL would depend on the application environment and the user can add his own routines to it.

In the subroutine description below, the parameter PELST points to the vector containing the numbers of the PEs to be used by the subroutine, followed by a -1 entry. This vector would have been obtained previously by a call to the PE Allocation Manager. All vectors or arrays must be in CWS and their physical addresses must be passed. These addresses are normally obtained from the CWS manager. If an array in a COMMON area is involved which is automatically placed in CWS by the MPLE, its physical addresses may be obtained by calling the PHYSAD function of the Mapper Handler.

Errors in the input parameters or detected during the computation are indicated by returning negative function values.

The type of arrays passed as parameters are indicated in the description of the routine by the first 1 or 2 letters of the name as follows:

Name beginning with:	Type of array
V	a vector of real numbers (32 bits)
IV	a vector of integers (16 bits)
CV	a vector of complex numbers (64 bits)
A	a matrix of real numbers
IA	a matrix of integers
CA	a matrix of complex numbers.

The dimension of the arrays is N for vectors, NxN for matrices where N is usually the last parameter.

3.2.18.1 FFT (PELST, VI, VO, N)

This subroutine computes the FFT of the vector VI of length N, and stores the result in VO. VI, VO are vectors of complex numbers (64 bits). If VI = VO, an in-place computation is possible.

3.2.18.2 IFFT (PELST, VI, VO, N)

Performs an inverse FFT. See FFT above for other details.

3.2.18.3 FFT2D (PELST, AI, AO, N)

This subroutine computes the 2-dimensional FFT of the NxN matrix AI and stores the result in AO. AI, AO are matrices of complex numbers (64 bits). If AI = AO, an in-place computation is performed.

3.2.18.4 IFFT2D (PELST, AI, AO, N)

Performs a 2-dimensional inverse FFT. See FFT2D above for other details.

3.2.18.5 VSUM (PELST, VA, VB, VO, N)

This performs the vector sum

$$VO_i = VA_i + VB_i, \quad i = 1 \text{ to } N$$

VA, VB, VO are vectors of real numbers.

3.2.18.6 VSUMI (PELST, IVA, IVB, IVO, N)

Same as VSUM except that IVA, IVB, IVO are vectors of integers.

3.2.18.7 VSUMC (PELST, CVA, CVB, CVO, N)

Same as VSUM except that VA, VB, VO are vectors of complex numbers.

3.2.18.8 ASUM (PELST, AA, AB, AO, N)

Same as VSUM except that AA, AB, AO are NxN matrices of real numbers.

3.2.18.9 ASUMI (PELST, IAA, IAB, IAO, N)

Same as VSUM except that IAA, IAB and IAO are NxN matrices of type integer.

3.2.18.10 ASUMC (PELST, CAA, CAB, CAO, N)

Same as VSUM except that CAA, CAB, CAO are NxN matrices of type complex.

3.2.18.11 VMUL (PELST, VA, VB, VO, N)

Performs the computation:

$$VO_i = VA_i \cdot VB_i, \; i = 1 \text{ to } N$$

The following five functions perform a similar computation, with the type of arrays indicated by the names of the parameters as described earlier:

3.2.18.12 VMULI (PELST, IVA, IVB, IVO, N)

3.2.18.13 VMULC (PELST, CVA, CVB, CVO, N)

3.2.18.14 AMUL (PELST, AA, AB, AO, N)

3.2.18.15 AMULI (PELST, IAA, IAB, IAO, N)

3.2.18.16 AMULC (PELST, CAA, CAB, CAO, N)

The following six subroutines perform element by element division on the kinds of arrays indicated by their parameter names.

3.2.18.17 VDIV (PELST, VA, VB, VO, N)

Performs the computation

$$VO_i = VA_i/VB_i, \quad i = 1 \text{ to } N.$$

3.2.18.18 VDIVI (PELST, IVA, IVB, IVO, N)

3.2.18.19 VDIVC (PELST, CVA, CVB, CVO, N)

3.2.18.20 ADIV (PELST, AA, AB, AO, N)

3.2.18.21 ADIVI (PELST, IAA, IAB, IAO, N)

3.2.18.22 ADIVC (PELST, CAA, CAB, CAO, N)

3.2.18.23 SQRT (PELST, VA, VO, N)

$$VO_i = \sqrt{VA_i} \text{ for } i = 1 \text{ to } N$$

3.2.18.24 ASQRT (PELST, AA, AO, N)

$$AO_{i,j} = \sqrt{AA_{i,j}} \text{ for } i = 1 \text{ to } N, \; j = 1 \text{ to } N$$

3.2.18.25 VLOG (PELST, VA, VO, N)

$$VO_i = \ln (VA_i) \text{ for } i = 1 \text{ to } N$$

3.2.18.26 ALOG (PELST, AA, AO, N)

$$AO_{i,j} = \ln (AA_{i,j}) \text{ for } i = 1 \text{ to } N, \; j = 1 \text{ to } N$$

3.2.18.27 VABSPH (PELST, CVI, VO, VP, N)

This computes the absolute value and phase of the complex elements of CVI:

$$\left.\begin{aligned} VO_i &= \text{abs } (CVI_i) \\ VP_i &= \text{phase } (CVI_i) \end{aligned}\right\} \quad i = 1 \text{ to } N$$

3.2.18.28 AABSPH (PELST, CAI, AVO, AVP, N)

Computes the absolute values and phases of the complex elements of the NxN complex matrix CAI.

3.2.18.29 MTMUL (PELST, AA, AB, AO, N)

Computes the matrix product of the NxN matrices AA, AB.

3.2.18.30 MTINV (PELST, AI, AO, N)

Computes the inverse of the NxN matrix AI.

3.2.18.31 VIP (PELST, VA, VB, S, N)

Computes the inner product of the vectors VA, VB of length N. The result is returned in the real scalar S.

3.2.18.32 CONV (PELST, VA, VB, VO, N)

Computes the convolution of the vectors VA, VB:

$$VO_i = \sum_{j=0}^{N-1} VA_{i-j} \cdot VB_j \quad \text{for } i = 1 \text{ to } N$$

4. MAINTAINABILITY AND RELIABILITY

The maintenance concept for the G-471 takes advantage of the fact that the parallel building blocks of the computer can be used to provide redundancy in case of parts failures. In most cases it would not be necessary to shut down the system when a component fault occurs. Rather, the remaining modules are reconfigured to continue operation. The design allows on-line maintenance, i.e. the removal of a faulty board while the rest of the system is powered and operating. In the ideal case the algorithms which are running on the machine at the time of a parts failure do not require all PEs or CWS banks or disk controllers to meet their execution-time constraints. In this case, a module which would have been idle acts as a spare and there is no reduction in throughput. If all PEs are required to meet a specified processing rate then the failure of a PE will temporarily slow the machine down, typically by the ratio of working PEs over the total number of PEs. If all banks of the CWS are required for the execution of a given algorithm, and if one of them fails, then disks can be used for the storage of the interim results, but this would usually slow down the operation substantially and would also require the availability of alternate software packages which can use disk for intermediate results. It is therefore advisable to size the CWS such that there is always at least one spare bank available. If one of the controllers or disk drives fails, it may affect the ability to obtain input data or dump output data fast enough to sustain the desired throughput. It is therefore again desirable to provide one more controller and disk drive than are necessary to meet the execution-time specifications.

The control computer should also be made redundant at least to the extent that it is needed for the operation of the array. It is probably not necessary to duplicate peripherals which are not directly used in the production jobs of a given installation.

The reliability block diagram for the G-471 architecture is shown in Figure 4-1. It assumes that the overall computer is organized into groups of not more than 32 PEs or memory banks each. For all near-time applications only one group

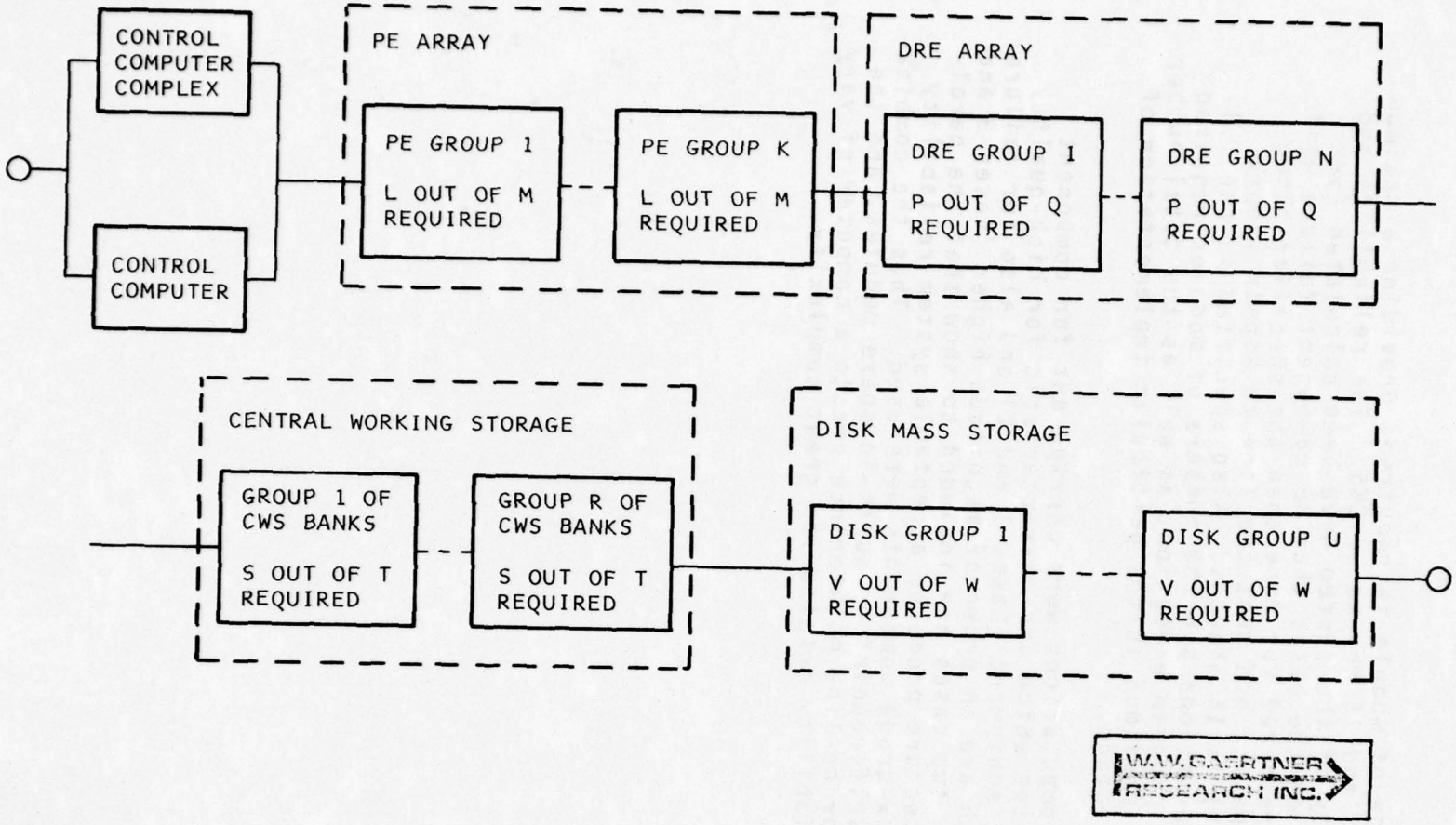


Figure 4-1. Reliability Block Diagram for G-471 Computer System

of each type of module is required, providing a maximum of 128 MIPS and 16 Mbytes of CWS. The reliability and availability of the system have been calculated (see Table 4-2) on the basis that no component failure will be left unrepaired for more than the specified "Time Between Repairs" and that the Time to Repair (board replacement) is 15 minutes. Also specified in the following computations are the numbers of modules required for actual program execution, as well as the total number of modules provided in the particular implementation of the architecture.

The computations were carried out for component failure rates which are characteristic for high-quality commercial equipment (Cases 1 and 2) and also for failure rates which are an order of magnitude higher (Cases 3 and 4). These two cases are included to show that the parallel architecture produces acceptable system reliability even if low-grade components were used. Thus the combination of "self-repair" by built-in spare modules, and the ability for on-line maintenance create a computer of very high availability despite its great complexity.

```
 CASE NUMBER.............................:01
C.......................................:
 MAXIMUM MISSION TIME....................:20,000. HOURS
C.......................................:
 MISSION TIME INCREMENT..................:1,000. HOURS
C.......................................:
 TIME BETWEEN REPAIRS....................:4.0 HOURS
 MEAN TIME TO REPAIR.....................:0.250 HOURS
C.......................................:
 FAILURE RATE OF CONTROL COMPUTER.......:+1.00000000E-04
C.......................................:
 TOTAL NUMBER OF COMPUTERS /GROUP.......:2
 NUMBER OF REQUIRED COMPUTERS /GROUP....:1
 NUMBER OF CONTROL COMPUTER GROUPS......:1
C.......................................:
 FAILURE RATE OF PE......................:+2.60000000E-04
C.......................................:
 TOTAL NUMBER OF PE'S PER GROUP.........:22
 NUMBER OF REQUIRED PE'S PER GROUP......:18
 NUMBER OF PE GROUPS.....................:1
C.......................................:
 FAILURE RATE OF DATA ROUTING ELEMENT...:+2.00000000E-04
C.......................................:
 TOTAL NUMBER OF DRE'S PER GROUP........:28
 NUMBER OF REQUIRED DRE'S PER GROUP.....:24
 NUMBER OF DRE GROUPS....................:1
C.......................................:
 FAILURE RATE OF CWS BANK................:+8.00000000E-04
C.......................................:
 TOTAL NUMBER OF CWS'S BANKS PER GROUP..:28
 NUMBER OF REQUIRED BANKS PER GROUP.....:25
 NUMBER OF CWS GROUPS....................:1
C.......................................:
 FAILURE RATE OF MASS STORAGE MODULE....:+1.50000000E-04
C.......................................:
 TOTAL NUMBER OF MSM'S PER GROUP..... :4
 NUMBER OF REQUIRED MSM'S PER GROUP.....:3
 NUMBER OF MSM GROUPS....................:1
```

Table 4-2 - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

CASE NO:01

TMISS	RSYS	MTBF	AVAIL
1000.	0.9989198	9.2521550E 05	1.0000
2000.	0.9978407	9.2521550E 05	1.0000
3000.	0.9967628	9.2521550E 05	1.0000
4000.	0.9956860	9.2521550E 05	1.0000
5000.	0.9946104	9.2521550E 05	1.0000
6000.	0.9935360	9.2521550E 05	1.0000
7000.	0.9924627	9.2521550E 05	1.0000
8000.	0.9913906	9.2521550E 05	1.0000
9000.	0.9903197	9.2521550E 05	1.0000
10000.	0.9892499	9.2521550E 05	1.0000
11000.	0.9881813	9.2521550E 05	1.0000
12000.	0.9871138	9.2521550E 05	1.0000
13000.	0.9860475	9.2521550E 05	1.0000
14000.	0.9849823	9.2521550E 05	1.0000
15000.	0.9839183	9.2521550E 05	1.0000
16000.	0.9828554	9.2521550E 05	1.0000
17000.	0.9817937	9.2521550E 05	1.0000
18000.	0.9807331	9.2521550E 05	1.0000
19000.	0.9796737	9.2521550E 05	1.0000
20000.	0.9786154	9.2521550E 05	1.0000
21000.	0.9775582	9.2521550E 05	1.0000

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

```
 CASE NUMBER.............................:02
C.......................................:
 MAXIMUM MISSION TIME....................:20,000. HOURS
C.......................................:
 MISSION TIME INCREMENT..................:1,000. HOURS
C.......................................:
 TIME BETWEEN REPAIRS....................:1.0 HOURS
 MEAN TIME TO REPAIR.....................:0.250 HOURS
C.......................................:
 FAILURE RATE OF CONTROL COMPUTER.......:+1.00000000E-04
C.......................................:
 TOTAL NUMBER OF COMPUTERS /GROUP.......:2
 NUMBER OF REQUIRED COMPUTERS /GROUP....:1
 NUMBER OF CONTROL COMPUTER GROUPS......:1
C.......................................:
 FAILURE RATE OF PE......................:+2.60000000E-04
C.......................................:
 TOTAL NUMBER OF PE'S PER GROUP.........:22
 NUMBER OF REQUIRED PE'S PER GROUP......:18
 NUMBER OF PE GROUPS.....................:1
C.......................................:
 FAILURE RATE OF DATA ROUTING ELEMENT...:+2.00000000E-04
C.......................................:
 TOTAL NUMBER OF DRE'S PER GROUP........:28
 NUMBER OF REQUIRED DRE'S PER GROUP.....:24
 NUMBER OF DRE GROUPS....................:1
C.......................................:
 FAILURE RATE OF CWS BANK................:+8.00000000E-04
C.......................................:
 TOTAL NUMBER OF CWS'S BANKS PER GROUP..:28
 NUMBER OF REQUIRED BANKS PER GROUP.....:25
 NUMBER OF CWS GROUPS....................:1
C.......................................:
 FAILURE RATE OF MASS STORAGE MODULE....:+1.50000000E-04
C.......................................:
 TOTAL NUMBER OF MSM'S PER GROUP..... :4
 NUMBER OF REQUIRED MSM'S PER GROUP.....:3
 NUMBER OF MSM GROUPS....................:1
```

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

CASE NO:02

TMISS	RSYS	MTBF	AVAIL
1000.	0.9998468	6.5275257E 06	1.0000
2000.	0.9996937	6.5275257E 06	1.0000
3000.	0.9995405	6.5275257E 06	1.0000
4000.	0.9993874	6.5275257E 06	1.0000
5000.	0.9992343	6.5275257E 06	1.0000
6000.	0.9990812	6.5275257E 06	1.0000
7000.	0.9989282	6.5275257E 06	1.0000
8000.	0.9987752	6.5275257E 06	1.0000
9000.	0.9986222	6.5275257E 06	1.0000
10000.	0.9984692	6.5275257E 06	1.0000
11000.	0.9983162	6.5275257E 06	1.0000
12000.	0.9981633	6.5275257E 06	1.0000
13000.	0.9980104	6.5275257E 06	1.0000
14000.	0.9978575	6.5275257E 06	1.0000
15000.	0.9977047	6.5275257E 06	1.0000
16000.	0.9975518	6.5275257E 06	1.0000
17000.	0.9973990	6.5275257E 06	1.0000
18000.	0.9972462	6.5275257E 06	1.0000
19000.	0.9970935	6.5275257E 06	1.0000
20000.	0.9969407	6.5275257E 06	1.0000
21000.	0.9967880	6.5275257E 06	1.0000

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

```
 CASE NUMBER.............................:03
C.......................................:
 MAXIMUM MISSION TIME....................:20,000. HOURS
C.......................................:
 MISSION TIME INCREMENT..................:1,000. HOURS
C.......................................:
 TIME BETWEEN REPAIRS....................:4.0 HOURS
 MEAN TIME TO REPAIR.....................:0.250 HOURS
C.......................................:
 FAILURE RATE OF CONTROL COMPUTER.......:+1.00000000E-03
C.......................................:
 TOTAL NUMBER OF COMPUTERS /GROUP.......:2
 NUMBER OF REQUIRED COMPUTERS /GROUP....:1
 NUMBER OF CONTROL COMPUTER GROUPS......:1
C.......................................:
 FAILURE RATE OF PE......................:+2.60000000E-03
C.......................................:
 TOTAL NUMBER OF PE'S PER GROUP.........:22
 NUMBER OF REQUIRED PE'S PER GROUP......:18
 NUMBER OF PE GROUPS.....................:1
C.......................................:
 FAILURE RATE OF DATA ROUTING ELEMENT...:+2.00000000E-03
C.......................................:
 TOTAL NUMBER OF DRE'S PER GROUP........:28
 NUMBER OF REQUIRED DRE'S PER GROUP.....:24
 NUMBER OF DRE GROUPS....................:1
C.......................................:
 FAILURE RATE OF CWS BANK................:+8.00000000E-03
C.......................................:
 TOTAL NUMBER OF CWS'S BANKS PER GROUP..:28
 NUMBER OF REQUIRED BANKS PER GROUP.....:25
 NUMBER OF CWS GROUPS....................:1
C.......................................:
 FAILURE RATE OF MASS STORAGE MODULE....:+1.50000000E-03
C.......................................:
 TOTAL NUMBER OF MSM'S PER GROUP..... :4
 NUMBER OF REQUIRED MSM'S PER GROUP.....:3
 NUMBER OF MSM GROUPS....................:1
```

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

CASE NO:03

TMISS	RSYS	MTBF	AVAIL
1000.	0.0592980	3.5395987E 02	0.9993
2000.	0.0035163	3.5395987E 02	0.9993
3000.	0.0002085	3.5395987E 02	0.9993
4000.	0.0000124	3.5395987E 02	0.9993
5000.	0.0000007	3.5395987E 02	0.9993
6000.	0.0000000	3.5395987E 02	0.9993
7000.	0.0000000	3.5395987E 02	0.9993
8000.	0.0000000	3.5395987E 02	0.9993
9000.	0.0000000	3.5395987E 02	0.9993
10000.	0.0000000	3.5395987E 02	0.9993
11000.	0.0000000	3.5395987E 02	0.9993
12000.	0.0000000	3.5395987E 02	0.9993
13000.	0.0000000	3.5395987E 02	0.9993
14000.	0.0000000	3.5395987E 02	0.9993
15000.	0.0000000	3.5395987E 02	0.9993
16000.	0.0000000	3.5395987E 02	0.9993
17000.	0.0000000	3.5395987E 02	0.9993
18000.	0.0000000	3.5395987E 02	0.9993
19000.	0.0000000	3.5395987E 02	0.9993
20000.	0.0000000	3.5395987E 02	0.9993
21000.	0.0000000	3.5395987E 02	0.9993

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

```
 CASE NUMBER.............................:04
C.......................................:
 MAXIMUM MISSION TIME....................:20,000. HOURS
C.......................................:
 MISSION TIME INCREMENT..................:1,000. HOURS
C.......................................:
 TIME BETWEEN REPAIRS....................:1.0 HOURS
 MEAN TIME TO REPAIR.....................:0.250 HOURS
C.......................................:
 FAILURE RATE OF CONTROL COMPUTER.......:+1.00000000E-03
C.......................................:
 TOTAL NUMBER OF COMPUTERS /GROUP.......:2
 NUMBER OF REQUIRED COMPUTERS /GROUP....:1
 NUMBER OF CONTROL COMPUTER GROUPS......:1
C.......................................:
 FAILURE RATE OF PE......................:+2.60000000E-03
C.......................................:
 TOTAL NUMBER OF PE'S PER GROUP.........:22
 NUMBER OF REQUIRED PE'S PER GROUP......:18
 NUMBER OF PE GROUPS.....................:1
C.......................................:
 FAILURE RATE OF DATA ROUTING ELEMENT...:+2.00000000E-03
C.......................................:
 TOTAL NUMBER OF DRE'S PER GROUP........:28
 NUMBER OF REQUIRED DRE'S PER GROUP.....:24
 NUMBER OF DRE GROUPS....................:1
C.......................................:
 FAILURE RATE OF CWS BANK................:+8.00000000E-03
C.......................................:
 TOTAL NUMBER OF CWS'S BANKS PER GROUP..:28
 NUMBER OF REQUIRED BANKS PER GROUP.....:25
 NUMBER OF CWS GROUPS....................:1
C.......................................:
 FAILURE RATE OF MASS STORAGE MODULE....:+1.50000000E-03
C.......................................:
 TOTAL NUMBER OF MSM'S PER GROUP....... :4
 NUMBER OF REQUIRED MSM'S PER GROUP.....:3
 NUMBER OF MSM GROUPS....................:1
```

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

CASE NO: 04

TMISS	RSYS	MTBF	AVAIL
1000.	0.9182491	1.1725180E 04	1.0000
2000.	0.8431815	1.1725180E 04	1.0000
3000.	0.7742507	1.1725180E 04	1.0000
4000.	0.7109550	1.1725180E 04	1.0000
5000.	0.6528338	1.1725180E 04	1.0000
6000.	0.5994641	1.1725180E 04	1.0000
7000.	0.5504574	1.1725180E 04	1.0000
8000.	0.5054570	1.1725180E 04	1.0000
9000.	0.4641355	1.1725180E 04	1.0000
10000.	0.4261920	1.1725180E 04	1.0000
11000.	0.3913504	1.1725180E 04	1.0000
12000.	0.3593572	1.1725180E 04	1.0000
13000.	0.3299794	1.1725180E 04	1.0000
14000.	0.3030033	1.1725180E 04	1.0000
15000.	0.2782325	1.1725180E 04	1.0000
16000.	0.2554868	1.1725180E 04	1.0000
17000.	0.2346005	1.1725180E 04	1.0000
18000.	0.2154217	1.1725180E 04	1.0000
19000.	0.1978108	1.1725180E 04	1.0000
20000.	0.1816396	1.1725180E 04	1.0000
21000.	0.1667904	1.1725180E 04	1.0000

*

Table 4-2 (continued) - Reliability and availability of typical G-471 computer system assuming different times between repair and different parts failure rates.

APPENDIX I

COMPUTATIONAL ASPECTS OF SIMPLE FILTERS

In this appendix we evaluate the computational complexity and storage requirements for simple filters. We assume that for a given filter its frequency characteristics are specified and that filtering is to be performed in time domain.

1. Low Pass, High Pass and Band Pass Filters

For these filters we assume that either the frequency components of the input are directly passed to the output (without attenuation or amplification and phase change) or completely suppressed. Thus for these we require a central working storage $n_{BCWS}=4N^2$ bytes for an input specified by an N X N real matrix. It is assumed that each real (complex) word takes 4 (8) bytes. Computationally we require $n_{AP}=8N^2\log_2 N$ real mult./sec. to perform Fourier and inverse Fourier transforms. These filters generate a byte traffic of $n_{BTCWS}=16N^2$ bytes between the central working storage and processors. Thus for these filters we have:

$$n_{BCWS} = 4N^2 \text{ bytes/frame}$$
$$n_{AP} = 8N^2\log_2 N \text{ bytes/frame}$$
$$n_{BTCWS} = 16N^2 \text{ bytes/frame}$$

2. Table Look-Up Filters

For these filters it is assumed that their frequency characteristics are specified by means of a table. In this case we need a working storage of $n_{BCWS}=12N^2$ bytes: $4N^2$ bytes for the real input array and $8N^2$ bytes for storing the table which contains information regarding the frequency characteristics of the filter. (Note in this situation the table may contain complex

entries). To implement the filter we have to perform two Fourier transforms and N^2 complex multiplications. Therefore, we require $n_{AP} = 4N^2 + 8N^2 \log_2 N$ real mult. In performing the filtering operation we generate a byte traffic of $n_{BTCWS} = 68N^2$ bytes: $16N^2$ bytes for performing the Fourier transform on the input, $20N^2$ for performing complex multiplications in frequency domain and $32N^2$ for performing the inverse Fourier transform. Thus for a table look-up filter:

$$n_{BCWS} = 12N^2 \text{ bytes/frame}$$

$$n_{AP} = 4N^2 + 8N^2 \log_2 N \text{ real mult./frame}$$

$$n_{BTCWS} = 68N^2 \text{ bytes/frame}$$

3. Homomorphic Filters for Multiplicative Processes

These filters are useful when the input is given by $I(x,y) = i(x,y).r(x,y)$ and it is desired to perform different filtering operations on i and r [1]. It is assumed that the input is real and by taking logarithms we have

$$\log_e I(x,y) = \log_e i(x,y) + \log_e r(x,y)$$

If i and r are predominantly low frequency and high frequency components respectively and we wish to deemphasize i and emphasize r, this can be achieved by passing $\log_e I$ through a filter whose gains in the low and high frequency spectrum are $\gamma_1 (<1)$ and $\gamma_2 (>1)$ respectively and performing an antilogarithm (exponentiation) on the output from the filter. The output can be approximately given as $i^{\gamma_1} r^{\gamma_2}$. This kind of filter, known as a homomorphic filter, is used in image enhancement.

We assume that the Fourier transform of the filtering function is Hermitian; this ensures that the output from the filter is real and simplifies exponentiation. To implement the filter

we require
$n_{BCWS} = 8N^2$bytes and $n_{AP} = 4N^2 + (k_{log} + k_{exp})N^2 + 8N^2 \log_2 N$ real mult. Here k_{log} (k_{exp}) is the ratio of the times to perform a logarithm (exponentiation) and a real multiplication. Byte traffic between the central working storage and processors can be determined as $n_{BTCWS} = 60N^2$bytes : $8N^2$ bytes for performing logarithms, $16N^2$ bytes for performing Fourier transform, $12N^2$ bytes for performing complex multiplication in frequency domain, $16N^2$ for performing inverse Fourier transform and $8N^2$bytes for exponentiation. Thus the complexity requirements of a homomorphic filter can be summarized as:

$$n_{BCWS} = 8N^2 \text{bytes/frame}$$

$$n_{AP} = 4N^2 + k_{log}N^2 + k_{exp}N^2 + 8N^2 \log_2 N \text{ real mult./frame}$$

$$n_{BTCWS} = 60N^2 \text{bytes/frame.}$$

Reference:

[1] Oppenheim, A.V., Schafer, R.W. and Stockham, T.G. "Nonlinear Filtering of Multiplied and Convolved Signals", Proc. IEEE, Vol. 56, pp. 1264-1291, August 1968.

(The reverse of this page is blank).

APPENDIX II

TWO-DIMENSIONAL IMAGE RESTORATION AND ENHANCEMENT FILTERS

Let $f(x,y)$ represent an image we wish to observe through a medium whose point spread function is given by $h(x,y)$ and which adds a noise component $\varepsilon(x,y)$. Then the observed image $g(x,y)$ can be given by:

$$g(x,y) = \int_{-\infty}^{\infty}\int h(x-w,y-z)f(w,z)dwdz + \varepsilon(x,y) \qquad \text{——(1)}$$

A basic problem in image restoration is to recover $f(x,y)$ from $g(x,y)$ by using suitable filters. There are many possible approaches to this problem [1] and in the following we consider the computational aspects of some of the filters reported in the literature.

1. Wiener Filters

Helstrom [2] proposed the least squared error approach to image restoration. His technique can be best explained by considering g, f and ε to be one-dimensional random variables and expressing the integral equation (1) in the matrix form as shown below:

$$g(x) = Hf(x) + \varepsilon(x) \qquad \text{——(2)}$$

Here g,f and ε are nx1 column matrices and H is a nxn matrix. The image restoration problem is solved by finding a matrix M such that M_g gives the restored image and the average error $E\{(f-Mg)^T(f-Mg)\}$ is minimized. It is assumed that f and ε are random (stochastic) processes whose correlation can be estimated. The restoration problem can be stated as:

Minimize $ER = E\{(f-Mg)^T(f-Mg)\}$
subject to $g = Hf + \varepsilon$.

By setting $\frac{\partial ER}{\partial M_{ij}} = E\{-2(hf+\varepsilon)^T J_{ji} f + 2(Hf+\varepsilon)^T J_{ji} M(Hf+\varepsilon)\} = 0$

where J_{ij} is an nxn matrix with all elements zero except for a 1 in its i th row and j th column, we obtain

$$E\{(Hf+\varepsilon)^T J_{ji} f\} = \{(Hf+\varepsilon)^T J_{ji} M(Hf+\varepsilon)\} \qquad \text{——(3)}$$

Let

$$W = \begin{bmatrix} E(f_1\varepsilon_1) & E(f_1\varepsilon_2) & \cdots & E(f_1\varepsilon_n) \\ E(f_2\varepsilon_1) & E(f_2\varepsilon_2) & \cdots & E(f_2\varepsilon_n) \\ \vdots & \vdots & & \vdots \\ E(f_n\varepsilon_1) & E(f_n\varepsilon_2) & & E(f_n\varepsilon_n) \end{bmatrix}$$

$$X = \begin{bmatrix} E(f_1^2) & E(f_1f_2) & \cdots & E(f_1f_n) \\ E(f_2f_1) & E(f_2^2) & \cdots & E(f_2f_n) \\ \vdots & \vdots & & \vdots \\ E(f_nf_1) & E(f_nf_2) & \cdots & E(f_n^2) \end{bmatrix}$$

and

$$Y = \begin{bmatrix} E(\varepsilon_1^2) & E(\varepsilon_1\varepsilon_2) & \cdots & E(\varepsilon_1\varepsilon_n) \\ E(\varepsilon_2\varepsilon_1) & E(\varepsilon_2^2) & \cdots & E(\varepsilon_2\varepsilon_n) \\ \vdots & & & \\ E(\varepsilon_n\varepsilon_1) & E(\varepsilon_n\varepsilon_2) & & E(\varepsilon_n^2) \end{bmatrix}$$

Then (3) reduces to:

$$X_i H_j^T + W_i^j = M_i[H(XH_j^T + W^j) + W^T H_j^T + Y^j]$$

where A_i, A^j and A_i^j represent the i th row, j th column and ij th element of matrix A. From the above expression M can be obtained as:

$$M = \frac{XH^T + W}{HXH^T + HW + W^TH^T + Y} \qquad \text{——(4)}$$

If we assume that the noise and image processes are independent of each other we can set W=0 and simplify (4) to:

$$M = \frac{XH^T}{HXH^T + Y} \qquad \text{——(5)}$$

The noise process is usually assumed to be white and so Y is diagonal. To estimate X let us consider the random variable sequence f_1, f_2, --, f_n. Assuming the sequence to be normal and finite-stationary we get the following density and joint-density functions [3]:

$$f(f_i) = \frac{1}{(2\pi\sigma_i^2)^{\frac{1}{2}}} e^{-(f_i-\mu_i)^2/2\sigma_i^2}$$

$$f(f_i, f_j) = \frac{1}{2\pi\sigma_i\sigma_j(1-r_{ij}^2)^{\frac{1}{2}}}$$

$$\exp\left\{-\frac{1}{2(1-r_{ij}^2)^{\frac{1}{2}}}\left[\frac{(f_i-\mu_i)^2}{\sigma_i^2} - \frac{2r_{ij}(f_i-\mu_i)(f_j-\mu_j)}{\sigma_i\sigma_j} + \frac{(f_j-\mu_j)^2}{\sigma_j^2}\right]\right\}$$

where $\mu_i = E(f_i)$, $i = 1,2,..,n$

and $\sigma_i^2 = E\{(f_i-\mu_i)^2\}$, $i = 1,2,..,n$.

r_{ij} is the correlation coefficient between the random variables f_i and f_j. If they are independent r_{ij} is zero. We can get from the above density functions the following:

$$E(f^i 2) = \sigma_i^2 + \mu_i^2$$

$$E(f_i f_j) = r_{ij}\sigma_i\sigma_j + \mu_i\mu_j.$$

It has been suggested that r_{ij} can be approximated by an exponential of the form $e^{-\beta|i-j|}$ [2].

The restored image

$$f=Mg=\frac{XH^T}{HXH^T+Y}\,g \qquad \text{——(6)}$$

can be evaluated by inverting the matrix (HXH^T+Y). (The evaluation can be simplified by assuming X and Y to be diagonal and their diagonal elements to be identical). It would be instructive to see how (6) appears in frequency domain. Applying Fourier transforms to the equation for g(x) we get

$$g(w) = h(w)\,f(w) + e(w)$$

When this equation is transformed into a matrix form we obtain

$$g(w) = H_F f(w) + e(w) \qquad \text{——(7)}$$

where H_F is a diagonal complex matrix and g, f and e are the Fourier transforms of g, f and e respectively and whose elements are arranged in column matrices. Since equation (7) is similar to (2) and the objective functions are related to each other by a constant (by Parseval's Theorem) the Fourier transform of the estimate of f(x) can be given as:

$$\delta = \frac{X_F H_F^T}{H_F X_F H_F^T + Y_F} \, g \qquad \text{——(8)}$$

where X_F and Y_F are matrices similar to X and Y and defined on the frequency components of f and ε.

$$X_F = \begin{bmatrix} E(\delta_1^2) & E(\delta_1\delta_2) & \text{----} & E(\delta_1\delta_n) \\ E(\delta_2\delta_1) & E(\delta_2^2) & \text{---} & E(\delta_2\delta_n) \\ \vdots & & & \\ E(\delta_n\delta_1) & E(\delta_n\delta_2) & \text{---} & E(\delta_n^2) \end{bmatrix}$$

$$Y_F = \begin{bmatrix} E(e_1^2) & E(e_1e_2) & \text{---} & E(e_1e_n) \\ E(e_2e_1) & E(e_2^2) & \text{---} & E(e_2e_n) \\ \vdots & \vdots & & \vdots \\ E(e_ne_1) & E(e_ne_2) & \text{---} & E(e_n^2) \end{bmatrix}$$

where $(\delta_1, \delta_2, \text{--}, \delta_n)$ and $(e_1, e_2, \text{--}, e_n)$ are the Fourier transforms of $(f_1, f_2, \text{--}, f_n)$ and $(\varepsilon_1, \varepsilon_2, \text{--}, \varepsilon_n)$ respectively. To evaluate the components of these matrices we proceed as follows.

$$\delta_k = \sum_{i=0}^{n-1} f_i W^{ik}, \quad k = 0, 1, \text{--}, n-1$$

$$e_k = \sum_{i=0}^{n-1} \varepsilon_i W^{ik}, \quad k = 0, 1, \text{--}, n-1$$

where $W = e^{-j\,2\pi/n}$. Hence

$$\delta_k \delta_l = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} f_i f_j W^{ik} W^{jl}$$

and

$$e_k e_l = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} \varepsilon_i \varepsilon_j W^{ik} W^{jl}$$

Therefore

$$E\{\delta_k \delta_l\} = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} E\{f_i f_j\} W^{ik} W^{jl}$$

$$E\{e_k e_l\} = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} E\{\varepsilon_i \varepsilon_j\} W^{ik} W^{jl}$$

If $E\{f_i f_j\} =$ for $i \neq j$ we get

$$E\{\delta_k \delta_l\} = \sum_{i=0}^{n-1} E\{f_i^2\} W^{i(k+l)}.$$

Similarly

$$E\{e_k e_l\} = \sum_{i=0}^{n-1} E\{\varepsilon_i^2\} W^{i(k+l)}.$$

Let δ^2 and e^2 represent the Fourier transforms of the sequences $(f_1^2, f_2^2, .., f_n^2)$ and $(\varepsilon_1^2, \varepsilon_2^2, .., \varepsilon_n^2)$ respectively. Then it can be seen that the i,j th element of $X_F(Y_F)$ is the (i+j) (mod n) th element of δ^2 (e^2). X_F and Y_F can be noted to be circulant matrices. In equation (8) though H_F and H_F^T are diagonal matrices still we have to invert the matrix $(H_F X_F H_F^T + Y_F)$ by conventional methods. If this inversion takes of $O(n^3)$ operations for a nxn matrix the least squares filter is clearly infeasible for large n.

If f, g and ε are assumed to be continuous stochastic processes it can be shown that the restoring filter can be given in frequency domain as:

$$m(w) = \frac{\bar{h}(w)}{h(w)\bar{h}(w) + \dfrac{\Phi_\varepsilon(w)}{\Phi_f(w)}}$$

and the restored image as:

$$\hat{f}(w) = \frac{\bar{h}(w)}{|h(w)|^2 + \dfrac{\Phi_\varepsilon(w)}{\Phi_f(w)}}\, g(w) \qquad \text{——(9)}$$

For a derivation of the above result see Helstrom [2]. Here h, g, $\hat{f}$ are the (stochastic) Fourier transforms of h, g and f respectively; and Φ_ε and Φ_f are the power spectra of the image and noise processes. A discrete version of equation (9) can be used instead of (8) because of the former's computational simplicity. (Note that equation (8) gives the optimal restored image whereas (9) usually does not.)

Two dimensional image functions can be handled as follows. Let g(x,y) be specified by G, an mxm matrix, and assume for convenience $0 \le x \le m-1$, $0 \le y \le m-1$ and x and y to be integers. Then the rows of G can be arranged into a column as shown below:

$$\begin{bmatrix} G_1 \\ G_2 \\ \cdot \\ \cdot \\ G_m \end{bmatrix} \qquad \begin{bmatrix} G_1^T \\ G_2^T \\ \\ \\ G_m^T \end{bmatrix}$$

Proceeding this way h and ε can be arranged as columns and the two dimensional restoration equation can be written in the form of (2). As an example let g, h, and ε be specified by 3x3 matrices G, H and E respectively as in the following:

$$G = \begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & g(1,2) \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

$$H = \begin{bmatrix} h(0,0) & h(0,1) & h(0,2) \\ h(1,0) & h(1,1) & h(1,2) \\ h(2,0) & h(2,1) & h(2,2) \end{bmatrix}$$

and

$$E = \begin{bmatrix} \varepsilon(0,0) & \varepsilon(0,1) & \varepsilon(0,2) \\ \varepsilon(1,0) & \varepsilon(1,1) & \varepsilon(1,2) \\ \varepsilon(2,0) & \varepsilon(2,1) & \varepsilon(2,2) \end{bmatrix}$$

Let

$$F = \begin{bmatrix} f(0,0) & f(0,1) & f(0,2) \\ f(1,0) & f(1,1) & f(1,2) \\ f(2,0) & f(2,1) & f(2,2) \end{bmatrix}$$

$$g(x,y) = \sum_{y_1=0}^{y} \sum_{x_1=0}^{x} h(x-x_1, y-y_1)\, f(x_1, y_1) + \varepsilon(x,y)$$

$$g(0,0) = h(0,0).f(0,0) + \varepsilon(0,0)$$

$$g(0,1) = h(0,1).f(0,0) + h(0,0)f(0,1) + \varepsilon(0,1)$$

.....

$$g(1,1) = h(1,1)f(0,0) + h(1,0)f(0,1) + h(0,1)f(1,0) + h(0,0)f(1,1) + \varepsilon(1,1)$$

$$g(1,2) = h(1,2)f(0,0) + h(1,1)f(0,1) + h(1,0)f(0,2) + h(0,2)f(1,0) + h(0,1)f(1,1) + h(0,0)f(1,2) + \varepsilon(1,2)$$

. . . . . .

$$
\begin{aligned}
g(2,2) = \quad & h(2,2)f(0,0) + h(2,1)f(0,1) \\
& + h(2,0)f(0,2) + h(1,2)f(1,0) \\
& + h(1,1)f(1,1) + h(1,0)f(1,2) \\
& + h(0,2)f(2,0) + h(0,1)f(2,1) \\
& + h(0,0)f(2,2) \qquad\qquad\qquad\qquad + \varepsilon(2,2)
\end{aligned}
$$

$$
g = \left[\begin{array}{ccc|ccc|ccc}
h(0,0) & 0 & 0 & & & & & & \\
h(0,1) & h(0,0) & 0 & & \bigcirc & & & \bigcirc & \\
h(0,2) & h(0,1) & h(0,0) & & & & & & \\
\hline
h(1,0) & 0 & 0 & h(0,0) & 0 & 0 & & & \\
h(1,1) & h(1,0) & 0 & h(0,1) & h(0,0) & 0 & & \bigcirc & \\
h(1,2) & h(1,1) & h(1,0) & h(0,2) & h(0,1) & h(0,0) & & & \\
\hline
h(2,0) & 0 & 0 & h(1,0) & 0 & 0 & h(0,0) & 0 & 0 \\
h(2,1) & h(2,0) & 0 & h(1,1) & h(1,0) & 0 & h(0,1) & h(0,0) & 0 \\
h(2,2) & h(2,1) & h(2,0) & h(2,1) & h(1,1) & h(1,0) & h(0,2) & h(0,1) & h(0,0)
\end{array}\right] f+\varepsilon
$$

$$
g = [g(0,0)\ g(0,1)\ g(0,2)\ \ldots\ldots\ldots g(2,0)\ g(2,1)\ g(2,2)]^T
$$

$$
f = [f(0,0)\ f(0,1)\ f(0,2)\ \ldots\ldots\ldots f(2,0)\ f(2,1)\ f(2,2)]^T
$$

$$
\varepsilon = [\varepsilon(0,0)\ \varepsilon(0,1)\ \varepsilon(0,2)\ \ldots\ldots\ldots \varepsilon(2.0)\ \varepsilon(2,1)\ \varepsilon(2,2)]^T
$$

$$
H = \begin{bmatrix} \tilde{H_1} & 0 & 0 \\ \tilde{H_2} & \tilde{H_1} & 0 \\ \tilde{H_3} & \tilde{H_2} & \tilde{H_1} \end{bmatrix}
$$

$$
H_1 = \begin{bmatrix} h(0,0) & 0 & 0 \\ h(0,1) & h(0,0) & 0 \\ h(0,2) & h(0,1) & h(0,0) \end{bmatrix}
$$

$$
H_2 = \begin{bmatrix} h(1,0) & 0 & 0 \\ h(1,1) & h(1,0) & 0 \\ h(1,2) & h(1,1) & h(1,0) \end{bmatrix}
$$

$$H_3 = \begin{bmatrix} h(2,0) & 0 & 0 \\ h(2,1) & h(2,0) & 0 \\ h(2,2) & h(2,1) & h(2,0) \end{bmatrix}$$

Equation (9) becomes in two dimensions:

$$\hat{f}(u,v) = \frac{h(u,v)}{|h(u,v)|^2 + \dfrac{\Phi_\varepsilon(u,v)}{\Phi_f(u,v)}} g(u,v) \qquad \text{——(10)}$$

COMPUTATIONAL COMPLEXITY OF WIENER FILTERS

We estimate the computational complexity of a filter in terms of the number of multiplications and byte storage it requires for its implementation. For our estimation we assume equation (10) and the following:

(A1) g, the observed image, is given by an N/2 X N/2 real matrix

(A2) h, the point spread function, is given by an N/4 X N/4 real matrix

and (A3) each real (complex) word takes 4(8) bytes.

Input for a Wiener filter consists of g,h and NXN real matrix specifying Φ_ε/Φ_f. (Note that Φ_ε/Φ_f is usually symmetric and does not require storage for N^2 real words). g and h, finite Fourier transforms of f and g, require $4N^2$ bytes each since they are Hermitian. Thus:

Input:

- g : N/2 X N/2 (real) $\supset$ $*N^2$ bytes
- h : N/4 X N/4 (real) $\supset$ $N^2/4$ bytes
- Φ_ε/Φ_f · N X N (real) $\supset$ $\lambda_1 4N2$ bytes

Working Storage:

- g $\rightarrow$ g: N X N (Hermitian) $\supset$ $4N^2$ bytes
- h $\rightarrow$ h: N X N (Hermitian) $\supset$ $4N^2$ bytes
- Φ_ε/Φ_f : $\lambda_1 4N^2$ bytes

The factor λ_1 reflects the sparsity and symmetry of Φ_f/Φ_ε; in practice it is ½ and can be as low as of $O(1/N^2)$. Since f and g can be stored in common locations we require a working storage

$$n_{BCWS} = 8N^2 + \lambda_1 4N^2 \text{ bytes.}$$

* In the sequel $\supset$ stands for "requires"

Since it takes at most $4N^2\log_2 N$ real multiplications to perform a Fourier transform we have:

$$g \rightarrow g \quad \supset 4N^2\log_2 N \text{ real mult.}$$

$$h \rightarrow h \quad \supset 4N^2\log_2 N \text{ real mult.}$$

$$f \rightarrow f \quad \supset 4N^2\log_2 N \text{ real mult.}$$

h(Hermitian), g(Hermitian) $\rightarrow$

$$\bar{h}\, g \quad \supset 2N^2 \text{ real mult.}$$

$$h \rightarrow \bar{h}\, h \quad \supset N^2 \text{ real mult.}$$

$\bar{h}\, g$ (Hermitian), $|h|^2$ (real, symmetric),

$$\Phi_\varepsilon/\Phi_f \text{ (real)} \rightarrow \frac{\bar{h}g}{|h|^2 + \Phi_\varepsilon/\Phi_f} \quad \supset 2K_{div}N^2 \text{ real mult.}$$

Here k_{div} is the ratio of the time to perform a real multiplication to the time to perform a real division. Since asymmetry in Φ_ε/Φ_f results in imaginary components in the restored image, it is usually taken to be symmetric in which case the above computation takes only $k_{div}N^2$ real multiplications. Hence the computational complexity of the filter can be given as $n_{AP} = N^2(12\log_2 N + 3 + \lambda_2 k_{div})$ real multiplications where $\lambda_2 \varepsilon \{1,2\}$. The value of λ_2 in practice is usually 1.

The traffic generated between the working storage and the processing elements is another measure of the complexity of the filtering algorithm. To perform a Fourier transform on g, we need to transfer the elements of g to the processing elements (N^2 bytes), perform FFT on one dimension, transfer to the working storage ($4N^2$ bytes), transfer from the working storage to the processing

elements to perform FFT on the other dimension ($4N^2$ bytes) and store the entire Fourier transform of g in the working storage ($4N^2$ bytes). Thus a net traffic of $13N^2$ bytes is generated between the processing elements and the working storage. Proceeding this way we have:

$$g \rightarrow g \supset 13N^2 \text{ bytes}$$

$$h \rightarrow h \supset 12.25N^2 \text{ bytes}$$

$$h, g, \Phi_\varepsilon/\Phi_f \rightarrow \frac{\bar{h}g}{|h|^2 + \Phi_\varepsilon/\Phi_f} \supset 12N^2 + (\lambda_1+\lambda_2)\, 4N^2 \text{ bytes}$$

$$\mathfrak{f} \rightarrow f \supset \lambda_2 16N^2 \text{bytes}$$

Thus the net traffic generated in performing the filter is $n_{BTCWS} = (37.25N^2 + \lambda_1 4N^2 + \lambda_2 20N^2)$ bytes.

Therefore for a Wiener filter we have

$$n_{BCWS} = 8N^2 + \lambda_1 4N^2 \text{ bytes/frame}$$

$$n_{AP} = N^2(12\log_2 N + 3 + \lambda_2 k_{div}) \text{ real mult./frame}$$

$$n_{BTCWS} = N^2(37.25 + 4\lambda_1 + 20\lambda_2) \text{ bytes/frame.}$$

λ_1 is usually 1/2 and can be as low as $O(1/N^2)$.

λ_2 is usually 1 and can be as high as 2.

2. CONSTRAINED LEAST SQUARES FILTERS [4, 5, 6]

For the matrix equation (2) Phillips [4] developed a technique to find an estimate $\hat{f}(x)$ for the solution $f(x)$. His approach is to find $\hat{f}(x)$ such that it minimizes:

$$\hat{f}^T C^T C \hat{f} \qquad \text{——(11)}$$

subject to $(g-H\hat{f})^T(g-H\hat{f}) = \varepsilon^T\varepsilon$ ——(12)

Matrix C is chosen such that:

$$f^T C^T C \hat{f} = \sum_{i=0}^{n-1} (\hat{f}_{i+1} - 2f_i + \hat{f}_{i-1})^2$$

i.e.
$$C = \begin{bmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

The reason for selecting (11) as an objective is to ensure that the estimate $f(x)$ is smooth and contains minimum amount of unwanted oscillations which are caused by the noise term $\varepsilon(x)$. To solve the minimization problem form the Lagrangian:

$$\phi = f^T C^T C f + \lambda\{(g-Hf)^T(g-Hf) - \varepsilon^T\varepsilon\}$$

and set $\frac{\partial \phi}{\partial f_i} = 0$, $i = 0, 1, .., n-1$. Here λ is a Lagrange multiplier.

$$\frac{\partial \phi}{\partial f} = 2C^T C f + g\{(g-Hf)^T \cdot \frac{\partial}{\partial f}(g-Hf) + (\frac{\partial}{\partial f}(g-Hf)^T)\ (g-Hf)\}$$

$$= 2C^T C f + \lambda\{-2H^T g + 2H^T H f\}$$

Hence $\hat{f} = \dfrac{H^T g}{H^T H + \gamma C^T C}$ ——(13)

where γ is $1/\lambda$.

Equation (13) is solved iteratively by selecting different values for γ until constraint (12) is satisfied. It may be noted that to find f from (13) one matrix inversion is needed for each iteration. Hunt [6] showed that when H represents a convolution equation (13) can be converted into an equivalent equation in frequency domain which is more amenable for computation. This equivalent equation can be given as:

$$\hat{\mathcal{f}}(w) = \frac{\bar{h}(w)g(w)}{|h(w)| + \gamma|c(w)|^2} \qquad \text{——(14)}$$

where $\mathcal{f}(w)$, $g(w)$, $h(w)$ and $c(w)$ are Fourier transforms of $f(x)$, $g(x)$, $h(x)$ and $c(x) = \{1, -2, 1, \ldots\}$ respectively. As may be noted equation (14) requires a division instead of a matrix inversion for each iteration. Equations (13) and (14) can be extended to two dimensions by following the same procedure explained for Wiener filters. Equation (14) becomes in two dimensions:

$$\hat{\mathcal{f}}(u,v) = \frac{\bar{h}(u,v)g(u,v)}{|h(u,v)|^2 + \gamma|c(u,v)|^2} \qquad \text{——(15)}$$

See Hunt [6] for different methods of selecting c for two dimensions. It is possible to develop an explicit nonlinear equation for γ that gives $\hat{f}$ directly. For one dimensional case:

Let $\alpha_i = |g(i)|^2$, $\beta_i = |h(i)|^2/|c(i)|^2$ and $\delta = n \sum_{i=0}^{n-1} \varepsilon_i^2$. Then $\lambda = 1/\gamma$ satisfies the following non-linear equation:

$$\sum_i \frac{\alpha_i}{(1 + \beta_i \lambda)^2} = \delta \qquad \text{———(16)}$$

where i is such that $c(i) \neq 0$. For a proof see Appendix 3. Equation (16) can be easily generalized to two dimensions.

COMPUTATIONAL CONSIDERATIONS OF CONSTRAINED LEAST SQUARES FILTERS

For our estimation of computational complexity of constrained least squares filters we assume that equation (15) is used to find δ iteratively by selecting different values for γ until constraint (12) is satisfied. (This was the approach presented by Hunt [6].) We also make assumptions (A1), (A2) and (A3) stated in Section 1.1.

Input for a constrained least-squares filter consists of:

g:N/2XN/2 (real)	$\supset N^2$ bytes	Input
h:N/4XN/4 (real)	$\supset N^2/4$ bytes	
c:NXN (real)	$\supset 4N^2$ bytes	

We need a working storage of $n_{BCWS} = 15N^2$ bytes since:

g:N/2XN/2 (real)	$\supset N^2$ bytes	Working Storage
$h \rightarrow h$:NXN (Hermitian)	$\supset 4N^2$ bytes	
$\bar{h}h/c\bar{c}$:NXN (real, sym.)	$\supset 2N^2$ bytes	
$\bar{h}g/c\bar{c}$:NXN (Hermitian)	$\supset 4N^2$ bytes	
$\hat{\delta} \rightarrow \hat{f}$:NXN (real)	$\supset 4n^2$ bytes	

Let us estimate the complexity of the filter in terms of real multiplications it requires for its implementation.

$(g \to g,\ h \to h,\ c \to c) \supset 12N^2 \log_2 N$ real mult.

$c \to c\bar{c} \supset N^2$ real mult.

$h\bar{h}/c\bar{c} \supset N^2 + k_{div}N^2/2$ real mult.

$\bar{h}g/c\bar{c} \supset 2N^2 + k_{div}N^2$ real mult.

For each iteration:

$$\frac{\bar{h}g/c\bar{c}}{h\bar{h}/c\bar{c} + \gamma} \supset 2k_{div}N^2/2 = k_{div}N^2 \text{ real mult.}$$

$h\hat{f} \supset 2N^2$ real mult.

$h\hat{f} \to h*f \supset 4N^2 \log_2 N$ real mult.

$(g-Hf)^T(g-Hf) \supset N^2$ real mult.

At the end of iterations:

$\hat{f} \to f: 4N^2 \log_2 N$ real mult.

Hence the implementation of the filter as suggested by Hunt takes approximately

$$n_{AP} = 16N^2 \log_2 N + 4N^2 + k_{div}3N^2/2 + p(4N^2 \log_2 N + 3N^2 + k_{div}N^2) \text{ real mult.}$$

where p is the number of iterations required to find the final filter (observed to be between 3 and 12 [6]).

The traffic generated between the working storage and processing elements can be estimated as follows:

$(g \to g,\ h \to h,\ c \to c) \quad \supset 13N^2 + 12.25N^2 + 16N^2$

$= 41.25N^2$ bytes

$c \to |c|^2 \quad \supset 4N^2 + 2N^2 = 6N^2$ bytes

$h\bar{h}/c\bar{c} \quad \supset 4N^2 + 2N^2 + 2N^2 = 8N^2$ bytes

$\bar{h}g/c\bar{c} \quad \supset 4N^2 + 4N^2 + 2N^2 + 4N^2 = 14N^2$ bytes

For each iteration:

$\dfrac{\bar{h}g/c\bar{c}}{h\bar{h}/c\bar{c} + \gamma} \quad \supset 2N^2 + 4N^2 + 4N^2 = 10N^2$ bytes

$h\phi \quad \supset 4N^2 + 4N^2 + 4N^2 = 12N^2$ bytes

$h\phi \to h*f = Hf \quad \supset 16N^2$ bytes

$(g-Hf)^T(g-Hf) \quad \supset 5N^2$ bytes

At the end of iterations:

$\phi \to f \quad \supset 16N^2$ bytes.

Hence the net traffic generated in using the filter is $n_{BTCWS} = 85.25N^2 + p(43N^2)$ where p is the number of iterations required to find the final filter.

Let us consider the alternate method where $\hat{\gamma}$ is determined using (16) and $\hat{f}$ is found directly (without iterations) from this value of γ. We need a working storage of $n_{BCWS} = 12N^2$ bytes because:

$g \rightarrow g \quad \supset 4N^2$ bytes

$h \rightarrow h \quad \supset 4N^2$ bytes

$c \rightarrow c \quad \supset 4N^2$ bytes

$g,h,c \rightarrow \bar{h}g/c\bar{c} \quad \supset 4N^2$ bytes

$g \rightarrow g\bar{g} \quad \supset 2N^2$ bytes

$h,c \rightarrow h\bar{h}/c\bar{c} \quad \supset 2N^2$ bytes

} Working Storage

The number of real multiplications required for this method can be determined as follows:

$\{g \rightarrow g,\ h \rightarrow h,\ c \rightarrow c\} \supset 12N^2 \log_2 N$ real mult.

$c \rightarrow |c|^2 \quad \supset N^2$ real mult.

$g \rightarrow |g|^2 \quad \supset N^2$ real mult.

$\bar{h}g/|c|^2 \quad \supset 2N^2 + k_{div}N^2$ real mult.

$h\bar{h}/c\bar{c} \quad \supset N^2 + k_{div}N^2/2$ real mult.

In each iteration (for determining γ):

$$\sum_{u,v} \frac{|g(u,v)|^2}{(1 + \hat{\lambda}|h(u,v)|^2/|c(u,v)|^2)^2} = \delta = n\varepsilon^T\varepsilon$$

$\supset N^2 + k_{div}N^2$ real mult.

After determining $\hat{\gamma}$ we need:

$$\delta = \frac{\bar{h}g/c\bar{c}}{h\bar{h}/c\bar{c} + \gamma} \quad \supset k_{div}N^2 \text{ real mult.}$$

$$\delta \rightarrow \hat{f} \quad \supset 4N^2\log_2 N \text{ real mult.}$$

Hence this method requires $n_{AP} = 16N^2\log_2 N + 5N^2 + 2.5k_{div}N^2 + p(N^2 + k_{div}N^2)$ real mult.

Byte traffic generated by considering the alternate method can be determined as follows:

$\{g \rightarrow g, h \rightarrow h, c \rightarrow c\}$ $\supset 41.25N^2$ bytes

$c \rightarrow |c|^2$ $\supset 6N^2$ bytes

$g \rightarrow |g|^2$ $\supset 6N^2$ bytes

$h\bar{h}/c\bar{c}$ $\supset 8N^2$ bytes

$\bar{h}g/c\bar{c}$ $\supset 14N^2$ bytes

For each iteration:

$$\sum \frac{|g|^2}{(1 + \hat{\lambda}|h|^2/|c|^2)^2} \quad \supset 4N^2 \text{ bytes}$$

At the end of iterations:

$$\delta = \frac{\bar{h}g/c\bar{c}}{h\bar{h}/c\bar{c} + \gamma} \quad \supset 10N^2 \text{ bytes}$$

$\hat{\delta} \rightarrow \hat{f}$ $\supset 16N^2$ bytes.

Hence the net traffic generated in this method is $n_{BTCWS} = 101.25N^2 + p(4N^2)$ bytes where p is the number of iterations required to find γ.

Hence for a constrained least squares filter we obtain:

Case 1: For iterative determination of f:

$$n_{BCWS} = 15N^2 \text{ bytes/frame}$$

$$n_{AP} = 16N^2\log_2 N + 4N^2 + k_{div}3N^2/2 + p_1(4N^2\log_2 N + 3N^2 + k_{div}N^2) \text{ real mult./frame}$$

$$n_{BTCWS} = 85.25N^2 + p_1(43N^2) \text{ bytes/frame}$$

p_1: number of iterations to obtain f.

Case 2: For iterative determination of γ:

$$n_{BCWS} = 12N^2 \text{ bytes/frame}$$

$$n_{AP} = 16N^2\log_2 N + 5N^2 + 2.5k_{div}N^2 + p_2(N^2 + k_{div}N^2) \text{ real mult./frame}$$

$$n_{BTCWS} = 101.25N^2 + p_2(4N^2) \text{ bytes/frame}$$

p_2: number of iterations required to find γ.

Let us compare cases 1 and 2 for $p_1 = p_2 = 8$. We have:

Case 1: $n_{BCWS} = 15N^2$ bytes/frame

$$n_{AP} = (48N^2\log_2 N + 28N^2 + 9.5k_{div}N^2) \text{ real mult./frame}$$

$$n_{BTCWS} = 429.25N^2 \text{ bytes/frame}$$

Case 2: $n_{BCWS} = 12N^2$ bytes/frame

$n_{AP} = (16N^2 \log_2 N + 13N^2 + 10.5k_{div}N^2)$ real mult./frame

$n_{BTCWS} = 133.25N^2$ bytes/frame.

3. PARAMETRIC WIENER FILTERS [7]

For these filters it is assumed that the image process f and noise process ε are stationary random processes with covariance matrices R_f and R_ε respectively. To obtain a parametric Wiener filter the following optimization problem is solved:

$$\text{Minimize} \quad f^T R_f^{-1} f \qquad \text{——(17)}$$

$$\text{Subject to } (g-Hf)^T R_\varepsilon^{-1} (g-Hf) = C \qquad \text{——(18)}$$

Applying the Lagrangian minimization technique we get the solution to the above problem as:

$$f = \frac{H^T R_\varepsilon^{-1}}{H^T R_\varepsilon^{-1} H + \gamma R_f^{-1}} \; g \qquad \text{——(19)}$$

where γ is a Lagrangian multiplier. In frequency domain the above equation becomes in two dimensions [7]:

$$\mathcal{f}(u,v) = \frac{h(u,v)\; g(u,v)}{|h(u,v)|^2 + \gamma \dfrac{\Phi_\varepsilon(u,v)}{\Phi_f(u,v)}} \qquad \text{——(20)}$$

where Φ_ε and Φ_f are the power spectra of ε and f obtained from covariance matrices R_ε and R_f. As usual $\mathcal{f}$, h and g are the Fourier transforms of f, h and g respectively.

COMPUTATIONAL ASPECTS OF PARAMETRIC WIENER FILTERS

We assume equation (20) and estimate the computational complexty and storage requirements for a parametric Wiener filter. The inputs to the filter are assumed to be g, an N/2XN/2 real matrix, h, an N/4XN/4 real matrix, and $\Phi = \Phi_\varepsilon / \Phi_f$, an NXN real matrix. The filter for its

implementation needs a working storage of $\lambda_2 6N^2 + 9N^2$ bytes since:

$\bar{h}g/\Phi \supset \lambda_2 4N^2$ bytes

$h\bar{h}/\Phi \supset \lambda_2 2N^2$ bytes

$h \rightarrow h \supset 4N^2$ bytes

$g \rightarrow \hat{g} \supset r4N^2$ bytes

$g \supset N^2$ bytes

} Working Storage

Here λ_2 reflects the symmetry of Φ; $\lambda_2 = 1$ if ϕ is symmetric. Otherwise it is 2.

Computationally we require:

$(g \rightarrow g,\ h \rightarrow h) \supset 8N^2 \log_2 N$ real mult.

$\bar{h}g/\Phi \supset 2N^2 + \lambda_2 k_{div} N^2$ real mult.

$h\bar{h}/\Phi \supset N^2 + \lambda_2 k_{div} N^2/2$ real mult.

In each iteration (for determining γ) we require

$\lambda_2 k_{div} N^2 + 3N^2 + 4N^2 \log_2 N$ real mult.

as in a constrained least square filter.

At the end of iterations:

$\hat{g} \rightarrow \hat{f} \supset 4N^2 \log_2 N$ real mult.

Thus the filter requires $n_{AP_2} = 12N^2 \log_2 N + 3N^2 + \lambda_2 k_{div} 3N^2/2 + p(4N^2 \log_2 N + 3N^2 + \lambda_2 k_{div} N^2)$ real mult. where p is the number of iterations to find the final filter.

Byte traffic can be decided as follows:

$(g \to g,\ h \to h) \quad \supset 25.25N^2$ bytes

$\bar{h}g/\Phi \quad \supset \lambda_1 4N^2 + 4N^2 + 4N^2 + \lambda_2 4N^2 =$

$$8N^2 + (\lambda_1 + \lambda_2)4N^2 \text{ bytes}$$

$h\bar{h}/\Phi \quad \supset \lambda_1 4N^2 + 4N^2 + \lambda_2 2N^2$ bytes

In each iteration:

$$\delta = \frac{\bar{h}g/\Phi}{h\bar{h}/\Phi + \gamma} \supset \lambda_2 10N^2$$

$h\hat{\delta} \quad \supset 12N^2$

$h\hat{\delta} \to h*\hat{f} = H\hat{f} \quad \supset 16N^2$

$(g-Hf)^T(g-Hf) \quad \supset 5N^2$

At the end of iterations:

$\hat{\delta} \to f \quad \supset 16N^2$

Hence the byte traffic generated can be given for this filter as $n_{BTCWS} = 53.25N^2 + \lambda_1 8N^2 + \lambda_2 6N^2 + p(\lambda_2 10N^2 + 33N^2)$ bytes. λ_1, as before, reflects the symmetry and sparsity of Φ.

Thus the computational complexity and storage requirements of the filter can be summarized as:

$$n_{BCWS} = \lambda_2 6N^2 + 9N^2 \text{ bytes/frame}$$

$$n_{AP} = 12N^2\log_2 N + 3N^2 + \lambda_2 k_{div} 3N^2/2 + p(4N^2\log_2 N + 3N^2 + \lambda_2 k_{div} N^2) \text{ real mult./frame}$$

$$n_{BTCWS} = 53.25N^2 + \lambda_1 8N^2 + \lambda_2 6N^2 + p(\lambda_2 10N^2 + 33N^2) \text{ bytes/frame}$$

λ_1 is usually 1/2 and can be as low as $O(1/N^2)$.

λ_2 is usually 1 and can be as high as 2.

4. COLE-STOCKHAM'S HOMOMORPHIC FILTERS [7, 8]

Cole and Stockham [8] suggest the following estimate for f(x,y) in the frequency domain:

$$\hat{f}(u,v) = \left\{ \frac{1}{|h(u,v)|^2 + \frac{\Phi_\varepsilon(u,v)}{\Phi_f(u,v)}} \right\}^{1/2} g(u,v) \qquad \text{——(21)}$$

storage requirements for the filter can be determined as follows:

g:N/2XN/2 (real) $\supset N^2$ bytes

h:N/4XN/4 (real) $\supset N^2/4$ bytes

$\Phi = \Phi_\varepsilon/\Phi_f$:NXN (real) $\supset \lambda_1 4N^2$ bytes

} Input

g → g:NXN (Hermitian) $\supset 4N^2$ bytes

h → h → $|h|^2$:NXN (real, sym.) $\supset 2N^2$ bytes

Φ $\supset \lambda_1 4N^2$ bytes

} Working Storage

Thus a working storage of $n_{BCWS} = 6N^2 + \lambda_1 4N^2$ bytes is needed where λ_1 is a factor which reflects the sparsity and symmetry of Φ. Computationally it requires $n_{AP} = 12N^2\log_2 N + N^2 + \lambda_2(N^2/2)(k_{div} + k_{sqrt} + 2)$ real mult. where $\lambda_2 = 1$ when Φ is symmetric; otherwise $\lambda_2 = 2$. Usually $\lambda_2 = 1$ and $\lambda_1 = 1/2$ and λ_1 may go as low as $O(1/N^2)$. k_{sqrt} is the ratio of the time to perform a square-root to the time to perform a real multiplication. The filter generates a byte traffic of $n_{BTCWS} = 37.25N^2 + \lambda_1 4N^2 + \lambda_2 20N^2$ bytes. Thus the complexity and requirements of the filter can be summarized as:

$n_{BCWS} = 6N^2 + \lambda_1 4N^2$ bytes/frame

$n_{AP} = 12N^2\log_2 N + N^2 + \lambda_2 N^2/2(k_{div}+k_{sqrt}+2)$ real mult./frame

$$n_{BTCWS} = 37.25N^2 + \lambda_1 4N^2 + \lambda_2 20N^2 \text{ bytes/frame}$$

λ_1 is usually 1/2 and can be as low as $O(1/N^2)$

λ_2 is usually 1 and can be as high as 2.

References for Appendix II

[1] Sondhi, M.M., "Image Restoration: The Removal of Spatially Invariant Degradations", Proc. IEEE, Vol. 60, pp. 842-853, 1972.

[2] Helstrom, C.W., "Image Restoration by the Method of Least Squares", J. Optical Soc. of America, Vol. 57, pp. 297-303, 1967.

[3] Papoulis, A., Probability, Random Variables and Stochastic Processes, McGraw-Hill, N.Y., 1965.

[4] Phillips, D.L., "A Technique for the Numerical Solution of Certain Integral Equations of the First Kind", J. Assoc. Comp. Mach., Vol. 9, pp. 84-97, January 1962.

[5] Twomey, S., "On the Numerical Solution of Fredholm Integral Equation of the First Kind by the Inversion of the Linear System Produced by Quadrature", J. Assoc. Comp., Mach., Vol. 10, pp. 97-101, January 1963.

[6] Hunt, B.R., "The Application of Constrained Least Squares Estimation to Image Restoration by Digital Computer", IEEE Trans. on Computers, Vol. C-22, pp. 805-812, September 1973.

[7] Hunt, B.R. and Andrews, H.C., "Comparison of Different Filter Structures for Image Restoration", Proc. 6th Ann. Hawaii Int. Conf. on Systems Sciences, January 1973.

[8] Cole, E.R., "The Removal of Unknown Image Blurs by Homomorphic Filtering", Report No. UTEC-CSc-74-029, Computer Science Div., University of Utah, Salt Lake City, Utah, June 1973.

(The reverse of this page is blank).

APPENDIX III

A NOVEL IMPLEMENTATION SCHEME FOR LEAST SQUARES FILTERS

Phillips [1] and Twomey [2] presented techniques to solve the integral equation:

$$\int_a^b k(x,y)f(y)dy = g(x) + \varepsilon(x)$$

when expressed as a quadrature in the matrix form:

$$A. \begin{bmatrix} f_0 \\ f_1 \\ . \\ . \\ f_i \\ . \\ . \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} g_0 \\ g_1 \\ . \\ g_i \\ . \\ . \\ g_{n-1} \end{bmatrix} + \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ . \\ . \\ \varepsilon_i \\ . \\ . \\ \varepsilon_{n-1} \end{bmatrix} \qquad \text{——(1)}$$

Here A is the matrix of quadrature coefficients for the discretized points $y_0, y_1, y_2 .., y_{n-1}$ and $x_0, x_1, x_2, .., x_{n-1}$; $f_i = f(y_i)$, $g_i = g(x_i)$ and $\varepsilon_i = \varepsilon(x_i)$. $\varepsilon_{(x)}$ represents errors in observing g(x) and is a random variable.

Phillips developed a procedure to find a solution $f = [\hat{f}_0, \hat{f}_1, .., \hat{f}_{n-1}]^T$ to the matricial equation (1) under the following conditions:

$$\text{Minimize} \sum_{i=0}^{n-1} (\hat{f}_{i+1} - 2\hat{f}_i + \hat{f}_{i-1})^2 \qquad \text{——(2a)}$$

$$\text{subject to } [Af-g]^T[Af-g] = \sum_{i=0}^{n-1} \varepsilon_i^2 = e \qquad \text{——(2b)}$$

His procedure is iterative, and requires the matrix inversion of A and one matrix inversion per iteration. Twomey improved his technique so that the solution procedure requires one matrix inversion per iteration and eliminates the inversion of A. Twomey's solution f is given as:

$$f = (A^TA + \hat{\gamma}H)^{-1}A^Tg \qquad \text{——(3)}$$

where $\hat{\gamma}$ is chosen iteratively to satisfy (2b) and H =

$$\begin{bmatrix} 1 & -2 & 1 & 0 & 0 & 0 & . & . & . \\ -2 & 5 & -4 & 1 & 0 & 0 & . & . & . \\ 1 & -4 & 6 & -4 & 1 & 0 & . & . & . \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & . & . \\ . & . & . & . & . & . & . & . & . \end{bmatrix}$$

Hunt [3] showed that if the integral in the integral equation represents a convolution (i.e., k(x,y) = k(x-y)),matrix inversion in (3) can be reduced to a scalar division by going into frequency domain. In each iteration his method requires a computational complexity of $O(n\log_2 n)$ operations. In this note we present a procedure whereby one can determine f directly without any iterations; we develop a non-linear equation for γ in terms of the frequency components of k and g from which $\hat{\gamma}$ that corresponds to $\hat{f}$ can be decided iteratively. Thus each iteration requires a complexity of O(n) operations. When n is large (as is the case in image processing where n can be of the order 256^2 or 512^2) significant savings on computational time can be achieved.

Proposed Method

To simplify our presentation of the method we need the following notation. A sequence $f = (f_0, f_1, f_2, .., f_{n-1})$ is denoted by f_1 and f_- when it is expressed as a column and row respectively:

$$f_1 = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ f_{n-1} \end{bmatrix} \qquad f_- = [f_0 \; f_1 \; .. \; f_{n-1}]$$

If a column or row is expressed as a sequence we indicate this as $f_\sim$. f_M stands for the diagonal matrix:

$$f_M = \begin{bmatrix} f_0 & & & \\ & f_1 & & \\ & & \ddots & \\ & & & f_{n-1} \end{bmatrix}$$

A* and $A^\dagger$ represent the complex conjugate and the Hermitian conjugate (transpose-complex conjugate) of matrix A. Then (2a) and (2b) can be expressed as:

Minimize $(f_\sim * c_\sim)_-(f_\sim * c_\sim)_1$ ——(3a)

Subject to $(g_\sim - k_\sim * f_\sim)_-(g_\sim - k_\sim * f_\sim)_1$

$= \varepsilon_-\varepsilon_1 = e$ ——(3b)

where $f_\sim = (f_0, f_1, .., f_{n-1})$, $c_\sim = (1, -2, 1, 0, 0, .., 0)$, $g_\sim = (g_0, g_1, .., g_{n-1})$, $k_\sim = (k_0, k_1, .., k_{n-1})$, and $\varepsilon_\sim = (\varepsilon_0, \varepsilon_1, .., \varepsilon_{n-1})$

Define finite Fourier transforms as in the following:

$$X(p) = \sum_{j=0}^{n-1} x(p)W^{-pj}$$

$$x(p) = \frac{1}{N} \sum_{j=0}^{n-1} X(p)W^{pj}$$

where $W = \exp(2\pi/n)$. Let $F_{\sim}$, $C_{\sim}$, $K_{\sim}$, $G_{\sim}$ and $E_{\sim}$ be the respective Fourier transforms of $f_{\sim}$, $c_{\sim}$, $k_{\sim}$, $g_{\sim}$ and $\varepsilon_{\sim}$. Then it follows that the conditions:

$$\text{Minimize } (F_{\sim}C_{\sim})^{*}_{-}(F_{\sim}C_{\sim})_1 \qquad \text{—— (4a)}$$

$$\text{subject to } (G_{\sim}-K_{\sim}F_{\sim})^{*}_{-}(G_{\sim}-K_{\sim}F_{\sim})_1 = E^{*}_{-}E_1 \qquad \text{—— (4b)}$$

are equivalent to (3a) and (3b) because of convolution and Parseval's theorems [4]. Hence if f satisfies (3a) and (3b) then its Fourier transform F satisfies (4a) and (4b) and vice-versa.

Recasting (4a) and (4b) in matrix form we have:

$$\text{Minimize } F^{*}_{-}C_M^{\dagger}C_MF_1 \qquad \text{—— (5a)}$$

$$\text{subject to } (G_1-K_MF_1)^{\dagger}(G_1-K_MF_1) = E^{*}_{-}E_1 = ne \qquad \text{—— (5b)}$$

where

$$K_M = \begin{bmatrix} K_o & & & \\ & K_1 & & \\ & & \ddots & \\ & & & K_{n-1} \end{bmatrix}$$

and

$$C_M = \begin{bmatrix} C_o & & & \\ & C_1 & & \\ & & \ddots & \\ & & & C_{n-1} \end{bmatrix}$$

Appendix IV presents a proof that F which satisfies (5a) and (5b) can be given as

$$F_1 = \frac{K_M^{\dagger}}{K_M^{\dagger}K_M + \gamma C_M^{\dagger}C_M} G_1 \qquad \text{—— (6)}$$

where $\hat{\gamma}$ is such that (5b) is satisfied. Since K_M and C_M are diagonal we get

$$F_1 = \begin{bmatrix} K_o^*/|K_o|^2 + \gamma|C_o|^2 & & \bigcirc \\ & K_1^*/|K_1|^2 + \gamma|C_1|^2 & \\ \bigcirc & & K_{n-1}^*/|K_{n-1}|^2 + \gamma|C_{n-1}|^2 \end{bmatrix} \begin{bmatrix} G_o \\ G_1 \\ \cdot \\ \cdot \\ \cdot \\ G_{n-1} \end{bmatrix}$$

$$\text{Hence } G_1 - K_M F_1 = \begin{bmatrix} G_o - |K_o|^2 G_o/|K_o|^2 + \gamma|C_o|^2 \\ G_1 - |K_1|^2 G_1/|K_1|^2 + \gamma|C_1|^2 \\ \cdot \\ \cdot \\ \cdot \\ G_{n-1} - |K_{n-1}|^2 G_{n-1}/|K_{n-1}|^2 + \gamma|C_{n-1}|^2 \end{bmatrix}$$

$$\text{and from (4b) } \sum_{i=o}^{n-1} |G_i|^2 (1 - |K_i|^2/|K_i|^2 + \gamma|C_i|^2)^2$$

$$= \sum_{i=o}^{n-1} |E_i|^2 = ne \qquad \text{—— (7)}$$

Letting $\alpha_i = |G_i|^2$, $\beta_i = |K_i|^2/|C_i|^2$, $\hat{\lambda} = 1/\hat{\gamma}$ and

δ = ne, equation (7) becomes:

$$\sum_{i} \frac{\alpha_i}{(1 + \beta_i \lambda)^2} = \delta \qquad \text{—— (8)}$$

where i is such that $C_i \neq 0$.

Once $\hat{\lambda}$ is determined F can be found directly by using (6).

By using the proposed method it is possible to increase the speed of computation by a factor of 2 to 4. In Hunt's method there are 4 fast Fourier transforms/inversions and one fast Fourier transform per iteration. Thus his method has a computational complexity of O(4F + pF) multiplications where F ($=O(n\log_2 n)$) is the number of multiplications required to perform one Fourier transform and p is the number of iterations needed for the final solution. The present method has a complexity of O(4F); here it is assumed that each iteration to solve equation (8) takes O(n) operations. Thus we can speed up the computation by 1 + p/4 times. Since Hunt reports that p is observed to be between 3 and 12, the speed-up factor may be anywhere between 1.75 and 4.

References for Appendix III

[1] Phillips, D.L., "A Technique for the Numerical Solution of Certain Integral Equations of the First Kind", J. Assoc. Comp. Mach., Vol. 9, pp. 84-97, January, 1962.

[2] Twomey, S., "On the Numerical Solution of Fredholm Integral Equation of the First Kind by the Inversion of the Linear System Produced by Quadrature", J. Assoc. Comp. Mach., Vol. 10, pp. 97-101, January 1963.

[3] Hunt, B.R., "The Application of Constrained Least Squares Estimation to Image Restoration by Digital Computer", IEEE Trans. on Computers, Vol. C-22, pp. 805-812, September 1973.

[4] Cooley, J.W., Lewis, P.A., and Welch, P.D., "The Finite Fourier Transform", IEEE Trans. Audio Electroaconst., Vol. AU-17, pp. 77-85, June 1969.

(The reverse of this page is blank).

APPENDIX IV

DERIVATION OF THE OPTIMAL FILTER IN FREQUENCY DOMAIN

In this appendix we show that the optimal solution f which minimizes $f^{\dagger}Cf$ subject to

$$(g-Hf)^{\dagger}(g-Hf) - k = 0$$

is given by:

$$f = \frac{H^{\dagger}g}{H^{\dagger}H + \gamma C}$$

where γ is a constant. Here f and g are nx1 complex column matrices; and $H = \text{diag}\,(H_0, H_1, .., H_{n-1})$ and $C = \text{diag}\,(C_0, C_1, .., C_{n-1})$ are diagonal complex matrices. Let $f_i = f_{ir} + jf_{ic}$, $g_i = g_{ir} + jg_{ic}$ and $H_i = H_{ir} + jH_{ic}$. Form the Lagrangian

$$\Phi = f^{\dagger}Cf + \lambda\{(g-Hf)^{\dagger}(g-Hf)-k\}$$

and set $\partial\phi/\partial f_{ir} = \partial\phi/\partial f_{ic} = 0$ to find the optimal solution.

Since $f^{\dagger}Cf = \sum_{i=0}^{n-1} C_i(f_{ir}^2 + f_{ic}^2)$,

$$\partial(f^{\dagger}Cf)/\partial f_{ir} = 2C_i f_{ir} \qquad \text{——(A1)}$$

$$\partial(f^{\dagger}Cf)/\partial f_{ic} = 2C_i f_{ic} \qquad \text{——(A2)}$$

Let $W = (g-Hf)^{\dagger}(g-Hf) = \sum_{i=0}^{n-1} (g_i-H_i f_i)^{*}(g_i-H_i f_i)$.

Then $\partial W/\partial f_{ir} = (g_i - H_i f_i)^*(-H_i)$

$$+ (-H_i)^*(g_i - H_i f_i)$$

$$= -2(g_{ir}H_{ir} + g_{ic}H_{ic}) + 2|H_i|^2 f_{ir} \qquad \text{——(A3)}$$

Similarly $\partial W/\partial f_{ic} = -2(g_{ic}H_{ir} - g_{ir}H_{ic}) + 2|H_i|^2 f_{ic}$ ——(A4)

Combining (A1) and (A3) and noting $\partial\phi/\partial f_{ir} = 0$,

$$C_i f_{ir} - \lambda\{(g_{ir}H_{ir} + g_{ic}H_{ic}) - |H_i|^2 f_{ir}\} = 0$$

Similarly $C_i f_{ic} - \lambda\{(g_{ic}H_{ir} - g_{ir}H_{ic}) - |H_i|^2 f_{ic}\} = 0$

Hence $C_i f_i - \lambda\{H_i^* g_i - |H_i|^2 f_i\} = 0$ for an optimal solution.

Therefore $f = \dfrac{H^\dagger g}{H^\dagger H + \gamma C}$ where $\gamma = 1/\lambda$.

☆U.S. GOVERNMENT PRINTING OFFICE: 1976-614-357/392