CR-152108

(NASA-CR-152108) PRELIMINARY STUDY FOR A NUMERICAL AERODYNAMIC SIMULATION FACILITY. PHASE 1: EXTENSION (Control Data Corp., St. Paul, Minn.) 434 p HC A19/MF A01 CSCL 01A

N78-19052

Unclas
G3/02 08630

PRELIMINARY STUDY

FOR A

NUMERICAL AERODYNAMIC SIMULATION FACILITY

SUMMARY REPORT — PHASE 1 EXTENSION

By: N. R. Lincoln

FEBRUARY, 1978

Prepared under Contract No. NAS2-9457 by:

CONTROL DATA CORPORATION
Research and Advanced Design Laboratory
4290 Fernwood Street
St. Paul, Minnesota 55112

for

AMES RESEARCH CENTER

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

SUMMARY REPORT – PHASE 1 EXTENSION

Phase I of the NASF study which was completed in October 1977 produced several conclusions about the feasibility of construction of a flow model simulation facility. A computer structure was proposed for the Navier-Stokes Solver (NSS), now called the Flow Model Processor (FMP), along with technological and system approaches. Before such a system can enter an intensive design investigation phase several tasks must be accomplished to establish uniformity and control over the remaining design steps, as well as clarifying and amplifying certain portions of the conclusions drawn in Phase 1.

In order of priority these were seen as:

1. Establishing a structure and format for documenting the design and implementation of the FMP facility.
2. Developing a complete, practically engineered design that would perform as claimed in the Phase 1 report.
3. Creating a design verification tool for NASA analysts, using a computerized simulation system.
4. Identifying key elements of the flow model three-dimensional codes to be used as metrics for verifying the progress of FMP design against a set of predefined system objectives.
5. Developing a programming language specification for a proposed FMP language to be tested by Ames and RADL personnel.
6. Coding of the key elements in the experimental language.
7. Hand compilation of the encoded program segment.
8. Submission of the hand-compiled elements to the FMP simulator for timing and data flow analysis.
9. Documentation and "packaging" of the resulting simulator and input code to permit NASA personnel to continue experiments and analysis.
10. Development of sufficiently detailed functional descriptions as to permit NASA personnel to develop their own simulations where necessary.
11. Refinement of Reliability Analysis to include realistic estimates of component counts.

These tasks were attacked with all of the Phase 1 personnel plus three additional designers and mathematicians. The major expenditure of time was in the revision and re-revision of the computer design and the production of a CPU Instruction Specification and Functional Specification which are included as appendices to the Final Report, and which were fundamental to all but the first task listed. An outline of the form and desired content of the basic specifications for software and hardware were produced as guidance and structure skeletons for future contract phases. The 3-D code analysis and language specification were not completed in this phase, as the language direction was changed several times in this admittedly preliminary study phase of the project.

PROJECT STATUS AND CONCLUSIONS

Most of the tasks undertaken in this extension phase were continuations of work initiated in Phase 1 of the NASF study. All of the tasks are expected to continue in some form as the full NASF architecture is defined in detail, and finally designed and implemented in subsequent phases of this project. The "best effort" engaged by Control Data Corporation has resulted in the following task status: *

1. Hardware description

   a. Performance metrics – The three-dimensional implicit code was analyzed to determine if extrapolations from the two-dimensional code done in Phase 1 were still valid. It was found that the Ames guidance regarding the extrapolations were sound, and conclusions about the computational load invoked by the 3-D code still hold from the Phase 1 report.

      No time was spent on the explicit code beyond determining the FMP instruction requirements for data dependent processing that differs in part from the implicit code behavior. The result was a decision to retain the APL vector operations of vector search and compare and the corresponding operations of compress, mask and merge. Bit string operations on long strings were eliminated as an unnecessary complication to the hardware and supplanted by a scalar (non vector) form of handling bit string operations. The performance degradation due to this change was estimated at less than 1 percent for the entire code execution.

      Pertinent segments of the implicit code were identified for further study. The sequence of subroutines or subprocesses used in forming the tridiagonal matrices from sweeps in the three mesh directions, and the tridiagonal solver constitute the obvious key computational burden of the implicit code, while the memory accessing patterns for both Main Memory and Backing Store can be derived from the examination of the AMATRX, FILTRX, FILTRY, FILTRZ sequences. The conclusion is that for a "first-cut" validation of the FMP hardware architecture, a minimum of these subprocesses and the BTRI/LUDEC (Block tridiagonal solver and LU decomposer for the solver) should be programmed, hand-compiled and simulated on the block-level simulator. In addition, the mathematical behavior of these sequences should be analyzed to determine if, in all practical cases, the arithmetic can be done in 32-bit mode to improve the throughput characteristics of this code in the 64/32-bit pipelines of the proposed FMP.

      A hand compilation of a segment of the beginning of the J sweep for the left-hand-side solution was accomplished. The limited code generated was constrained by the time remaining after decisions were finalized on the FORTRAN extensions to be proposed. This region of the code is critical to the performance of the FMP since it exercises the memory system in the most inefficient manner possible in the implicit code. Data cannot be streamed directly into vector arithmetic operations, but first must be GATHERED from discontiguous columns of data in the original mesh. With the minimal resources remaining, it was decided that this sequence was the most interesting to hand-compile and block-model with the GPSS simulator.

      The sequence of instructions was submitted to the Version 1 FMP model, with the result that a floating-point rate of 933 megaflops was achieved under these worst-case conditions. The implication (which will be explored more thoroughly in the next study phase) is that, in fact, this small segment is truly the worst worst-case and thus the average computation rate will be well in excess of the one-gigaflop threshhold sought by NASA.

At the time of this writing, the hand compiling and block simulation have not been completed, but should be available at the time of submission of the final report.

b. Functional design – Design of the originally proposed Control Data FMP was revised and a preliminary set of functional and instruction specifications was produced for the FMP. The major design changes involved two main thrusts:

- Reduction of complexity of the Map Unit and Vector Unit by constraining the generality originally proposed for those units.
- Improvements in reliability by increasing the amount of Single Error Correction Double Error Detection encoding throughout the Vector Units and associated buffers, the addition of more checking logic, and an additional pipeline for "instantaneous" swapping into the vector arithmetic ensemble as failures are detected.

The technological risks of using a new circuit family in the FMP which were strongly emphasized in the Phase 1 report were re-examined and the decision was made to proceed on two concurrent paths in the development of the FMP. These were to pursue the new technological developments aggressively while at the same time, assessing the configurations and reliabilities of the FMP as if it were to be built out of existing ECL LSI as used in the STAR-100 program. It was decided to take an approach to the newer generation LSI that would not invalidate all of the architectural or block-level design, so that those tasks could proceed somewhat independently of the technology development. This decision thus permits Control Data to postpone the recommendation for logic family until much later in the design cycle, and thus delay consideration until more firm commitments can be made regarding the risks and schedules for a new logic family. It has been determined as a bottom line that an FMP can be built, and operated with acceptable reliability, from the existing LSI family being employed on the STAR-100A.

The block-level simulation system required by Ames personnel to permit an ongoing verification of the design and the overall performance objectives is under development. A last minute decision was made to base the first, highest-level design model on the readily available General Purpose Simulation System (GPSS) so that it could be easily installed at any site performing FMP analysis.

The first version of the simulator has been completed and is being delivered under separate cover to NASA Ames with the submission of this Final Report. The simulator provides a substantial amount of statistical data about the behavior of the various FMP units (Swap, Map, Memory, Scalar, Vector) when executing code sequences. This data permits the designers to evaluate alternate strategies for organization and implementation of the major components of the FMP. NASA analysts can use the data to verify that the internal characteristics of the FMP as documented in the Functional and Instruction Specifications are truly represented by the current version of the GPSS model.

Test runs of the model immediately prior to completing this report have disclosed that there is need for refinement in some areas, thus the model is not complete to the extent needed by the design engineers. It is adequate in current form, however, for NASA analysts to evaluate the behavior of these initial FMP designs.

c. Reliability assessment – A reliability analysis was conducted of the revised FMP design, using experimental statistics from existing logic family data, expectations of the LSI and memory characteristics of componentry borrowed from the STAR-100A project, and projections made by our technological team for as yet untried technological developments. The key conclusion was that the total FMP including Backing Storage would suffer an expected rate of failure of 4 per operating month, if a reasonable maintenance schedule was followed to clean out single errors that are automatically corrected by the SECDED networks. This failure rate is expected from the existing technology family of LSI. The failure rate is almost halved if a newer technology can be utilized for the FMP.

Since the major rates of failure (system interruption) revolve around the Vector Units, the additional (ninth) unit was designed into the FMP to permit quick restart capability, thus improving the overall machine availability.

2. Software description

   a. Programming language – A software description was developed, providing a "straw-man" FORTRAN language to be tested by Ames and Control Data programmers, and a description of the fundamental operating system properties required by the NASF, constructed in a format suitable for evolving into an operating system specification.

      The FORTRAN dialect was created after much thought about automatic vectorization of standard FORTRAN constructs led Control Data to the conclusion that the compiler writers and compilers need a little assistance in the generation of optimum object code for parallel machines. Ames staff comments on the original extensions to FORTRAN based on the STAR FORTRAN compiler, led RADL researchers to abandon the multitude of extensions therein and concentrate on the definition of a CODO (Concurrent DO loop) structure, within which all operations are explicitly vector.

      The minimization of language enhancements reduces the risks attendant to retraining programming personnel, as well as major compiler developments, to implement new syntax. This simplification is accomplished at the cost of more intensive effort on compiler optimization of source code. The optimization techniques are not unique to the FMP in that they have their direct counterparts in the scalar optimization performed in most product compilers available in 1978. The risk of providing a mature compiler with this capability is reasonably low compared to the alternatives of creating a wholly new language and compiler system from scratch.

   b. Operating system – The operating system description attempts to reduce to a bare minimum those functions which must uniquely be implemented in the FMP. Further, every attempt must be made to utilize existing system software on the non-FMP processors, which will be procured essentially off-the-shelf, in order to minimize the software risks highlighted in the Phase 1 report.

      The operating system document is intended to evolve into formal specification form in later phases of this project, as are all portions of the NASF project. A system should be created to handle the updating and editing of these documents as a central means for coordinating the total project. To this end, outlines for the specification of language structure and compiler structure should have been completed in this study phase. The prototypes originally intended to be used for this function (the ANS 77 language specification, and the STAR FORTRAN internal maintenance specification) proved insufficient (and in some cases unnecessarily complicated) to fill this role. The outline of these specifications should be done at the earliest possible moment in the next project phase.

FINAL WORD

The achievement of the original NASF project goals appears at this point to be more possible than when the feasibility was first examined at the beginning of Phase 1. Continued study of the code requirements and refinement of the design have led to simplifications in the FMP that reduce the software and hardware risks below those originally derived for the NASF. A major factor in reducing risks and meeting the performance and schedule goals for the FMP rest in the willingness of the algorithm developers to bend their thinking somewhat toward a more intimate knowledge of the hardware for which they are creating programs. This flexibility in adapting thought and code to a special machine architecture has been a major, positive characteristic of the Ames flow model mathematicians and programmers, with their experience on ILLIAC IV and the 7600 on which to draw. Thus the success of the FMP seems assured if the intimate marriage of code developers, hardware designers, and technologists can continue for the life of the project.

PRELIMINARY STUDY

FOR A

NUMERICAL AERODYNAMIC SIMULATION FACILITY

FINAL REPORT – PHASE 1 EXTENSION

By: N. R. Lincoln

Contributions By: D. R. Resnick
F. M. Green

FEBRUARY, 1978

Prepared under Contract No. NAS2-9457 by:

CONTROL DATA CORPORATION
Research and Advanced Design Laboratory
4290 Fernwood Street
St. Paul, Minnesota 55112

for

AMES RESEARCH CENTER

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

PREFACE

This Final Report presents additional findings resulting from an extension of the Preliminary Study for a Numerical Aerodynamic Simulation Facility (NASF) as determined by Control Data Corporation. The document consists of five sections. Other than Section 1, Introduction, each section addresses a specific aspect of the Control Data study.

Section 1 is a background of the Phase 1 study and what has been achieved since the issuance of the study. Section 2, Hardware Description, presents a more detailed definition of the FMP than that provided in the Phase 1 report. Section 3, Reliability Assessment, gives information on the methodology and analysis used to estimate failure of hardware components plus the estimated failure rate data. Section 4, Software Description, expands previous discussions of proposed FMP software. Section 5, Appendixes, presents both the Instruction and Functional Computer FMP Specifications plus appendixes on a Programmed Device Controller and Serial Trunk Controller Procedures.

In addition to this Final Report, a separate Summary Report presents the salient findings of this preliminary study and summarizes the extension of the first phase of a program for the development of a Numerical Aerodynamic Simulation Facility.

TABLE OF CONTENTS

Section 1

INTRODUCTION

Section 1

INTRODUCTION

Since early in the year 1977, the Ames Research Center of the National Aeronautics and Space Administration (NASA) and the Research and Advanced Design Laboratory (RADL) of Control Data Corporation have been conducting a cooperative research program in the investigation of the feasibility and applicability of extremely high performance computers to the process of airframe design. The first phase of this effort culminated in a Final Report produced by Control Data for NASA in October 1977. Like the first phase investigations themselves, that report attempted to answer several questions posed by Ames researchers such as:

> "How much compute power is necessary to achieve the design and engineering goals of NASA aerodynamacists?" Answer: "Some computing power in excess of one-billion floating-point computations per second."
>
> "Can such a computer be built for operation in the early 1980s, with acceptable reliability and availability?" Answer: "Yes."
>
> "What are the architectural alternatives for such a computer?" Answer: "Either a pipeline or array processor of the SIMD variety is best suited to the task."
>
> "What technologies are available for implementing such a high powered machine?" Answer: "In practical supply or operating parameters, very few technologies. The computer should be built with the mature and still extendable silicon emitter-coupled logic for speed."
>
> "What system architecture is suitable for the entire computational facility?" Answer: "A distributed processing system similar to that implemented for the STAR-100 family."
>
> "What are the programming considerations for such a machine?" Answer: "FORTRAN should be the computational programming language with extensions to permit specific access to hardware parallelism, when necessary."

The Phase 1 report consciously included much tutorial material to assist Ames personnel in perceiving the state of the computer design and construction art at the same level as Control Data deals with new computer developments. Included in the report were preliminary designs and descriptions of a possible candidate machine architecture, installation design, and programming approaches for the review and commentary of the Ames project staff. Discussions between Ames and Control Data researchers during the preparation of draft material for the Final Report led to substantial revisions in the technical design of the language and machine structure. Subsequent to the publication of that report, similar discussions have led to even more changes in the implementation of the algorithm, language, and computer hardware.

Since this initial effort was intended to be a part of an ongoing development project leading to the actual design of all system components and to the construction of a facility to perform the aerodynamic simulations, an extension of the study was indicated, preliminary to launching more refined design and simulation of the system.

This study extension was intended to bridge the gap between the completion of the feasibility phase and the second, detailed specification phase of the research effort.

PHASE 1 EXTENSION DESCRIPTION

An examination of the state of the project after the Final Report for Phase 1 was conducted and the apparent needs for the Phase 2 stage were outlined. Several desirable tasks became evident from that analysis, from NASA commentary regarding the true feasibility of the proposed approach, and from NASA questions regarding the broader applicability of the system being considered. In addition, a three-dimensional form of the Ames "Implicit code" (whose 2-D form was used in the Phase 1 Final Report to evaluate the design) became available. Thus a form of the solution methodology closer to what is expected to be used in the final facility was at hand for analysis with the proposed system architecture.

These factors led to the establishment of a formal "Phase 1 Extension" whose purpose was to continue the Phase 1 work in several areas and to provide the springboard for the Phase 2 work on detailed design, analysis, and specification of system components. To this end several tasks were identified:

1. Analysis of the three-dimensional model operating on the proposed computer architecture.
2. Development of a proposed programming language form.
3. A more detailed description of the computational engine structure so that timing and storage estimates could be made.
4. Development of tools to provide verification of:
    a. the ability of a given design to perform the calculations in the time required;
    b. the reality of creating hardware that will perform as described in (a.) above.
5. Description of the overall operating system function to support the computational workload.
6. Refinement of reliability estimates for the computational hardware.

With the resources available, and the brief three-month period for completion it was decided that all of the tasks could not be completed to the point of detailed specification, or final structural design of either hardware or software. Instead the Phase 1 Extension was conceived as a means for providing the structure within which all subsequent phases would be carried out, and as a period when a number of proposed language and hardware schemes could be reconciled with Ames staff members. Past experience indicates that the most appropriate means for structuring and controlling a development project of this magnitude has been most effectively achieved through the identification and implementation of "standard" specifications for each system component.

The basis for the report which follows is thus a skeleton structure of specifications for the computational hardware, the operating system, the language and its processor, and its reliability. In some cases sufficient detailed work was completed to permit the beginning of a complete specification. For example, since much of the STAR-100A Scalar Unit and maintenance philosophy were to be employed in Control Data's proposed architecture, the STAR-100A specification was used as a basis for the FMP hardware specification. This specification will be modified and updated as design continues through the Phase 2 and final construction portions of this project.

An outline for an operating system specification was generated and used as the basis for a preliminary system description of the overall operating system. In subsequent phases of the project, each of the paragraphs in the outline will be replaced by specification information rather than the functional requirement or preliminary design data as provided in this report.

Although it was intended to provide a full specification for the language and compiler in this phase, the effort was beyond the means of RADL to accomplish in the three months allowed. This is partly due to the changes in direction for the language that have resulted from interaction with Ames staff and with Control Data's language specialists. An initial attempt to base a specification on the ANS FORTRAN 77 specification was abandoned as approval of that standard has been delayed, and as it became apparent that the specification as written could not be easily used as a guide for experimental programming. The use of the STAR-100 FORTRAN compiler documentation proved to be undesirable as Ames personnel insisted on abandoning some of the artificial constructs for vector processing that were felt to be difficult to use, or to understand. The result then, in this first pass, is a presentation of a "strawman" proposal for a FORTRAN language based on ANS FORTRAN 77, with several extensions designed for the flow model computations to be done at Ames.

The process of providing tools for architectural and hardware verification has proceeded down divergent paths also. At the time of submission of this report, a tape is being provided to Ames which will allow them to perform computerized evaluations of the behavior of the overall computational segment of the installation with varying forms of machine language coding.

The report that follows then presents further proposals for the hardware architecture and is, in essence, a proposal for the form and content of specifications to be generated in full in the next phase of this project. It is expected that these proposals will lead to discussions between Control Data and Ames within the coming months to arrive at a final architectural project structure and project management and control format based on documentation methodology and management.

Unlike the Phase 1 report, this report contains no major answers to major questions. However, the feasibility of construction of the desired facility appears greater now than it did in Phase 1. Further design attempts to reduce the size and complexity of the major processor have improved chances of its being built with existing technology, thus reducing one of the significant risks highlighted in the Phase 1 study. The direction for the continuation of design and implementation of the total facility is becoming clearer and more practical as the cooperative study continues.

RELATIONSHIP TO OVERALL PROJECT

In the initial study phase, the computational engine was referred to as the NSS (Navier-Stokes Solver). The existence of such a massively powerful system, with the attendant major investments in facilities, money, and personnel mandate an examination of the broader applicability of such a system. Thus, to symbolize the more catholic potential of this facility, the computational engine has been renamed the Flow Model Processor (FMP). The other computers in the system (regardless of computational capability) are called Front-End Processors (FEPs) to eliminate the need for identifying a particular brand or architecture, and the intelligent (programmable) communications equipment and attachment devices are called Programmable Device Controllers (PDCs). Regardless of the shape and content of the final system, these major components will exist in some form and thus the use of this terminology pervades this report.

As stated in the previous section, the outlines of documentation in specification form are to establish the structure for documentation for following stages of the project. If a fixed form of specification can be agreed upon for each component (language, compiler, FMP monitor, and FMP hardware) then a rigorous control system can be established in Phase 2 wherein all design changes are referenced to their particular area of specification and an automated audit trail provided for each specification update, thus tracing the evolution of the system, and fixing responsiblity (by designer) for any given change in the specifications. It is at this stage of the project wherein such management devices must be created lest the project be faced with virtual chaos in Phase 2 and later phases of the effort.

Thus the interaction with NASA project managers is essential concerning the form (not the content) in which all technical matters are submitted for review and decision from this point forward. As stated in the previous section, the outlines or specification skeletons offered here are not expected to be all inclusive but to form the basis for detailed discussions at a later time.

Section 2

HARDWARE DESCRIPTION

Section 2

HARDWARE DESCRIPTION

PERFORMANCE METRICS

For this study the three-dimensional forms of the implicit and explicit codes were submitted to Control Data for additional evaluation of the FMP design that has evolved since Phase 1. The Phase 1 effort of code analysis was primarily directed to the implicit form because its computational behavior was easier to estimate since the number of times arithmetic is performed is fixed rather than somewhat data dependent, as happens in the explicit form of the Navier-Stokes solution. The emphasis in this study was extended to the three-dimensional code primarily because of the data already derived for the two-dimensional model described in the Phase 1 study.

ANALYSIS OF VECTORIZED 3-D MODELS

The analysis of the current FORTRAN codes undertaken in this phase was intended to accomplish the following items:

A. Updating of statistical data on computational behavior of the code for the three-dimensional versions to contrast with the data taken from the STAR-100 in Phase 1 of this study.

B. Identification of the key areas of the codes to isolate benchmark candidates for measuring performance.

C. Experimentation with various forms of source language coding to produce vectorization.

D. Develop segments of code that could be 'hand-compiled' for the FMP to illustrate the machine-language-level execution of portions of the computation.

E. Analysis of memory access patterns induced by the code in three dimensions.

An inordinate amount of project resources were quickly absorbed in the redesign of some portions of the FMP and in pursuing several alternatives for the programming language. Thus the objectives of this section of the study became redirected to match the time available, and to answer some more pressing questions. The results of each original objective are as follows:

A. No statistical counting was done for either the three-dimensional implicit or explicit codes on the STAR-100. Computational counts for the implicit code which were projected for three dimensions in the Phase 1 report, appear to be roughly matched to the 3-D implicit version now in hand.

B. It is obvious to the casual analyst that the region of code in the 3-D implicit program containing the AMTRX, FILTRX, FILTRY, FILTRZ and three subroutine calls to the metric computations XXM, YYM, ZZM constitute the critical area for analysis of the model. From Table 5-40 of the Phase 1 report, the 'left-hand-side' calculations including those code sequences account for 19,779,456 operations out of a total of 24,666,130 operations, or about 80 percent of the total operations. In addition, the 'sweeps' made in this code in the three directions of the matrix create all of the expected access patterns that need to be analyzed for the FMP. No effort was expended on a similar analysis of the explicit code due to lack of resources and time. The original scalar coding in FORTRAN of these segments is given again in Figure 2-1.

```
      DO 12 J = 1,JMAX
      R1 = XX(J,1)*HDX
      R2 = XX(J,2)*HDX
      R3 = XX(J,3)*HDX
      R4 = XX(J,4)*HDX
C
C*******AMATRX
C
      RR = 1./Q(KL,1,J)
      U = Q(KL,2,J)*RR
      V = Q(KL,3,J)*RR
      W = Q(KL,4,J)*RR
      UU = U*R1+V*R2+W*R3
      UT = U**2+V**2+W**2
      C1 = GAMI*UT*.5
      C2 = Q(KL,5,J)*RR*GAMMA
      C3 = C2 - C1
      C4 = R4+UU
      C5 = GAMI*U
      C6 = GAMI*V
      C7 = GAMI*W
      D(J,1,1) = R4
      D(J,1,2) = R1
      D(J,1,3) = R2
      D(J,1,4) = R3
      D(J,1,5) = 0.
      D(J,2,1) = R1*C1 - U*UU
      D(J,2,2) = C4+R1*GAM2*U
      D(J,2,3) = -R1*C6+R2*U
      D(J,2,4) = -R1*C7+R3*U
      D(J,2,5) = R1*GAMI
      D(J,3,1) = R2*C1-V*UU
      D(J,3,2) = R1*V-R2*C5
      D(J,3,3) = C4+R2*GAM2*V
      D(J,3,4) = -R2*C7+R3*V
      D(J,3,5) = R2*GAMI
      D(J,4,1) = R3*C1-W*UU
      D(J,4,2) = R1*W-R3*C5
      D(J,4,3) = R2*W-R3*C6
      D(J,4,4) = C4+R3*GAM2*W
      D(J,4,5) = R3*GAMI
      D(J,5,1) = (-C2+2.*C1)*UU
      D(J,5,2) = R1*C3-C5*UU
      D(J,5,3) = R2*C3-C6*UU
      D(J,5,4) = R3*C3-C7*UU
      D(J,5,5) = R4+GAMMA*UU
C
C******END OF AMATRX
C
  12  CONTINUE
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code

```
      DO 25 J=JA,JB
      RJ = 1./Q(KL,6,J)
      RMJ = RM*RJ
      RR = RMJ*Q(KL,6,J-1)
      RF = RMJ*Q(KL,6,J+1)
      DO 23 N=1,5
      A(J,N,1) = -D(J-1,N,1)
      A(J,N,2) = -D(J-1,N,2)
      A(J,N,3) = -D(J-1,N,3)
      A(J,N,4) = -D(J-1,N,4)
      A(J,N,5) = -D(J-1,N,5)
      B(J,N,1) = 0.0
      B(J,N,2) = 0.0
      B(J,N,3) = 0.0
      B(J,N,4) = 0.0
      B(J,N,5) = 0.0
      C(J,N,1) = D(J+1,N,1)
      C(J,N,2) = D(J+1,N,2)
      C(J,N,3) = D(J+1,N,3)
      C(J,N,4) = D(J+1,N,4)
      C(J,N,5) = D(J+1,N,5)
      A(J,N,N) = A(J,N,N)-RR
      B(J,N,N) = C8
      C(J,N,N) = C(J,N,N)-RF
23    F(J,N)=S(KL,N,J)
25    CONTINUE
C
C*****END OF FILTRX
C
C
C  S MUST BE ZERO ON B.C.
      CALL BTRI(2,JM)
      DO 21 J = 2,JM
      S(KL,1,J) = F(J,1)
      S(KL,2,J) = F(J,2)
      S(KL,3,J) = F(J,3)
      S(KL,4,J) = F(J,4)
21    S(KL,5,J) = F(J,5)
20    CONTINUE
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

```
      COMMON/VAR0/S(720.5,30)
      COMMON/VAR1/X(720,30),Y(720,30),Z(720,30)
      COMMON/VAR3/P(120,30),XX(60,4),YY(60,4),ZZ(60,4)
      LEVEL 2,Q,S,X,Y,Z
      COMMON/COUNT/NC,NC1
      COMMON/FLSH/DX2,DY2,DZ2
C
C  XI METRICS FORMED FOR A K,L LINE IN J
C
C
C  SYMMETRY
C
      K = M
      L = LA
      J1=J1A
      J2=J2A
      KL = (L-1)*ND+K
      DO 10 J = J1,J2
      RJ = Q(KL,6,J)
      IF(K.EQ.1) GO TO 50
      IF(K.EQ.KMAX) GO TO 51
      XK = (X(KL+1,J)-X(KL-1,J))*DY2
      YK = (Y(KL+1,J)-Y(KL-1,J))*DY2
      ZK = (Z(KL+1,J)-Z(KL-1,J))*DY2
      GO TO 72
50    CONTINUE
      XK = (-3.*X(KL,J)+4.*X(KL+1,J)-X(KL+2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(KL+1,J)-Y(KL+2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(KL+1,J)-Z(KL+2,J))*DY2
      GO TO 72
51    CONTINUE
      XK = (3.*X(KL,J)-4.*X(KL-1,J)+X(KL-2,J))*DY2
      YK = (3.*Y(KL,J)-4.*Y(KL-1,J)+Y(KL-2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(KL-1,J)+Z(KL-2,J))*DY2
 72   CONTINUE
      IF(L.EQ.1) GO TO 52
      IF(L.EQ.LMAX) GO TO 53
      XL = (X(KL+ND,J)-X(KL-ND,J))*DZ2
      YL = (Y(KL+ND,J)-Y(KL-ND,J))*DZ2
      ZL = (Z(KL+ND,J)-Z(KL-ND,J))*DZ2
      GO TO 60
52    CONTINUE
      XL = (-3.*X(KL,J)+4.*X(KL+ND,J)-X(KL+2*ND,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(KL+ND,J)-Y(KL+2*ND,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(KL+ND,J)-Z(KL+2*ND,J))*DZ2
      GO TO 60
53    CONTINUE
      XL = (3.*X(KL,J)-4.*X(KL-ND,J)+X(KL-2*ND,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(KL-ND,J)+Y(KL-2*ND,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(KL-ND,J)+Z(KL-2*ND,J))*DZ2
 60   CONTINUE
      XX(J,1) = (YK*ZL-ZK*YL)*RJ
      XX(J,2) = (ZK*XL-XK*ZL)*RJ
      XX(J,3) = (XK*YL-YK*XL)*RJ
      XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
 10   CONTINUE
      RETURN
      END
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

```
      SUBROUTINE BTRI(ILA,IUA)
      COMMON/BTRID/A(60,5,5),B(60,5,5),C(60,5,5),D(60,5,5),F(60,5)
      DIMENSION H(5,5)
      REAL L11,L21,L22,L31,L32,L33,L41,L42,L43,L44,L51,L52,L53,L54,L55
      IL=ILA
      IU=IUA
      IS=IL+1
      IE=IU-1
C     INSERT LUDEC
      L11=1./B(IL,1,1)
      L21=B(IL,2,1)
      U12=B(IL,1,2)*L11
      L22=1./(B(IL,2,2)-L21*U12)
      U13=B(IL,1,3)*L11
      U14=B(IL,1,4)*L11
      U15=B(IL,1,5)*L11
      L31=B(IL,3,1)
      L32=B(IL,3,2)-L31*U12
      U23=(B(IL,2,3)-L21*U13)*L22
      L33=1./(B(IL,3,3)-U13*L31-U23*L32)
      U24=(B(IL,2,4)-L21*U14)*L22
      U25=(B(IL,2,5)-L21*U15)*L22
      L41=B(IL,4,1)
      L42=B(IL,4,2)-L41*U12
      L43=B(IL,4,3)-L41*U13-L42*U23
      U34=(B(IL,3,4)-L31*U14-L32*U24)*L33
      L44=1./(B(IL,4,4)-U14*L41-U24*L42-U34*L43)
      U35=(B(IL,3,5)-L31*U15-L32*U25)*L33
      L51=B(IL,5,1)
      L52=B(IL,5,2)-L51*U12
      L53=B(IL,5,3)-L51*U13-L52*U23
      L54-B(IL,5,4)-L51*U14-L52*U24-L53*U34
      U45=(B(IL,4,5)-L41*U15-L42*U25-L43*U35)*L44
      L55=1./(B(IL,5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
C     COMPUTE LITTLE R S
      D1=L11*F(IL,1)
      D2=L22*(F(IL,2)-L21*D1)
      D3=L33*(F(IL,3)-L31*D1-L32*D2)
      D4=L44*(F(IL,4)-L41*D1-L42*D2-L43*D3)
      D5=L55*(F(IL,5)-L51*D1-L52*D2-L53*D3-L54*D4)
C     COMPUTE BIG R S
      F(IL,5)=D5
      F(IL,4)=D4-U45*D5
      F(IL,3)=D3-U34*F(IL,4)-U35*D5
      F(IL,2)=D2-U23*F(IL,3)-J24*F(IL,4)-U25*D5
      F(IL,1)=D1-U12*F(IL.2)-U13*F(IL,3)-U14*F(IL,4)-U15*D5
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

```
C       COMPUTE C PRIME FOR FIRST ROW
        DO 12 M=1,5
        D1=L11*C(IL,1,M)
        D2=L22*(C(IL,2,M)-L21*D1)
        D3=L33*(C(IL,3,M)-L31*D1-L32*D2)
        D4=L44*(C(IL,4,M)-L41*D1-L42*D2-L43*D3)
        D5=L55*(C(IL,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
        B(IL,5,M)=D5
        B(IL,4,M)=D4-U45*D5
        B(IL,3,M)=D3-U34*B(IL,4,M)-U35*D5
        B(IL,2,M)=D2-U23*B(IL,3,M)=U24*B(IL,4,M)-U25*D5
12      (BIL,1,M)=D1-U12*B(IL,2,M)-U13*B(IL,3,M)-U14*B(IL,4,M)-U15*D5
        DO 13 I=IS,IE
C       COMPUTE B PRIME*BIGR
        DO 14 N=1,5
14      F(I,N)=F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)-A(I,N,3)*F(I-1,3)
     1  )-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
C       COMPUTE B PRIME
        DO 11 N=1,5
        DO 11 M=1,5
11      H(N,M)=B(I,N,M)-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)-A(I,N,3)*
     1 B(I-1,3,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,5,M)
C       INSERT LUDEC AGAIN
        L11=1./H(1,1)
        L21=H(2,1)
        U12=H(1,2)*L11
        L22=1./(H(2,2)-L21*U12)
        U13=H(1,3)*L11
        U14=H(1,4)*L11
        U15=H(1,5)*L11
        L31=H(3,1)
        L32=H(3,2)-L31*U12
        U23=(H(2,3)-L21*U13)*L22
        L33=1./(H(3,3)-U13*L31-U23*L32)
        U24=(H(2,4)-L21*U14)*L22
        U25=(H(2,5)-L21*U15)*L22
        L41=H(4,1)
        L42=H(4,2)-L41*U12
        L43=H(4,3)-L41*U13-L42*U23
        U34=(H(3,4)-L31*U14-L32*U24)*L33
        L44=1./(H(4,4)-U14*L41-U24*L42-U34*L43)
        U35=(H(3,5)-L31*U15-L32*U25)*L33
        L51=H(5,1)
        L52=H(5,2)-L51*U12
        L53=H(5,3)-L51*U13-L52*U23
        L54=H(5,4)-L51*U14-L52*U24-L53*U34
        U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
        L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
C       COMPUTE LITTLE R S
        D1=L11*F(I,1)
        D2=L22*(F(I,2)-L21*D1)
        D3-L33*(F(I,3)-L31*D1-L32*D2)
        D4-L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
        D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

```
C       COMPUTE BIG R v S
        F(I,5)=D5
        F(I,4)=D4-U45*D5
        F(I,3)=D3-U34*F(I,4)-U35*D5
        F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
        F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
C       COMPUTE C PRIMES
        DO 15 M=I,5
        D1=L11*C(I,1,M)
        D2=L22*(C(I,2,M)-L21*D1)
        D3=L33*(C(I,3,M)-L31*D1-L32*D2)
        D4-L44*(C(I,4,M)-L41*D1-L42*D2-L43*D3)
        D5-L55*(C(I,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
        B(I,5,M)=D5
        B(I,4,M)=D4-U45*D5
        B(I,3,M)=D3-U34*B(I,4,M)-U35*D5
        B(I,2,M)=D2-U23*B(I,3,M)-U24*B(I,4,M)-U25*D5
15      B(I,1,M)=D1-U12*B(I,2,M)-U13*B(I,3,M)-U14*B(I,4,M)-U15*D5
13      CONTINUE
        I=IU
C       COMPUTE B PRIME*BIG R FOR LAST ROW
        DO 17 N=1,5
17      F(I,N)-F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)-A(I,N,3)*
       *F(I-1,3)-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
C       COMPUTE B PRIME
        DO 18 N=1,5
        DO 18 M-1,5
18      H(N,M)=B(I,N,M0-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)-A(I,N,3)*
       *B(I-1,2,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,5,M)
C       INSERT LUDEC AGAIN
        L11=1./H(1,1)
        L21=H(2,1)
        U12=H(1,2)*L11
        L22=1./(H(2,2)-L21*U12)
        U13=H(1,3)*L11
        U14=H(1,4)*L11
        U15=H(1,5)*L11
        L31=H(3,1)
        L32=H(3,2)-L31*U12
        U23=(H(2,3)-L21*U13)*L22
        L33-1./(H(3,3)-U13*L31-U23*L32)
        U24=(H(2,4)-L21*U14)*L22
        U25=(H(2,5)-L21*U15)*L22
        L41=H(4,1)
        L42=H(4,2)-L41*U12
        L43=H(4,3)-L41*U13-L42*U23
        U34=(H(3,4)-L31*U14-L32*U24)*L33
        L44-1./(H(4,4)-U14*L41-U24*L42-U34*L43)
        U35=(H(3,5)-L31*U15-L32*U25)*L33
        L51=H(5,1)
        L52=H(5,2)-L51*U12
        L53=H(5,3)-L51*U13-L52*U23
        L54=H(5,4)-L51*U14-L52*U24-L53*U34
        U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
        L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

```
C       COMPUTE LITTLE R ~ S
        D1=L11*F(I,1)
        D2=L22*(F(I,2)-L21*D1)
        D3=L33*(F(I,3)-L31*D1-L32*D2)
        D4=L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
        D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
C       COMPUTE BIG R ~ S
        F(I,5)=D5
        F(I,4)=D4-U45*D5
        F(I,3)=D3-U34*F(I,4)-U35*D5
        F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
        F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
        I=IU
20      I=I-1
        DO 19 N=1,5
19      F(I,N)=F(I,N)-F(I+1,1)*B(I,N,1)-F(I+1,2)*B(I,N,2)-F(I+1,3)*B(I,N,3)
     1  )-F(I+1,4)*B(I,N,4)-F(I+1,5)*B(I,N,5)
        IF (I.GT.IL)GOTO20
        RETURN
        END
```

Figure 2-1. Scalar Code Taken From STEP of 3-D Code (Cont.)

C. The segments of code referred to in B. above were subjected to coding in a variety of FORTRAN dialects. The result was the proposal for a set of FORTRAN extensions which are found in Section 4 of this report. Figure 2-2 presents a moderate recoding of the first of three sweeps of the left-hand-side code being analyzed as an example of how such code might appear to the applications programmer. The implicit code, admittedly, is structurally simple, and creates no data dependent vector operations of any magnitude. The vector coding of the implicit code can therefore be done in a very straightforward manner, using the CODO constructs (see Section 4) almost exclusively to provide optimum processing on the FMP.

D. An effort was initiated late in this phase to hand compile the entire key segment of the implicit code. The delay in initiating this effort arose from several fruitless attempts at over enrichment of the FORTRAN syntax to support the FMP. The result has been that only a small segment could be prepared for publication in the time remaining. Figure 2-3 gives the hand compilation example for lines 100 through 170 of the code shown in Figure 2-2.

```
100=          DO 20 L=2,LMAX-1
110=    C***FILTRX
120=    C
130=          CODO J=1,JMAX;K=2,KMAX
140=          RJ=Q(K,L,6,J)
150=          XK=(X(K+1,L,J)-X(K-1,L,J))*DY2
160=          YK=(Y(K+1,L,J)-Y(K-1,L,J))*DY2
170=          ZK=(Z(K+1,L,J)-Z(K-1,L,J))*DY2
180=          XL=(X(K,L+1,J)-X(K,L-1,J))*DZ2
190=          YL=(Y(K,L+1,J)-Y(K,L-1,J))*DZ2
200=          ZL=(Z(K,L+1,J)-Z(K,L-1,J))*DZ2
210=          D(J,1,2)=HDX*((YK*ZL-ZK*YL)*RJ)
220=          D(J,1,1)=HDX*(-OMEGA*(Z(K,L,J)*(RJ*(YK*ZL-ZK*YL))
230=         1  -Y(K,L,J)*RJ*(XK*YL-YK*XL)))
240=          D(J,1,4)=HDX*((XK*YL-YK*XL)*RJ)
250=          D(J,1,3)=HDX*((ZK*XL-XK*ZL)*RJ)
260=    C
270=    C****1    *    *AMATRX
280=    C
290=          RR= 1./Q(K,L,1,J)
300=          U = Q(K,L,2,J)*RR
310=          V = Q(K,L,3,J)*RR
320=          W = Q(K,L,4,J)*RR
330=          UU = U*D(J,1,2)+V*D(J,1,3)+W*D(J,1,4)
340=          UT = U**2+V**2+W**2
350=          C1 = GAMI*UT*.5
360=          C2 = Q(K,L,5,J)*RR*GAMMA
370=          C3=C2-C1
380=          C4=D(J,1,1)+UU
390=          C5=GAMI*U
400=          C6=GAMI*V
410=          C7=GAMI*W
420=          D(J,1,5) = 0.
430=          D(J,2,1) = D(J,1,2)*C1-U*UU
440=          D(J,2,2) = C4+D(J,1,2)*GAM2*U
450=          D(J,2,3) = -D(J,1,2)*C6+D(J,1,3)*U
460=          D(J,2,4) = -D(J,1,2)*C7+D(J,1,4)*U
470=          D(J,2,5) = D(J,1,2)*GAMI
480=          D(J,3,1) = D(J,1,3)*C1-V*UU
490=          D(J,3,2) = D(J,1,2)*V-D(J,1,3)*C5
500=          D(J,3,3) = C4+D(J,1,3)*GAM2*V
510=          D(J,3,4) = -D(J,1,3)*C7+D(J,1,4)*V
520=          D(J,3,5) = D(J,1,3)*GAMI
530=          D(J,4,1) = D(J,1,4)*C1-W*UU
540=          D(J,4,2) = D(J,1,2)*W-D(J,1,4)*C5
550=          D(J,4,3) = D(J,1,3)*W-D(J,1,4)*C6
560=          D(J,4,4) = C4+D(J,1,4)*GAM2*W
570=          D(J,4,5) = D(J,1,4)*GAMI
580=          D(J,5,1) = (-C2+2.*C1)*UU
590=          D(J,5,2) = D(J,1,2)*C3-C5*UU
600=          D(J,5,3) = D(J,1,3)*C3-C6*UU
610=          D(J,5,4) = D(J,1,4)*C3-C7*UU
620=          D(J,5,5) = D(J,1,1)+GAMMA*UU
630=    C
640=    C****1    *    END OF AMATRX
650=    C
660=          ENDCD
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP

```
670=  C
680=        CODO J=2,JMAX-1;N=1,5;K=2,KMAX-1
690=        A(J,N,1) = -D(J-1,N,1)
700=        A(J,N,2) = -D(J-1,N,2)
710=        A(J,N,3) = -D(J-1,N,3)
720=        A(J,N,4) = -D(J-1,N,4)
730=        A(J,N,5) = -D(J-1,N,5)
740=        B(J,N,1) = 0.0
750=        B(J,N,2) = 0.0
760=        B(J,N,3) = 0.0
770=        B(J,N,4) = 0.0
780=        B(J,N,5) = 0.0
790=        C(J,N,1) = D(J+1,N,1)
800=        C(J,N,2) = D(J+1,N,2)
810=        C(J,N,3) = D(J+1,N,3)
820=        C(J,N,4) = D(J+1,N,4)
830=        C(J,N,5) = D(J+1,N,5)
840=        ENDCD
850=  C
860=        CODO J=2,JMAX-1;N=1,5;k=2,KMAX-1
870=        RMJ=RM/RJ
880=        A(J,N,N)=A(J,N,N)-RMJ*(Q(K,L,6,J-1)
890=        B(J,N,N) = CS
900=        C(J,N,N)=C(J,N,N)-RMJ*D(K,L,6,J+1)
910=        F(J,N)=S(K,L,N,J)
920=        ENDCD
930=  C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
 940=  C
 950=  C****1    END OF FILTRX
 960=  C
 970=  C
 980=  C  S MUST BE ZERO ON B.C.
 990=  C      INSERT LUDEC
1000=        CODO K=2,KMAX-1
1010=        L11=1./B(2,1,1)
1020=        L21=B(2,2,1)
1030=        U12=B(2,1,2)*L11
1040=        L22=1./(B(2,2,2)-L21*U12)
1050=        U13=B(2,1,3)*L11
1060=        U14 = B(2,1,4)*L11
1070=        U15=B(2,1,5)*L11
1080=        L31=B(2,3,1)
1090=        L32=B(2,3,2)-L31*U12
1100=        U23=(B(2,2,3)-L21*U13)*L22
1110=        L33=1./(B(2,3,3)-U13*L31-U23*L32)
1120=        U24=(B(2,2,4)-L21*U14)*L22
1130=        U25=(B(2,2,5)-L21*U15)*L22
1140=        L41=B(2,4,1)
1150=        L42=B(2,4,2)-L41*U12
1160=        L43=B(2,4,3)-L41*U13-L42*U23
1170=        U34=(B(2,3,4)-L31*U14-L32*U24)*L33
1180=        L44=1./(B(2,4,4)-U14*L41-U24*L42-U34*L43)
1190=        U35=(B(2,3,5)-L31*U15-L32*U25)*L33
1200=        L51=B(2,5,1)
1210=        L52=B(2,5,2)-L51*U12
1220=        L53=B(2,5,3)-L51*U13-L52*U23
1230=        L54=B(2,5,4)-L51*U14-L52*U24-L53*U34
1240=        U45=(B(2,4,5)-L41*U15-L42*U25-L43*U35)*L44
1250=        L55=1./(B(2,5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
1260=  C     COMPUTE LITTLE R S
1270=        D1=L11*F(2,1)
1280=        D2=L22*(F(2,2)-L21*D1)
1290=        D3=L33*(F(2,3)-L31*D1-L32*D2)
1300=        D4=L44*(F(2,4)-L41*D1-L42*D2-L43*D3)
1310=        D5=L55*(F(2,5)-L51*D1-L52*D2-L53*D3-L54*D4)
1320=  C     COMPUTE BIG R S
1330=        F(2,5)=D5
1340=        F(2,4)=D4-U45*D5
1350=        F(2,3)=D3-U34*F(2,4)-U35*D5
1360=        F(2,2)=D2-U23*F(2,3)-U24*F(2,4)-U25*D5
1370=        F(2,1)=D1-U12*F(2,2)-U13*F(2,3)-U14*F(2,4)-U15*D5
1380=        ENDCD
1390=  C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
1400=  C      COMPUTE C PRIME FOR FIRST ROW
1410=  C
1420=         CODO M=1,5;K=2,KMAX-1
1430=         D1=L11*C(2,1,M)
1440=         D2=L22*(C(2,2,M)-L21*D1)
1450=         D3=L33*(C(2,3,M)-L31*D1-L32*D2)
1460=         D4=L44*(C(2,4,M)-L41*D1-L42*D2-L43*D3)
1470=         D5=L55*(C(2,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
1480=         B(2,5,M)=D5
1490=         B(2,4,M)=D4-U45*D5
1500=         B(2,3,M) = D3-U34*B(2,4,M)-U35*D5
1510=         B(2,2,M) = D2-U23*B(2,3,M)-U24*B(2,4,M)-U25*D5
1520=         B(2,1,M) = D1-U  *B(2,2,M)-U13*B(2,3,M)-U14*B(2,4,M)
1530=        1    -U15*D5
1540=         ENDCD
1550=  C
1560=         DO 13 I=3,JMAX-2
1570=  C       COMPUTE B PRIME*BIGR
1580=  C
1590=         CODO N=1,5;K=2,KMAX-1
1600=         F(I,N)=F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)
1610=        1 -A(I,N,3)*F(I-1)-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
1620=         ENDCD
1630=  C
1640=  C       COMPUTE B PRIME
1650=  C
1660=         CODO N=1,5;M=1,5;K=2,KMAX-1
1670=         H(N,M)=B(I,N,M)-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)
1680=        1 -A(I,N,3)*B(I-1,3,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,
1690=        2    5,M)
1700=         ENDCD
1710=  C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
1720=  C      INSERT LUDEC AGAIN
1730=  C
1740=         CODO K=2,KMAX-1
1750=         L11=1./H(1,1)
1760=         L21=H(2,1)
1770=         U12=H(1,2)*L11
1780=         L22=1./(H(2,2)-L21*U12)
1790=         U13=H(1,3)*L11
1800=         U14=H(1,4)*L11
1810=         U15=H(1,5)*L11
1820=         L31=H(3,1)
1830=         L32=H(3,2)-L31*U12
1840=         U23=(H(2,3)-L21*U13)*L22
1850=         L33=1./(H(3,3)-U13*L31-U23*L32)
1860=         U24=(H(2,4)-L21*U14)*L22
1870=         U25=(H(2,5)-L21*U15)*L22
1880=         L41=H(4,1)
1890=         L42=H(4,2)-L41*U12
1900=         L43=H(4,3)-L41*U13-L42*U23
1910=         U34=(H(3,4)-L31*U14-L32*U24)*L33
1920=         L44=1./(H(4,4)-U14*L41-U24*L42-U34*L43)
1930=         U35=(H(3,5)-L31*U15-L32*U25)*L33
1940=         L51=H(5,1)
1950=         L52=H(5,2)-L51*U12
1960=         L53=H(5,3)-L51*U13-L52*U23
1970=         L54=H(5,4)-L51*U14-L52*U24-L53*U34
1980=         U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
1990=         L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
2000=  C      COMPUTE LITTLE R'S
2010=         D1=L11*F(I,1)
2020=         D2=L22*(F(I,2)-L21*D1)
2030=         D3=L33*(F(I,3)-L31*D1-L32*D2)
2040=         D4=L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
2050=         D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
2060=  C      COMPUTE BIG R'S
2070=         F(I,5)=D5
2080=         F(I,4)=D4-U45*D5
2090=         F(I,3)=D3-U34*F(I,4)-U35*D5
2100=         F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
2110=         F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
2120=         ENDCD
2130=  C
2140=  C      COMPUTE C PRIMES
2150=  C
2160=         CODO M=1,5;K=2,KMAX-1
2170=         D1=L11*C(I,1,M)
2180=         D2=L22*(C(I,2,M)-L21*D1)
2190=         D3=L33*(C(I,3,M)-L31*D1-L32*D2)
2200=         D4=L44*(C(I,4,M)-L41*D1-L42*D2-L43*D3)
2210=         D5=L55*(C(I,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
2220=         B(I,5,M)=D5
2230=         B(I,4,M)=D4-U45*D5
2240=         B(I,3,M) = D3-U34*B(I,4,M)-U35*D5
2250=         B(I,2,M) = D2-U23*B(I,3,M)-U24*B(I,4,M)-U25*D5
2260=         B(I,1,M) = D1-U12*B(I,2,M)-U13*B(I,3,M)-U14*B(I,4,M)
2270=        1    -U15*D5
2280=         ENDCD
2290=  C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
2300=   13     CONTINUE
2310=   C
2320=          I=JMAX-1
2330=   C       COMPUTE B PRIME*BIG R FOR LAST ROW
2340=   C
2350=          CODO N=1,5;K=2,KMAX-1
2360=          F(I,N)=F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)
2370=         1 -A(I,N,3)* F(I-1,3)-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
2380=          ENDCD
2390=   C
2400=   C       COMPUTE B PRIME
2410=   C
2420=          CODO N=1,5;M=1,5;K=2,KMAX-1
2430=          H(N,M)=B(I,N,M)-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)
2440=         1 -A(I,N,3)*B(I-1,3,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,
2450=         2   5,M)
2460=          ENDCD
2470=   C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
2480=  C      INSERT LUDEC AGAIN
2490=  C
2500=         CODO K=2,KMAX-1
2510=         L11=1./H(1,1)
2520=         L21=H(2,1)
2530=         U12=H(1,2)*L11
2540=         L22=1./(H(2,2)-L21*U12)
2550=         U13=H(1,3)*L11
2560=         U14=H(1,4)*L11
2570=         U15=H(1,5)*L11
2580=         L31=H(3,1)
2590=         L32=H(3,2)-L31*U12
2600=         U23=(H(2,3)-L21*U13)*L22
2610=         L33=1./(H(3,3)-U13*L31-U23*L32)
2620=         U24=(H(2,4)-L21*U14)*L22
2630=         U25=(H(2,5)-L21*U15)*L22
2640=         L41=H(4,1)
2650=         L42=H(4,2)-L41*U12
2660=         L43=H(4,3)-L41*U13-L42*U23
2670=         U34=(H(3,4)-L31*U14-L32*U24)*L33
2680=         L44=1./(H(4,4)-U14*L41-U24*L42-U34*L43)
2690=         U35=(H(3,5)-L31*U15-L32*U25)*L33
2700=         L51=H(5,1)
2710=         L52=H(5,2)-L51*U12
2720=         L53=H(5,3)-L51*U13-L52*U23
2730=         L54=H(5,4)-L51*U14-L52*U24-L53*U34
2740=         U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
2750=         L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
2760=  C      COMPUTE LITTLE R!S
2770=         D1=L11*F(I,1)
2780=         D2=L22*(F(I,2)-L21*D1)
2790=         D3=L33*(F(I,3)-L31*D1-L32*D2)
2800=         D4=L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
2810=         D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
2820=  C      COMPUTE BIG R!S
2830=         F(I,5)=D5
2840=         F(I,4)=D4-U45*D5
2850=         F(I,3)=D3-U34*F(I,4)-U35*D5
2860=         F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
2870=         F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
2880=         I=JMAX-1
2890=  200    I=I-1
2900=  C
2910=         CODO N=1,5;K=2,KMAX-1
2920=         F(I,N)=F(I,N)-F(I+1,1)*B(I,N,1)-F(I+1,2)*B(I,N,2)
2930=        1 -F(I+1,3)*B(I,N )-F(I+1,4)*B(I,N,4)-F(I+1,5)*B(I,N,5)
2940=         ENDCD
2950=  C
2960=         IF (I.GT.2)GOTO200
2970=  C
2980=         CODO J=2,JMAX-1;K=2,KMAX-1
2990=         S(K,L,1,J)=F(J,1)
3000=         S(K,L,2,J)=F(J,2)
3010=         S(K,L,3,J)=F(J,3)
3020=         S(K,L,4,J)=F(J,4)
3030=         S(K,L,5,J)=F(J,5)
3040=         ENDCD
3050=  C
3060=  20 CONTINUE
3070=  C
```

Figure 2-2. Proposed Recoding of Scalar 3-D Code Taken From STEP (Cont.)

```
 DIMENSION Q(100,100,6,100),X(100,100,100),Y(100,100,100),
1   Z(100,100,100),D(100,5,5),A(100,5,5),B(100,5,5),
2   C(100,5,5),S(100,100,5,100),F(100,5),H(5,5)

 DO 20 L=2,LMAX-1

    RLOADI L,2        STARTING VALUE
    SUBX LMAX,ONE,&S0001

 CODO J=1,JMAX;K=2,KMAX

    MPYX KMAX,JMAX,&S0002     FORM VECTOR LENGTH MAXIMUM

 RJ=Q(K,L,6,J)

    PACK KMAX,&&DQ1,&S0003     &&DQ1 CONTAINS DIMENSIONS OF Q
    PACK &S0002,&&DSP,&0003    FORM TEMP VECTOR AT DYNAMIC SPACE
    SHIFTI &S0001,6,&S0004     ITEM COUNT CONVERT TO BIT ADDRESS
    ADDX &&DSP,&S0004,&&DSP    UPDATE DYNAMIC SPACE POINTER
    MPX  &&DQ2,SIX,&S0004      COMPUTE SKIP DISTANCE IN Q
    MAP,F=GATHR,R1=&S0003,R3=&S0004,W1=&0003.

 XK=(X(K+1,L,J)-X(K-1,L,J))*DY2
 YK=(Y(K+1,L,J)-Y(K-1,L,J))*DY2
 ZK=(Z(K+1,L,J)-Z(K-1,L,J))*DY2

    PACK &&XL,&&DSP,&D0003
    SHIFTI &&XL,6,&S0005
    ADDX &S0005,&&DSP,&&DSP    UPDATE DSP
    PACK HUNDRD,&DX,&D0004      RECORD LENGTH

    MAP,F=GATHR,R1=&D0004,R3=TNTHSND,W1=&D0003.

    PACK &&YL,&&DSP,&D0005
    SHIFTI &&YL,6,&S0005
    ADDX &S0005,&&DSP,&&DSP    UPDATE DSP
    MAP,F=GATHR,R1=&D0005,R3=TNTHSND,W1=&D0005

    PACK &&ZL,&&DSP,&D0006
    SHIFTI &&ZL,6,&S0005
    ADDX &S0005,&&DSP,&&DSP     UPDATE DSP
    MAP,F=GATHR,R1=D0006,R3=TNTHSND,W1=&D0006

    MAP,R2=&DDY2,D=BCAST.
    BUFF,WB1=S2,E=8,F=000.      SETUP BROADCAST OF DY2 IN BUFFER
```

(Continued)

Figure 2-3. Hand-Compiled Example of a Segment of FORTRAN Code

```
   PACK TNTHSND,&DXK,&DXK
   PACK TNTHSND,&DYK,&DYK
   PACK TNTHSND,&DZK,&DZK
   ADDX SIXFOUR,&D0003,&D0007
   SUBX &D0003,TWOWDS,&D0008
   PACK TNTHSND,&D0007,&D0007
   PACK TNTHSND,&D0008,&D0008

   ADDX SIXFOUR,&D0005,&D0009
   SUBX &D0005,TWOWDS,&D0010
   PACK TNTHSND,&D0009,&D0009
   PACK TNTHSND,&D0010,&D0010

   ADDX SIXFOUR,&D0006,&D0011
   SUBX &D0006,TWOWDS,&D0012
   PACK TNTHSND,&D0011,&D0011
   PACK TNTHSND,&D0012,&D0012

   MAP,R1=&D0007,R2=&D0008,W1=&DXK=VU.
   BUF,A=RB1,B=1,E=0,F=000.    BROADCAST FROM BUFFER
   VEC,F=(A-B)*D,A=S1,B=S2,D=RB1,W1=AR1.

   MAP,R1=&D0009,R2=&D0010,W1=&DYK=VU.
   BUF,A=RB1,B=1,E=0,F=000.    BROADCAST DY2 FROM BUFFER
   VEC,F=(A-B)*D,A=S1,B=S2,D=RB1,W1=AR1.

   MAP,R1=&D0011,R2=&D0012,W1=&DZK=VU.
   BUF,A=RB1,B=1,E=0,F=000.    BROADCAST DY2 FROM BUFFER
   VEC,F=(A-B)*D,A=S1,B=S2,D=RB1,W1=AR1.

XL=(X(K,L+1,J)-X(K,L-1,J))*DZ2
YL=(Y(K,L+1,J)-X(K,L-1,J))*DZ2
ZL=(Z(K,L+1,J)-Z(K,L-1,J))*DZ2
```

Figure 2-3. Hand-Compiled Example of a Segment of FORTRAN Code (Cont.)

Figure 2-4 provides some summary information about the 'assembly language' form in which the FMP code is presented, as compiled by a 'pseudo compiler'. Some of the conventions such as using a special character "&" to delineate compiler-generated variables, descriptors and arrays has been taken from the STAR FORTRAN compiler scheme. A brief description of the code follows.

The operation RLOADI stands for the scalar function LOAD REGISTER with IMMEDIATE data. The value 2 will appear as part of the actual instruction. The register called L will be defined as a permanent register out of the 256 available to the programmer in the Scalar Processor.

SUBX is the operation SUBTRACT INDEX (or address). The register called ONE is canonically defined as register 16 in all scalar units of the STAR family. The temporary register &S0001 is set up to be used at the DO loop termination sequence (not shown in Figure 2-3).

MPYX stands for Multiply Index value (non floating point). The result will be the vector length to be processed for the metric arrays X, Y and Z. The CODO statement permits the compiler to generate a series of GATHER RECORD operations to form a long vector which makes the vector arithmetic more efficient. Figure 2-5 gives a visualization of the matrix as it would be stored in memory if the dimensions of X, Y, Z and Q were (10,10,10) and LMAX, KMAX and JMAX were each 10. The numbers in the blocks indicate their sequential storage addresses. Thus Q(1,1,1) would be block 00, Q(2,1,1) would be block 01, Q(1,10,1) would be block 90, and Q(1,1,2) would be block 100.

The CODO statement creates a vector operation that, for each J, removes from memory a vector of length KMAX-2. This action will result in a new vector consisting (referring to Figure 2-5) of block 00 through 09 followed by blocks 100 through 109, blocks 200 through 209 and so on up to block 909. The GATHER RECORD operation makes a random reference to memory for the first element addressed (J=1,2,3 . . . JMAX) and then retrieves the data following (K=2,3 . . . KMAX) at 'near-pipeline rates' of eight 64-bit operands per minor cycle. The columns of data gathered in this manner are stored sequentially in memory.

A series of scalar instructions preceding the GATHER instruction forms the descriptors to be used by the Vector and Map Units. The instruction PACK merges the rightmost sixteen bits of a register into the sixteen-bit length field of another register (which normally contains the array base address) and places the result in another register (in this case temporary, compiler-assigned descriptors).

Temporary vectors (which will never be transmitted to Backing Store) are assigned dynamically in the same manner as used on the STAR-100 computers. A fixed register, called the Dynamic Space Pointer (DSP) contains the address of the first available location in free (unused) memory. Temporary vectors are allocated and deallocated in this region of Main Memory by using the DSP, and then updating the DSP to the next free location.

The operation SHIFTI (shift register immediate) shifts the field length (which is an item count of 64-bit words) left six places to form the bit address with which the DSP can be updated.

The MAP instruction sets up the READ 1 trunk with the base address of Q and a record length of KMAX (number of words in each column or record), and the READ 3 trunk with the increment used to proceed through memory for every J (refer to previous discussion of Figure 2-5 where J=1,block=00,J=2,block=100; then the memory increment would be 100). The WRITE 1 bus, W1, is set up with the address of the temporary vector, which will later be assigned to the variable RJ. The function code GATHR indicates a GATHER operation. The presence of a field length in the READ 1 setup indicates that the operation is to to be a GATHER RECORD.

- Scalar code uses mnemonics identical to those implemented for the STAR-100 Computer family

- Form for vector machine code is:

  Unit name (Vector, Map, Buffer, or Swap, or the first letter of those names . . V, M, B, S)

  Subfunction field name (R1 . . means READ 1 setup)

  An = sign followed by the value for that field

- All internal scalars created by the compiler are given sequential names beginning with '&S'. Thus the first scalar temporary created by the compiler would be &S0001.

- All internal vector temporaries created by the compiler are given sequential names beginning with '&V'. Thus the first vector temporary would be named &V0001.

- Descriptors (register file pointers to vectors and arrays) are assigned internal names which begin with '&D' and followed by the array name. Thus a vector declared by the programmer as in a DIMENSION statement:

  DIMENSION AAA(100)

  would have a descriptor assigned to it with the name &DAAA. Likewise an internal vector temporary created by the compiler with the name &V0001 would have a descriptor assigned with the name &D0001.

- An example of the form of the language, a memory-to-memory vector addition operation, is given as:

  MAP,R1=&DAAA,R2=&DBBB,W1=&DCCC=VU.

  VECTOR,F=A+B,A=RB1,B=RB2,W1=AR1.

  The READ 1, READ 2, WRITE 1 setups take their base addresses and vector lengths from the register file-contained descriptors &DAAA (for array AAA), &DBBB (for array BBB), and &DCCC (for array CCC). The WRITE 1 setup statement includes the expression =VU, which indicates the WRITE 1 data is to come from the Vector Unit (VU).

  The Vector Unit instruction indicates a function code of a simple add (see 3.2.1.160 of the functional specification for complete list of codes and their representation). The A operand stream will come from S1 (Source 1 from the Map Unit) and the B operand stream will come from S2 (Source 2 from the Map Unit). The WRITE 1 bus output will come from the AR1 trunk (Arithmetic Result 1).

Figure 2-4. Explanation of Machine Language Coding for FMP Produced by FORTRAN Compiler

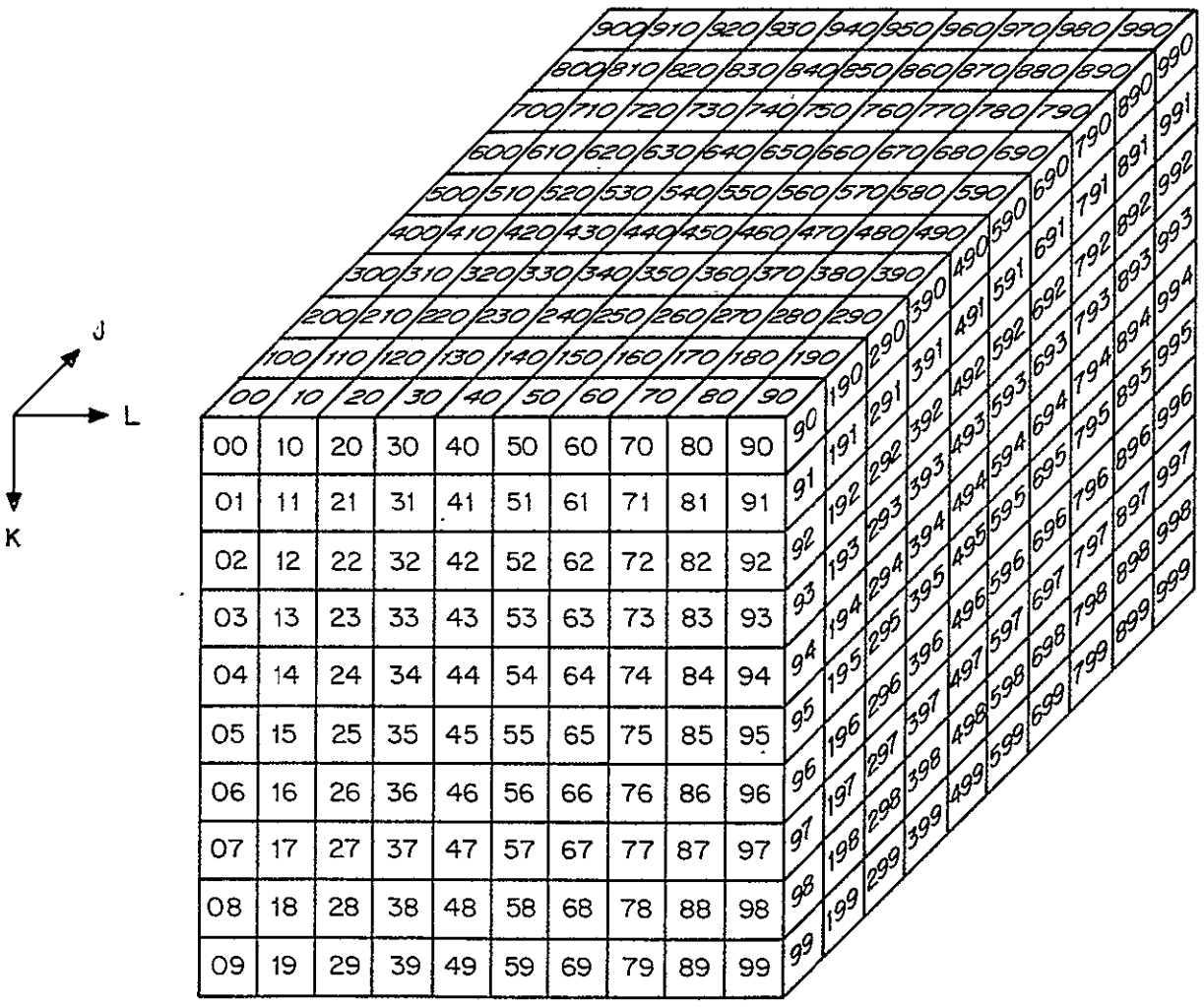


Figure 2-5. Storage Allocation for Flow Variables in Main Memory Allocation

At the conclusion of the MAP operation, a vector of length 10,000 (if JMAX,KMAX and LMAX=100) will be formed in memory and assigned the descriptor &D0003, which will be changed (by code not shown) to &DRJ.

The machine coding which follows basically repeats the descriptor setup sequence just depicted for the elements of Q, but for the metric array elements X, Y, Z. After forming the temporary vector &D0006, the compiler generates the MAP function and BUFFER function which together use the READ 2 stream (R2) to transfer the quantity DY2 to the Buffer Unit as a broadcast operand. Since the GATHER operations use only READ 1 and WRITE 1, this operation can proceed concurrently with the MAP operation preceding it.

The setup of temporary descriptors &D0007, &D0008, &D0009, &D0010, &D0011, &D0012 essentially offsets the starting addresses of the X, Y and Z vectors that have been gathered previously by + and - one word. (adding 64 to a bit base address is the same as offsetting the address by one word).

The MAP, BUF and VEC instruction sets that appear at the end of Figure 2-3 accomplish the vector subtraction of the near-adjacent elements (K+1, K-1) of vector X, Y, and Z. The result of the subtraction is multiplied by the broadcast value of DY2 which has been preloaded into the buffer, producing two floating-point operations per pair of input X operands. In 32-bit mode this process would yield 32 floating-point operations per minor cycle for a computation rate of 3.2 gigaflops. Note that this operation is memory-to-memory since the vectors are too long for the buffer (except for the broadcast quantity DY2).

The next sequence of code would be the formation of the XL, YL and ZL components. A 'dumb' compiler would produce two GATHER RECORD operations for each metric array (X(K,L+1,J) and X(K,L-1,J) instead of remembering that the X(K,L,J) gathered for the XK, YK and ZK metrics computations could be retained in memory and used in the next step of L to provide the L-1 elements.

The amount of 'smarts' necessary to accomplish the retention of previously gathered vectors in X, Y or Z is no different than the intelligence needed to retain the counterpart scalar values between one computation or iteration in a DO loop.

E. Examination of the implicit code in light of memory accesses required revealed that two distinct approaches were dictated. First the assumption that the entire code could be performed in 32-bit mode gave hope that the total data base and all temporary vectors could be held resident in Main Memory. Thus it was necessary to determine first whether this was true, given the explosion of temporaries, when long vectors are created for efficiency reasons. Secondly, the potential need for 64-bit accuracy in the calculations made it obvious that a 64-bit version could not be held in Main Memory; thus access patterns for the Backing Store had to be examined. Finally, it had to be determined if the calculations themselves could be done in the required time given for desired FMP responses to customers.

First, consider the 32-bit case. Using the example of left-hand-side coding given in Figure 2-2, the basic memory requirements are:

1. Flow variables Q(100,100,6,100)=6,000,000
2. Metrics X(100,100,100), Y(100,100,100),Z(100,100,100)=3,000,000
3. Update matrix S(100,100,5,100)=5,000,000

In 32-bit mode this would require 7,000,000 64-bit words of the 8,000,000 proposed for the FMP.

To make the GATHER and subsequent vector computations more efficient by using long vectors, it is desirable to gather planes in the J direction for this segment of the code. Thus there would be resident in Main Memory at any one time in left-hand-side solution, a number of planes of data, each 10,000 32-bit words in length (5000 64-bit words):

1. Six planes of Q flow variables --30,000 words.

2. Three planes of metric data (to keep the L,L+1, and L-1 planes available for the next step in L), each in the X, Y and Z directions for a total of nine planes --45,000 words.

3. A plane's worth of A, B, C and D data with 25 elements for each point in the plane (five by five block) --500,000 words.

4. A plane of the update array S, in gathered form --25,000 words.

5. A plane each for the H and F intermediate matrices used in the BTRI sequences (5 by 5) --375,000 words.

The grant total of large data temporaries is then 975,000 words plus the 7 million words for major variables equals 7,975,000 words. An 8 million (nominal) word FMP actually contains 8,388,608 words of memory. The operating system requires 65,536 thus leaving 8,323,072 words for useful storage. The remainder after allocating known temporaries and flow variables is then 8,323,072 - 7,975,000 = 348,072 words. Since the BTRI sequences will use the vector buffers for intermediate storage, this remainder seems adequate at this time to hold a 32-bit version of the three-dimensional implicit code, in the form available for this study.

It is obvious from this example that a 64-bit version would substantially overflow the Main Memory capacity. If the major portion of the data base for the 64-bit version must reside in Backing Store (flow variables and metrics plus the S array) then the capability must exist to transfer the data required by the calculations at a sufficiently high rate to match the computation rate, in order to achieve the performance objectives of the FMP. The scheme proposed by NASA Ames personnel consists of storing data in the Backing Store matrix in a manner different from that used in the 32-bit mode. In this 64-bit mode case, a basic transfer block of 32,768 for the Backing Store has packed in it all of the variables needed for a given point in the mesh. Thus the six Q values and three metric (X,Y,Z) values would be stored in a single physical block. The vector lengths for each of the nine vectors in the block would be 32,768/9=3640 elements maximum, which is efficient for the Vector Unit and Map Unit to process memory-to-memory or within the vector buffer (which is 8192 words long).

If the dimensions of the mesh are 100,100,100 (referring again to the storage scheme of Figure 2-5) a non-integral number of columns of data from the mesh will reside in a block. Since the intent is to move data from the Backing Store in 'slabs' (see Figure 2-6), it would be better to always allocate integral rows and columns to physical storage blocks. Thus for a 100,100,100 mesh, 30 columns of each of the major variables would be stored in each physical block. This means that Q(1,1,1,1) through Q(99,3,1,1) would be stored contiguously, followed immediately by Q(1,1,2,1) through Q(99,3,2,1). The last flow variable in the physical block Q(99,3,6,1) would be followed by the first metric X(1,1,1).

The computational sweep in the J direction would then require the transmission of the first physical block, the nth physical block, the 2nth block, and so on until a 'slab' in the J direction is completely transferred to Main Memory. Figure 2-6 shows such a slab for a mesh of 10,10,10, for a single variable. The physical blocks in this case contain the following:

Physical Block 1 = blocks 00-29

Physical Block 2 = blocks 30-59

Physical Block 3 = blocks 60-89

Physical Block 4 = blocks 90-99

Physical Block 5 = blocks 100-129

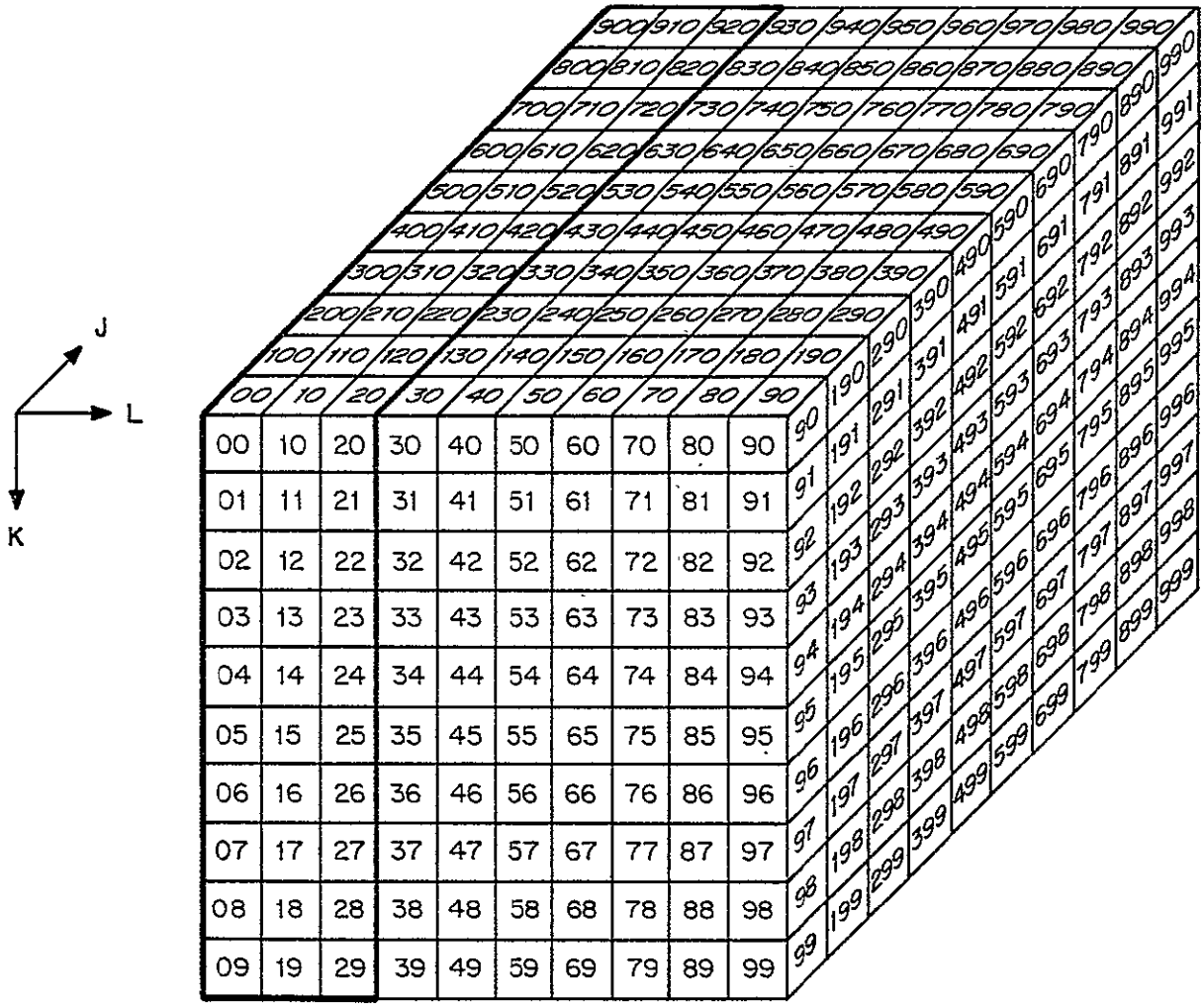


Figure 2-6. Slab Slicing of Matrices for Backing Store Transfers

Proceeding to input a slab in the J direction for Figure 2-6 then would transfer physical block 1,6,11,16 and so on.

Note two effects of this storage system:

1. A certain percentage of the physical block transfers contain no useful data. Thus the transfer efficiency is affected by that amount.

2. The maximum length vectors without performing a GATHER operation are an integral number of columns in length, and the GATHER operation in any direction is now aperiodic since the data for J=1,2,3 is not stored at regular memory intervals because of the 'holes' left in some physical blocks, and matrices are no longer homogenous (elements of other matrices are stored sequentially imbedded in other matrices in actual physical storage).

The programming of this technique utilizing BUFFER IN/BUFFER OUT and CODO constructs (see Section 4) remains as a great challenge to enliven the next phase of this study.

A slab in the J direction would require the transfer of one hundred physical blocks (or ten blocks if the dimensions were as in Figure 2-6) between Main Memory and Backing Storage. This would mean the allocation of 100*32,768 or 3,276,800 words of Main Memory for buffering of the slab. In the J direction, computation (and the necessary SCATTER/GATHER operations) cannot proceed until all data is in place in Main Memory. To keep the computational rate up, a slab must be moved into a buffer while calculations are being performed on a different slab in another buffer. This would then require 2*3,276,800=6,553,600 words of Main Memory, with no room to hold the S matrix. Thus the S matrix may have to be combined with the Q and X, Y, Z matrices in physical blocks. It appears that sufficient Main Memory capacity exists to support this scheme, however.

If this allocation and transfer scheme is feasible then the transfer rates must be investigated. The method employed by this 'slab' technique implies a full transfer of all mesh, metric, and update matrix variable into and out of Main Memory during the J direction sweep. It then appears possible to bring in physical blocks constituting the full plane at J=1 and to solve both sets of equations (K,L sweep) while the plane remains in Main Memory. Referring to the example in Figure 2-6, this would mean holding blocks 00 through 99 in Main Memory for the solution in the columnar and row-wise directions. The amount of data transferred would be one full transfer of all variables for a single K,L sweep of the mesh. Total data would then be 14 million words (9 million for flow variables and metrics and 5 million for S update array). If all the metric, mesh and update data is intermingled in physical blocks, then all data must be moved both ways (to and from the Backing Store) even though many variables such as the X, Y, Z metrics are not updated, and thus would not otherwise find their way back to Backing Store. Taking the two sets of transfers then, 14 million words would be transferred to Main Memory for the J sweep, 14 million for the K,L sweep, and the entire 14 million transferred back at the end of that particular processing. Four transfers (counting both directions) of 14 million words equals 56,000,000 words per time step. If 256 time steps is a representative example then $1.434*10^{10}$ words would be transmitted for a single problem solution taking about eight minutes.

This volume of data transfer would require 30,000,000 words per second to be moved at sustained rates. This becomes .3 words per clock cycle. The Backing Store is capable of providing .2 words per clock cycle with the present design at a sustained rate. The conclusion to be drawn is that a different physical block allocation scheme should be devised which reduces the total data transferred by at least 10 percent, since the volumetric efficiency of the physical block storage (because of the holes left in the blocks) is not 100 percent, and the transfer rate is slower than desired. Alternative transfer rates are possible with the Backing Store, but should be decided on quickly since they affect the major structure of the Backing Store design.

With the limitations just discussed it appears that the memory access patterns for Backing Store transfers are sustainable for the known 3-D code. The 32-bit mode version exhibits similar access difficulties, but recent analysis (see results of simulation in this section) indicates that the Map Unit can support the computational rate required under worst-case conditions -- in 32-bit mode.

CONCLUSIONS OF ANALYSIS IN THIS INTERIM PHASE

A great deal is being learned about the behavior of the implicit three-dimensional code when programmed for the FMP. It appears that more analysis must be done involving the tradeoffs between cost, performance, size and bandwidth of the memory systems when concentrating on the implicit code. For example, it is not altogether clear that a much lower-cost, higher-capacity Main Memory would not be more desirable than the system presently proposed for the FMP. Programmability and compiler optimization are severely impacted by the need for clever slicing schemes to move data between Backing Store and Main Memory.

The GPSS model that is being developed may provide the necessary tool for trying different forms of Main Memory and Backing Store. The next phase will provide the opportunity to hand-compile the balance of the key segments to determine the best memory approaches.

The major analytical effort in this phase ended up focusing on storage capacity and bandwidth, and to a minor degree on the capability of performing non-sequential access to memory for the J sweeps. These have emerged as first-order effects on the performance of the FMP. In the next study phase, the second-order effects must also be analyzed more closely to determine if the pipelines and Map Unit can be overlapped efficiently (and without great programming difficulty), and whether they are properly matched to the problem.

FUNCTIONAL DESIGN

The Phase 1 Extension effort produced a proposed structure for the FMP and the overall system in which it is to be imbedded. In this report all hardware design has been focused on the FMP and in particular those parts which are newly designed and not borrowed to some extent from the STAR-100 family. Thus considerable effort has been applied to the Vector and Map Units which are the key to the arithmetic bandwidth of the FMP. The description of the resulting design may be found in hardware specification form in Appendices A and B of this report. The Instruction Specification gives the summary of how the FMP performs each operation, whilst the Functional Specification illuminates such material as the behavior of the particular arithmetic system chosen, transfer and clock rates, and maintenance interface data.

BLOCK DIAGRAMS

The revised block diagrams of the FMP can be found in the Functional Specification. More detailed design work has been done on the Map Unit, but the level of block description given in the specification represents that in which confidence can be placed at this time. Formal specifications for the Programmable Device Controller are currently under way, with completion due by summer 1978. Detailed specifications for equipments already in place in standard systems, such as the high performance disks, have been omitted. An unscheduled and extraordinary amount of the total project effort has been expended on the analysis and redesign of the FMP and its instructions. This factor led to other portions of this study being reduced in scope over what was originally planned. This was essential, since the programming and timing estimates and the block simulation efforts had to await the creation of a workable engineering design of the new components.

Comparison With/Difference From Phase 1 Design

Examination of the block diagrams and description in the Functional Specification will reveal several major changes in design since the release of the Phase 1 Final Report.

The most significant change is the more detailed structure of the Vector Unit, wherein there are three sets of identical functional elements (two front-end adders, two multipliers and two back-end adders) and three checking elements instead of the two called out in the Phase 1 plan. Further, the fully general interconnection scheme of Phase 1, wherein any element could be connected to any other, has been reduced to a more practical (from an engineering and parts count point of view) set of interconnections. This constraint then led to the definition of explicit interconnections called out by the allowable arithmetic instructions. The result of this redesign is a significant reduction in hardware, and a substantial increase in the checking of results. This is due to the fact that although there are six separate arithmetic elements, the front-end adder is not a full floating-point adder, but contains only the prenormalize networks needed for initiating a floating-point addition, and the back-end adder has only the post-normalize network, and is shared by the multiply element for forming the final sum of all partial sums and carries generated by the multiply element. Although this back-end adder is shared by the multiply element, an auxiliary port (which required very little hardware) has been provided to bring in another operand to be added to the product. Checking probabilities are enhanced since at any one clock cycle two of the six elements will be idle (and thus possibly checking their partners) because of the constrained instruction set that permits a maximum of three floating-point operations to be called out at a time (for example A+(C*D) leaves one multiplier and one back-end adder free).

SECDED is now carried within the Vector Unit on all trunks not imbedded in arithmetic elements, rather than parity bits.

The vector buffer, while still physically contained within a Vector Unit, is programmed separately with its own instruction (9E).

SECDED is now carried within the Buffer Unit, instead of simple parity.

The Buffer Unit has only two READ ports instead of four. This reduces the hardware parts count and also the risk of not being able to get fast enough RAMs for this unit by 1980.

Nine Vector Units (vector pipelines) form an arithmetic ensemble instead of eight, as described in the Phase 1 report. A simple method for providing ports for the nine units in the Map Unit, and a method for switching one pipeline off-line and another on-line, made it possible to provide for quick recovery in the event of a pipeline failure.

All bit addressing has been eliminated from the Map Unit. All bit strings used for control vectors and order vectors must begin on a 32-bit boundary. Alignment of bits strings to these boundaries can be done by the Scalar Processor with its copious idle time. To facilitate string logicals and alignment, the Scalar Processor has added to it two double-length shift instructions (operation codes 20 and 21 hex) which are not available on the STAR-100.

The Swap Unit has been given a Backing Store imap table to enable setting regions busy for monitor purposes, or to permit explicit input/output in a limited form to be performed by the job mode program.

Instruction Specification

The instruction specification may be found in appendix A of this report. This specification gives the behavior of all FMP instructions. To permit using portions of the actual STAR-100 design and its documentation system, the FMP Instruction Specification was designed to not overlap the STAR-100 features, but to appear to be a mutually exclusive functional entity. Thus instead of changing an existing instruction (such as Vector Add Upper, op code 80) to become the Swap Unit instruction, the FMP defines such an instruction as illegal, and uses one of the STAR-100 illegal instructions as the SWAP instruction (56 op code). The purpose in this is to permit future expansion of the FMP instruction set to embody desirable STAR instructions, but more importantly to open the avenue of STAR-100 simulation of the FMP and vice-versa, since all instruction decode and control on both machines is done with microcode.

Functional Specification

The Functional Specification may be found in appendix B of this report. Certain functions have not yet been defined, and are indicated by the phrase "to be defined", or "designed at a later time", or some similar disclaimer. Certain other functions which correspond to their STAR-100 counterparts have not yet been designed, but the STAR-100A feature is described to give the "flavor" of the function. An example would be the description of microcode loading and diagnostic control for the FMP which is, as an interim, taken directly from the STAR-100A Functional Specification.

Rationale for Design Approaches

After a year of consideration, analysis, design and redesign, it is felt that the FMP structure given in the included specifications represents a reasonable engineering approach to providing computations in excess of one billion per second. As the design undergoes more detailed study some changes are made, and in other cases convictions grow stronger regarding the approaches taken. Several items that were examined following the release of the Phase 1 report, and in response to commentary on that report are presented here.

Logic Family

A second-generation logic family of high-speed ECL LSI was proposed as a means for keeping the "real estate" and the parts count for the FMP down to a reasonable level, not to mention the reduction of power and cooling requirements. As was pointed out, this choice of a non-existent but promising technology was the leading technological risk for the project. RADL personnel still feel that pursuit of this goal of a denser LSI parts (LSI-II is important to the project but not essential to meeting the stated goals of performance and reliability). This can be achieved by continuing the reduction in complexity of the FMP hardware as understanding of its behavior with the actual mathematical codes improves. Efforts are underway to carefully analyze the design and construction of the FMP with existing LSI chips and packaging.

On the other hand, the desirability of a cooler, smaller, faster LSI system for the FMP has motivated the pushing of semiconductor manufacturers to pursue the next-generation technology. For the purposes of this study, all estimates of space, power, and speed are based on the currently pursued parameters for this new generation. In future reports such quantities will be stated in terms of construction with either family of logic.

Parts Count vs Performance Tradeoffs

One of the major engineering concerns regarding the buildability of a machine as powerful as the FMP is the sheer numbers of parts, and hand-tooled interconnections as they affect reliability and maintainability. The nine pipelines proposed for the FMP are felt to be the limit of the number of parallel functional units that should be assembled in one place with the existing technologies. At present, these pipelines can produce 4800-million 32-bit floating-point results per second (at a 10 nanosecond clock cycle), peak rate. If studies indicate that 32-bit mode only could be applied to codes using the FMP, and if code analysis proves that the FMP could sustain computations at 50 percent of peak rate, then half as many pipelines would be needed (actually 5 instead of 9) with a consequent major reduction in hardware parts and a commensurate improvement in reliability.

Pursuant to this parts reduction, the original generalized Vector Unit, with four read ports from the Buffer Unit and total interconnectability, created larger parts counts per unit than was felt desirable. Based on an analysis of the 3-D implicit code, it was discovered that a 20 percent reduction in parts in those areas would affect the performance of the FMP on that code (counting only vectorized processes) by less than 3 percent (thus yielding an overall affect much less than 3 percent). It is in areas such as these that continued analysis of the codes, and their match with the hardware, is beginning to pay off.

Choice of Instructions

The basic instruction set for the FMP was derived from the STAR-100A. This provides leverage on the generation of diagnostics and utilities which can be executed in the Scalar Processor alone. Thus when the FMP is first powered up and in checkout, the wealth of software and checkout experience gained from the STAR-100A project can be applied to the FMP. With a completely checked out Scalar Processor and input/output system, checkout and maintenance of the remaining units can be greatly facilitated.

Instruction extensions to the Scalar Processor were minimized to reduce the need for new scalar diagnostic development, and to limit somewhat the functions accessible by the monitor operating system. The two monitor instructions for setting up the Swap Unit, Map Unit and for communicating with the PDC were considered the basic minimum necessary to support the operating system. A vector SWAP counterpart to the STAR-100 register file SWAP, then was included for Backing Store interchange. Finally, by limiting the actual VECTOR and MAP instructions to three, the remainder of the STAR instruction set could be made illegal or legal depending on needs of the software implementors (for simulation purposes) without supplanting from the instruction set the unique FMP instructions. Note that the 9D and 9E are variable length instructions depending upon how many individual streams are being set up, while the 9F is a fixed length vector instruction executed by the Vector Unit. In effect, each of the FMP vector instructions (Map, Buffer and Arithmetic) are somewhat microcoded versions of the original STAR vector operations. This ability to "build your own vector" is useful when generating object code for computations such as the Navier-Stokes solutions.

The choice of a microcoded level of instruction also somewhat simplifies the engineering since, in many instances, a single bit in an FMP vector instruction controls a single gate or single fanout in the processor without the need for extensive decode and timing controls.

Extensibility

As noted above, at the discretion of the system developers, other STAR-type instructions could be invoked, either to be implemented by simulation or by additional hardware as the particular case warrants. Within each FMP instruction extra room has been left for extensions to be defined later. For example, all address fields are larger than the maximum allowable memory space requires today. Thus, although 8-million words of bipolar RAM appears to be the practical limit at this time for main memory construction, addressing has been retained for 32-million words. Likewise, the Backing Store addressing permits addressing 1-billion words of data, while only 256-million words seems practical or necessary at this time.

Finally, a number of instructions have been left undefined for both the STAR-100A and the FMP, both in the 32-bit scalar class and in the 64-bit vector class. As a function comes to light that would be desirable for STAR, a similar function might be included in the FMP.

BLOCK-LEVEL SIMULATION

One of the methods considered best for measuring the behavior of a particular design is to develop a computerized model of that design, submit to that model characteristic code sequences, and extract the predicted execution of those code sequences in the face of conflicts for resources such as memory, functional units, and input/output buses. To provide NASA with a tool which can be used to measure various kinds of computations on the FMP, a simulation model could be built which could then be run by NASA personnel at their initiative on whatever computers were available. Thus, various analytical teams could examine machine performance in their areas of interest.

The purpose of this task was to supply a package containing the necessary materials to permit Ames researchers to create input decks of programs that might be run on an FMP and then run these programs through a model (provided as part of the package). The resultant data could provide timing information, location of significant bottlenecks, and storage access patterns for review of the Backing Store and

input/output strategies. At the time of this writing the package is not yet in a form which can be used by NASA, but it should be ready by the time of submission of the final draft of this report.

Simulation With GPSS vs LSISYS

The large-scale computer development operation of Control Data (STAR-100 and FMP being examples) bases its design and construction cycle on a series of automated design and documentation tools, which utlimately produce magnetic tapes that are then used to automatically (or nearly so) fabricate the silicon chips and main circuit boards for the object computer systems.

Three levels of simulation of a design provided in this system, called LSISYS (Large Scale Integration Simulation System), are General Block-Level Simulation, Detailed Block-Level Simulation, and Gate-Level Simulation. The first, and most general, of these consists of writing FORTRAN subroutines representing the behavior of a given block of logic (say the entire Swap Unit), and integrating them into the LSISYS main program. The second level, which can only be started when actual machine design is underway, consists of utilizing a library of basic logic blocks (for example, a basic 32-bit wide adder network among others), selecting the desired building blocks, placing them on "pseudo-boards", and interconnecting all data trunks and controls as groups of wires. The third and most detailed level of simulation consists of placing actual arrays on actual boards (with software) and routing all real signals (by software). This final stage represents a complete verification of the design with the hardware as it will actually be built.

Each level requires a certain amount of resource to prepare, with the least resource required by the highest, most coarse model for the machine and the most resources required by the gate-level simulation. Machine time requirements can be as high as 150 times greater for the gate-level simulation than for the same functions modeled at overall block level. The major advantage in using this LSISYS implementation is that any block or group of blocks may be replaced with their counterpart detailed block or gate models, and the entire assemblage run as one whole unit. Thus as design of various components proceeds at different speeds, the entire ensemble can be simulated to validate a particular gate design without requiring the remainder of the machine to be at the same level of gate design. The running time advantages of running in this mixed mode are also quite substantial over running the entire computer simulation at the gate level.

The disadvantage of LSISYS is that one must be quite familiar with the inner structure of the simulator in order to write generalized block models which can be incorporated. Further, the intimate interconnection of simulator flags, variables, and parameters makes integration a lengthy affair of compilations involving the entire simulator. For more detailed design verification and analysis this disadvantage is outweighed by the amount of discrete information that can be obtained from the general level of simulation, while detailed block and gate simulation are easier to submit to simulation in all cases.

At the same time as RADL investigators began to discover the amount of resources needed to integrate a basic block model of the FMP into the LSISYS simulator, Ames personnel disclosed their most immediate needs for simulation data. It became obvious that the level of detail required at this point in the FMP project was even more superficial than planned for the LSISYS block model. In particular, general statistics about the performance of the Main Memory and Backing Store under different loading conditions is a key concern of both RADL and NASA. Such a general, statistical model could be provided by more readily available, standard simulation systems such as ASPOL, SIMULA, or GPSS, which have been in existence on a variety of computer systems for many years.

The peripheral disadvantage of using the LSISYS mechanism for this evaluation function by NASA analysts was that the system is not a standard, supported Control Data product, which is generally available. Instead, it is used and supported by the STAR Development Division, and can only run on a fully-configured 7600 system. Choosing a general-purpose simulator such as GPSS could then make the basic block simulator system available on a more general class of computers.

A brief examination of the simulators available to RADL via CYBER services and a survey of simulator experience among the project staff led RADL to select GPSS (General Purpose Simulation System) since it and a resident expert were readily available. Its availability on the general CYBER computers as well as IBM machines housed at Ames further justified the choice.

It is expected that as design continues the GPSS model will be refined, and selected simulation analysis of throughput conducted in that system. As the detailed block-level design proceeds however, input will be prepared for LSISYS so that the actual hardware design can be verified for functional, as well as performance, characteristics.

Methodology for Simulation

The FMP has been subdivided into the components of Main Memory, Backing Store and processing unit models, which are linked together. While the models for the memories are developed to represent as close as possible the actual hardware construction, the balance of the processor model consists of an instruction interpretation and decode, and execution timing segments for the swap, vector and map functions invoked by the decoded instructions. This elementary model does not process actual data, although vector addresses and lengths are examined from the input source code.

The input to the model is a series of machine instructions, memory-contained descriptors for vectors, and initial register file contents. The simulator assumes that it is dealing solely with a job mode computational program and is able to time the execution of sequences of instructions, and to produce the memory access patterns resulting from various influences. While the model is running, input/output activity can be simulated by either random, or block sequential accesses. Output data is limited to execution time by instruction and memory activity at this time.

The simulation package to be delivered under separate cover with the final draft of this report will also include an itemized list of the regions of source input (to GPSS) where CYBER/IBM incompatibilities exist, and necessary corrective actions to be taken if the system is to be run on IBM equipment.

After delivery of the simulator, and for the duration of any subsequent phases in which RADL is involved, design updates will be provided for the simulator. To this end some form of documentation and simulator update control system will be devised.

Relationship to Future Simulation

The GPSS level simulator is intended primarily for the use of analyst teams outside RADL to evaluate the ability of the FMP to meet project objectives. It is not capable of being utilized as a verification that the specified hardware is capable of being built. That function still rests with LSISYS which is the primary design tool for the RADL FMP designers. Thus GPSS can be viewed as a management tool, while LSISYS

is the designers tool. Since the input forms and even the amount of detail submitted to the systems differ widely, some form of controls will need to be introduced to ensure that the LSISYS model and the GPSS model actually represent the same FMP, for as long as GPSS is used by NASA personnel to validate the FMP approach. Since LSISYS is capable of yielding the same data with a higher degree of refinement and more closely represents the hardware being designed, it is suggested that, despite its stated disadvantages, LSISYS be used as the continuing design review mechanism by NASA.

In short, GPSS provides a well-documented and expeditious means of getting some simulation results for NASA use, but LSISYS is the long-run solution to ensuring that the real design meets Ames objectives.

Results of Simulation

Development of the hand coding of the implicit code, and the GPSS model to reflect accurately the design of the FMP to date, has severely restricted the amount of data that could be obtained before the conclusion of this phase. It was decided to encode a small portion of the J sweep that is intensively 'memory limited', as data must be gathered for each J in the K,L planes. The computation of the metric differentials and formation of the RHO variables into a long vector were hand coded for Figure 2-3. The results of performing 34 scalar operations, 3 GATHER operations and 3 vector arithmetic operations over a 100, 100,100 mesh is the achievement of a 933-megaflop computational rate. This sequence was chosen as the worst case, since memory is being accessed in the most inefficient way.

The same computations in the L direction would achieve close to 3.2 billion floating-point operations per second, while those in the K direction would attain a rate of 1.8 billion floating-point operations per second..

A listing of the GPSS output is too voluminous to include in this report and is being delivered to Ames under separate cover. This listing contains many statistical counts to illustrate the internal behavior and, in particular, the bottlenecks encountered by a given code sequence.

In the next phase of this study effort, the complete left-hand-side calculation will be coded in extended FORTRAN, hand-compiled, and passed through the simulator.

Section 3.

RELIABILITY ASSESSMENT

Section 3

FMP RELIABILITY ASSESSMENT

INTRODUCTION

The Phase I study for this project produced some reliability projections of a general nature as part of the Final Report. That phase did not include design beyond a conceptual stage and, therefore, reliability data, particularly for the FMP, was primarily subjective. One of the tasks for the Phase I contract extension was to carry the FMP design to a point which would permit making preliminary parts count estimates. These, in turn, could then be used for a reliability assessment with more credibility since it possesses a sounder base.

METHODOLOGY

Since the FMP design proposed by Control Data is based on the STAR-100A, its technologies, and extensions of them, considerable actual detail exists and could be exploited. The basic technology, that of the STAR-100A, is ECL LSI-I, 168-gate array integrated circuits in 52-pin leadless carriers, packaged up to 150 per printed circuit board.

The reliability analysis of the FMP was done by functional unit. Some of the units are anticipated to be the same as those of the STAR-100A (or very nearly so). For these units actual parts counts were used for printed circuit (PC) assemblies already designed. In addition, sufficient data was available to define an average PC assembly (or model). The units to be developed uniquely for the FMP were then defined sufficiently to determine an estimated number of assemblies (or boards) per unit. The model was then applied to these board counts for a failure rate per functional unit.

RELIABILITY ANALYSIS

MODEL PC ASSEMBLY

The model, or typical PC assembly mentioned above, was derived from existing designs. The items which contribute virtually all the failure mechanisms for this assembly are stated in Table 3-1 with counts and failure rates.

TABLE 3-1. MODEL PC ASSEMBLY

Component Counts and Failure Rates

Component	Count	Expected Failure Rate (per million hours)
Board Vias	18,500	0.00005
LSI Connectors	8,468	0.0014
Solder Joints	660	0.0003
Capacitors (ceramic)	300	0.014
Omega Resistors (buried in board)	1,323	0.0004
LSI ICs	147	0.2

By extending these counts and rates and summing the results, a failure rate of 0.0465 per thousand hours is obtained for the model, or average, board (assembly).

RELIABILITY PROJECTION BY FUNCTIONAL UNIT

The current STAR-100A Scalar Processor was used after deleting the associative memory portion. The STAR-100A memory module was used but with substitution of an anticipated 4K memory chip for the present 1K chip. This resulted in using 0.2×10^{-6} failure rate in place of 0.1×10^{-6} which is the rate for the 1K chip. This module was also used for the input/output buffer.

The existing design for the PDC and 50-Mbit data set have also been directly applied to the FMP. This is also true for the memory fanout, so for these three items, actual parts counts with their expected failure rates could be used.

The remaining units, or items, required estimation based on the level of design to date. These are the Map Unit, Memory Interchange, Vector Unit (9), Input/Output Distributor, and Backing Store. The Backing Store design is based on 65K CCD chips, 128 per board. The other units, or items, are new designs and have progressed to a reasonable block-level. Board count estimates were made and the above model board was used as basis for reliability analysis. Table 3-2 lists all the above units, their estimated board (PC assembly) counts, and approximate chip (IC) counts/board.

TABLE 3-2. BOARD COUNT BY UNIT

Unit	Estimated Board Count	Approximate Chips/Board
Scalar Processor	14	147
Main Memory	1216	181
Memory Interchange*	4	147
Memory Fanout	6	112
I/O Unit - PDC**(8)	144	150
I/O Buffer	38	181
I/O Distributor	2	147
Map Unit	6	147
Vector Unit***(9)	72	147
Backing Store	2100	143

*Includes Swap Unit
**Includes Data Set
***Includes Buffer Unit

The Memory and Input/Output Buffer consist of high-speed, bipolar chips packaged in a stack configuration; each stack is 32,768 39-bit words (includes SECDED). The above board counts are for 8 million 64-bit words (plus SECDED) of Main Memory and ¼ million 64-bit words (plus SECDED) of Input/Output Buffer.

The PDC and data set comprise somewhat different technology in that the PDC consists primarily of TTL circuits and the data set is basically an analog device. In addition, these units are packaged differently and employ air cooling.

The Backing Store design, for this analysis, assumes a 65K-bit CCD chip. It also would be packaged differently than the LSI logic of the FMP. The above board count is for a Backing Store of 268 million 64-bit words (plus SECDED).

The results of the analysis are presented in Table 3-3 which gives both raw failure rate and an expected rate after taking SECDED into account. In addition, the mean time between failures (MTBF) by unit are shown.

TABLE 3-3. EXPECTED FAILURE RATE BY FUNCTIONAL UNIT

Unit	Expected Failure Rate (per thousand hours)		Percent Checked by SECDED	MTBF (hours) with SECDED
	Raw	With SECDED		
Scalar Processor	0.9319	0.8387	10	1,192
Main Memory	21.2448	0.0212	99.9	47,170
Memory Interchange*	0.1860	0.0930	50	10,753
Memory Fanout	0.2244	0.1122	50	8,913
I/O Unit – PDC**(8)	0.2896	0.2896	0	3,453
I/O Buffer	0.6639	0.0007	99.9	1,428,571
I/O Distributor	0.0930	0.0650	30	15,384
Map Unit	0.2790	0.1674	40	5,974
Vector Unit***(9)	3.3480	3.0132	10	332
Backing Store	67.6500	0.0677	99.9	14,771
Overall	94.9106	4.6687		214

*Includes Swap Unit
**Includes Data Set
***Includes Buffer Unit

A factor of 1000 improvement was used for SECDED correction in the memory functions which seems conservative because of the size of the memory (assuming good memory maintenance). This is because one chip contains only one bit of a word and two failing chips have low probability of being in the same word. However, if this assumed SECDED correction improvement is off by a factor of 10, the total rate with SECDED would only change from 4.6687 to 5.4740; SECDED effectively removes memory as a large contributor to the system failure rate.

The raw failure rate indicates a system failure roughly twice a day with over 90 percent of these failures being memory circuit chip failures (4K bipolar and 65K CCD). SECDED improves this to an expected failure rate of about 3.4 per month. The majority (2/3) of the remaining failures with SECDED included are in the Vector Units. Because of the great difficulty in providing correction of results in high-speed arithmetic pipelines, a 9th functional pipe line has been added to the 8 required for operation. This, along with the self-checking features built into the pipe, means that a failure in a pipeline can cause the Maintenance Control Unit to replace the failing pipe. The job that was running at the time of the pipeline failure is lost but no further time is lost. The failing pipe is connected to the maintenance processor to be fixed off-line.

EFFECTS OF LSI-II ON RELIABILITY

There are two circuit developments that can have an effect on system reliability using LSI-I circuits: going to a new generation of ECL, LSI circuits and using a 256K CCD chip. It is hoped that a circuit density improvement of 4 times can be achieved. This development should reduce the number of logic boards by a factor of three (not four because it is expected that the number of coax connections on a board will not increase by a factor of 4 but perhaps by a factor of from 1.5 to 2). The number of boards in a pipeline is then expected to be reduced from 8 boards to 3 with reliability improving by a factor of approximately 2.

The 256K CCD chip enables the Backing Store to be built with about 550 boards instead of 2,100. A reliability improvement of about 2 can also be expected here. (The boards become slightly more complex, the SECDED improvement factor decreases slightly, and the more complex chip will have a higher failure rate.)

These changes can be expected to yield the failure rates per thousand hours shown below:

	Raw Failures	Failure Rate With SECDED	MTBF (hours) With SECDED
Logic boards	2.68	2.290	437
Memory	22.57	0.023	43,487
Backing Store	33.83	.034	29,412
Overall	59.08	2.347	426

The raw failure is improved by 38 percent and the SECDED failure rate by 50 percent.

FMP AVAILABILITY ASSESSMENT

Admittedly, some of the above material is subjective; this is of necessity at this early design stage. Detailed design is lacking on most elements of the FMP, component and board counts are estimated, effects of SECDED are speculative since sufficient data are not yet collected, and device failure rates and modes are not yet established for devices of the future.

Since the failure rates previously mentioned show that raw failures are dominated by Main Memory and Backing Store, and since these FMP units are ideally suited for the application of SECDED, a more objective analysis of maintenance strategy for failures covered by SECDED is presented. Ideally, over 90 percent of the raw failures in the FMP could be essentially eliminated from the overall failure rate if SECDED could effectively cover the failures by correcting single-bit failures. The improvement factor of 1,000 used above for SECDED results in MTBF of about 5.4 years for Main Memory and about 1.7 years for Backing Store. This leaves the overall MTBF of 214 hours dependent mostly on the Vector Units which have a spare unit that can be switched off-line for repair.

The computer industry, throughout its history, wherein memories without error correction have been utilized, has developed a cultural habit of immediately removing all symptoms (intermittent or solid) of memory failure. With the advent of error correction (SECDED) in a memory system, single-bit memory failures are corrected automatically, thus deferring the necessity of removal until a more convenient time. Allowing single-bit failures to accumulate in the memory eliminates the consequent emergency maintenance time and reduces remedial maintenance time. This not only increases the time available to the customer but reduces the cost of memory maintenance.

To realize these benefits a memory maintenance strategy must be developed which allows the accumulation of single-bit failures, with sufficiently low risk of system (double-bit) failures. This is done by exploring the risks involved with accumulating single-bit failures in the memory until a planned periodic remedial maintenance period. Such failures are corrected by the SECDED mechanism making the system appear to have no failures, thereby improving the effective failure rate (or MTBF).

In memory systems having parity checking, all failures are considered fatal because the information stored or read has no credibility. The term "Fatal" is used because usually operation is halted, and emergency diagnosis and remedial action is taken to restore confidence in the memory. A memory system utilizing SECDED corrects single-bit failures and detects double-bit errors which are considered fatal for the above reason. It follows, then, that single-bit failures can be accumulated until a double-bit failure occurs. At that time the memory could be swept clean of all failures restoring confidence in the system.

Such a memory maintenance strategy may be acceptable in some circumstances. However, since the double-bit error like the parity error could occur at any time during customer use, its occurrence could be quite costly.

On the other hand, fatal failure can be predicted as likely to occur at or beyond some specified time in the future, given that the failure rate of the storage devices is specified. The probability, P_F, of such an occurrence may be chosen so that it is quite likely that no fatal failure would occur during use (the maintenance interval) if maintenance were scheduled at or before that time to remove all single-bit failures. This strategy calls for removing all storage device failures every maintenance interval, M, such that the probability of a fatal failure is no greater than P_F.

The probability, P_F, of a fatal failure (double-bit) occurring during a maintenance interval is one minus the probability of success, no fatal failures, during the maintenance interval. This probability of success is the product of the probabilities that the next failure will not be fatal for each interval between failures within the maintenance interval. These latter probabilities of success are one minus the probability that the failure will be fatal (double-bit). Thus:

$$1\text{-}P_F = [1 - \frac{1(c\text{-}1)P_a}{T\text{-}1}][1 - \frac{2(c\text{-}1)P_a}{T\text{-}1}] \ . \ . \ . \ [1 - \frac{n(c\text{-}1)P_a}{T\text{-}n}]$$

where c is the number of bits in a SECDED word (including syndrome bits), P_a is the probability that the two failing devices will have matching failing areas, T is the total number of storage devices in the memory system, and n is the number of devices that fail during a maintenance interval. (Note that if devices always fail in their entirety, $P_a = 1$; this is worst case.)

Parameters c, P_a, and T are generally known for a given system, based on its design. Using various values of n, the equation can be solved until a P_F is obtained which is below an acceptable risk of a fatal failure occurring during a maintenance interval. With n established, the maintenance interval, M, can be determined from:

$$M = \frac{n}{dT}$$

where d is the device failure rate. For risk of less than 0.1 (10 percent), P_F can be translated into MTBF (in years) of fatal failures by:

$$MTBF = \frac{M}{8760\ P_F}$$

This procedure can now be applied to the FMP Main Memory and Backing Store. For this analysis, an acceptable risk of fatal failure, P_F, during a maintenance interval, M, is assumed to be 0.01 (1 percent). The other parameter which must be assumed for lack of established data is P_a, the probability that failing devices will have matching failing areas. For this reason a worst-case value of 1 is used, assuming the entire device fails. With all other parameters known for a given design:

Parameter	Memory	Backing Store
P_F	0.01	0.01
P_a	1	1
c	39	523
d	$2x10^{-7}$	$2x10^{-7}$
T	159,744	267,776
Results:		
n	8	3
M	10 days	2 days
MTBF	2.9 yrs	0.5 yrs

These MTBF figures fall short of those stated earlier, most notably for the Backing Store. The improvement factor resulting from this analysis of SECDED can be determined using the raw failure rates (or MTBF) from Table 3-3 for Main Memory and Backing Store and the MTBF figures determined above with SECDED.

The improvement factor determined in this manner is somewhat in excess of 500 for Main Memory and 300 for Backing Store. It should be noted, however, that a worst-case value of 1 was assumed for P_a.

Referring back to Table 3-3 and substituting the above values of MTBF for Main Memory and Backing Store, the overall MTBF for the FMP becomes 207 as opposed to 214 in Table 3-3.

As stated above, $P_a = 1$ was used since a value is not yet established by statistical data. If this turns out to be smaller (and it is reasonable to expect this to happen) the results of this analysis are improved considerably. For example, $P_a = 0.3333$ for Main Memory produces the factor of 1000 improvement with SECDED over raw failures.

Backing Store, on the other hand, would require a P_a of 0.12 to obtain a 1,000 times improvement by SECDED. However, one more parameter deserves consideration; this is c, or the number of bits in a SECDED word. Design considerations thus far have established the Backing Store as 512 data bits plus 11 SECDED bits for c = 523. If a data size of 128 is used (c = 128 + 9 = 137), a P_a of about 0.45 would produce the factor of 1000 improvement. This may become a powerful argument for a 137-bit SECDED word size for Backing Store as design progresses.

The above consideration of changes in P_a and c for Main Memory and Backing Store also produce improved values of n, accumulated failures, and M, maintenance interval. The changes and the results are summarized below.

Parameter	Main Memory	Backing Store
P_F	0.01	0.01
P_a	0.3333	0.45
c	39	137
d	$2x10^{-7}$	$2x10^{-7}$
T	159,744	280,576
Results:		
n	15	9
M	19 days	6 days
MTBF	5.6 yrs	1.7 yrs

Section 4

SOFTWARE DESCRIPTION

Section 4

SOFTWARE DESCRIPTION

THE PROGRAMMING LANGUAGE

In the first phase report, the subject of programming languages for the FMP was discussed. The conclusion was that FORTRAN, despite its technical deficiencies, was the most likely language to be readily adopted by the applications programmers. The language, PASCAL, is rapidly becoming the predominant system programming language and is therefore recommended as the language in which to write operating systems and compilers for the FMP complex. The specification of the PASCAL dialect should be accomplished in the third design investigation phase of the NASF project.

The purpose of this particular report is to introduce the FORTRAN language extensions felt to be essential for the programming of the FMP. After review of several alternatives with NASA investigators and Control Data FORTRAN specialists, it was determined that the most probable solution to the language problem was the absolute minimization of new language constructs and syntax. This was deemed necessary because of the time required to implement wholly new language features in an existing compiler, while at the same time trying to create new object code output from that compiler optimized for a new architecture such as the FMP.

Several key decisions were made. They are:

- Management of the Vector Buffer Unit would be reserved for the compiler, as the compiler manages the Register File for the CDC STAR computer or the X, A and B registers for the CDC CYBER family.
- Management of the Backing Store would be left to the programmer, using the statements BUFFER IN and BUFFER OUT as modified in this proposal.
- The STAR FORTRAN vector facility requiring explicit descriptions of the vector lengths in arithmetic statements would be abandoned.
- The programmer must aid the compiler in producing vector operations by describing regions in which the compiler is to perform vectorization.
- Some facilities must be provided to assist the programmer in moving code which presently exists in subroutines into in-line sequence, thereby reducing the overhead attendant to JUMP operations, in order to make vectorization easier.

Compilers have difficulty vectorizing programs containing many subroutine calls since the FORTRAN CALL doesn't restrict the behavior of such subroutines, thus permitting recursion or vector overlap to nullify the possibilities of vectorization of the calling routine.

THE LANGUAGE PROPOSAL

It was originally intended to provide a full FMP language specification in this report. The amount of time and resources available for this phase made such a detailed specification impossible, particularly after many man-hours had been consumed in fruitless pursuit of several bankrupt alternatives.

What follows then, is a proposal for a language which would have to be fleshed out in full 'spec' form in subsequent phases of the NASF study.

Base Language

The basic language for the FMP should be FORTRAN which conforms to the ANSI Standards of 1966, to be replaced by the 1978 ANS Standard for which approval is expected soon. For the FMP first installation a FORTRAN based on the 1966 standard would be acceptable since compilers for this version abound on most standard computer products that might be considered for the Front-End Processors. Conversion of the NASF to the 1978 standard should await the availability of compilers at least as mature as those now extant for commercial computers.

The choices available with Control Data Computers are the CDC CYBER FORTRAN Extended compiler with extensions to permit array dimensions of up to 5 (refer to the CDC FORTRAN Extended Reference Manual, publication number 60497800) or the STAR-100 FORTRAN compiler (refer to the CDC STAR FORTRAN Language Reference Manual, publication number 60386200). Obviously, similar compilers are available on a wider range of equipments, but the Control Data investigations have been directed toward modification of the CDC compilers for this purpose.

The Extensions

CODO Statement

The CODO (Concurrent DO) statement invokes a block of FORTRAN code terminated by an ENDCD statement. The form of the CODO statement is:

CODO i = e_1, e_2 (,e_3)

or:

CODO i = e_1, e_2 (,e_3); j = e_4, e_5 (,e_6)

or:

CODO i = e_1, e_2 (,e_3); j = e_4, e_5 (,e_6); k = e_7, e_8 (,e_9)

where i, j and k are integer variables, e_1 through e_9 are integer expressions, and the terms (,e_3), (,e_6), and (,e_9) are optional parameters.

The meaning of the CODO and ENDCD statements is to define a block of code which must meet the following restrictions:

- No subroutine or function calls are permitted in a CODO block, with the exception of certain built-in functions (implicit functions) such as SQRT. In the case of allowed functions the compiler does not generate a generalized subroutine call, but either includes the object code in-line, or generates a special call to a vector FORTRAN subroutine (if the object time parameter is of type vector).

- No branch statements, IF statements, or ASSIGN statements are permitted.
- No input/output statements are permitted.
- The FORTRAN programmer is assuring the compiler that the variables declared in the CODO block do not conflict in storage, via FORTRAN 'tricks' such as EQUIVALENCE or overlaid data due to passed parameters in subroutines pointing to conflicting arrays.

In exchange for these restrictions the FORTRAN programmer is assisting the compiler in the generation of optimal code for the FMP, using vector operations.

Simple CODO Statements

The expression:

```
CODO I=1,100
A(I)=B(I)+C(I)
ENDCD
```

results in the generation of a string of code which, when executed, will perform a memory to memory vector add of the elements in vectors B and C.

The expression:

```
CODO I=1,100;J=1,100
A(I,J)=B(I,J)+C(I,J)
ENDCD
```

performs the element by element sum of arrays B and C, placing results in A.

Complex CODO Statements

Many statements in CODO blocks must be analyzed carefully by the compiler to produce optimum code for the FMP. In the previous examples, the relationship between the source FORTRAN statement and the resultant object code is clearly seen. In most instances this relationship becomes obscured, and thus more complex forms of object code are produced.

The expression:

```
CODO I=1,100
RR=A(I)+B(I)
ENDCD
```

forms a one hundred element temporary vector called RR, and places therein the sum of the vectors B and C. Temporary vectors produced in this manner in CODO blocks cannot be referenced outside of CODO blocks, although one CODO block may reference a temporary vector produced by another CODO block. Temporary vectors of this type may not be referenced by subscripts (that is, RR(1)) within the CODO block.

The expression:

```
CODO J=1,100
D(I,K)=A(J,I,K)+B(J,I,K)
ENDCD
```

produces a temporary vector of one hundred elements containing the sum of the columnar elements of A and B, and stores the result in a sequentially arranged column in the array D at the effective memory location of D(1,I,K). Note however, that D is only defined as a two-dimensional array. Thus the third columnar dimension is created by the compiler as a temporary region for storage of data. In effect then, the array D contains a two-dimensional matrix, each element of which is a 100-element vector.

As in the previous case, arrays produced in this way may only be referenced within and among CODO blocks. Thus they may not be passed as parameters or referenced in COMMON, EQUIVALENCE or BUFFER IN/BUFFER OUT statements.

This form permits the direct inclusion of existing codes, in their present FORTRAN form, by simply 'blocking' groups of vectorable statements into CODO blocks, without destroying the original form of computation and associated documentation. Note that this example is equivalent to the use of:

```
DIMENSION D(100,100,100),A(100,100,100),B(100,100,100)
CODO J=1,100
D(J,I,K)=A(J,I,K)+B(J,I,K)
ENDCD
```

which would permit the programmer to reference the array D or any of its elements in any legal FORTRAN manner. In Section 2, a sample coding is given of the kernel of the left-hand-side calculations. From Figure 2-2 the following has been extracted to illustrate the use of the principles discussed so far.

```
100=          DO 20 L=2,LMAX-1
110=    C***FILTRX
120=    C
130=          CODO J=1,JMAX;K=2,KMAX
140=          RJ=Q(K,L,6,J)
150=          XK=(X(K+1,L,J)-X(K-1,L,J))*DY2
160=          YK=(Y(K+1,L,J)-Y(K-1,L,J))*DY2
170=          ZK=(Z(K+1,L,J)-Z(K-1,L,J))*DY2
180=          XL=(X(K,L+1,J)-X(K,L-1,J))*DZ2
190=          YL=(Y(K,L+1,J)-Y(K,L-1,J))*DZ2
200=          ZL=(Z(K,L+1,J)-Z(K,L-1,J))*DZ2
210=          D(J,1,2)=HDX*((YK*ZL-ZK*YL)*RJ)
220=          D(J,1,1)=HDX*(-OMEGA*(Z(K,L,J)*(RJ*(YK*ZL-ZK*YL))
230=         1  -Y(K,L,J)*RJ*(XK*YL-YK*XL)))
240=          D(J,1,4)=HDX*((XK*YL-YK*XL)*RJ)
250=          D(J,1,3)=HDX*((ZK*XL-XK*ZL)*RJ)
```

1. The CODO block produces vectors by varying both the J and K indices. The K index is varied first from 2 to KMAX, then the J index is incremented and the K index scanned through its range again. Lines 140 through 200 in the original scalar code formed a series of scalar temporaries. Inclusion in the CODO block for this version causes the scalar temporaries to become vector temporaries. In this case the compiler is capable of generating a GATHER operation of KMAX-2 elements for each J from the arrays X, Y, Z, and Q. The result of these GATHER operations is to produce a series of 'invisible' temporary vectors (known only to the compiler, and unnamed in the FORTRAN source) of length (KMAX-2)* JMAX.

2. The 'invisible' temporaries are then combined arithmetically according to the FORTRAN source code to produce 'visible' temporaries (that is, named by the programmer), such as RJ, XK, YK, ZK, XL, YL, and ZL. As described previously, these scalar-appearing temporaries are actually vectors. Statements 210 through 250 then combine these temporaries to form the array segments for D, which will be a three-dimensional array, each element of which is a vector of length KMAX-2.

3. The truly scalar value OMEGA which is defined in scalar portions of the FORTRAN source input, is used as a scalar in the CODO block. In line 220, it is broadcast as an operand in the multiplication operation involving the array Z.

4. The compiler will check all references to ensure that they are conformal (that is, all dimensions are equal for all operations). A left-hand-side (of the equals sign) operand can have one 'invisible' dimension provided by the compiler, based on the dimensions of the operands on the right-hand side. However all references to that operand must use the same dimensionality. Thus the expression:

```
CODO K=2,KMAX
D(I,J)=A(K,I,J)
ENDCD
CODO L=1,LMAX
D(I,J)=A(L,I,J)
ENDCD
```

   is not allowed, since at compile time, the storage allocation and object code necessary to assign the array D cannot be determined because LMAX might not equal KMAX at execution time.

Boundary Conditions

The CODO statement, in the forms discussed previously, causes a uniform application of the CODO indices to all variables possessing those indices as subscripts (or implied subscripts when the compiler has generated an internal, 'invisible', temporary vector). The addition of a special operator to each indexing statement (.CCNT.--concurrent index) permits varying index values with differing results.

The statements:

```
CODO I=1,100.CCNT.J=1,100
A(I)=B(J)+C(J)
ENDCD
```

are equivalent to:

```
CODO I=1,100
A(I)=B(I)+C(I)
ENDCD
```

since the operator .CCNT. indicates that the index J is incremented concurrently with the index I. This feature can be utilized in:

```
CODO I=1,100.CCNT.J=1,200,2
A(J)=B(I)+C(I)
ENDCD
```

which forms the element by element sum of the arrays B and C and places the results in every other element position of the array A (which obviously must be dimensioned at least by two hundred elements).

The .CCNT. operator requires the definition of certain end cases in the use of several concurrent indices:

1. The number of index steps implied by the values of the index limit and index step need not be identical for all index variables associated by the concurrent operator (.CCNT.). Thus the statement:

   ```
   CODO I=1,100.CCNT.J=2,99
   ```

   permits both the index I and the index J to have different starting and ending values.

2. The use of .CCNT. implies that the two indices are synchronous. In the example in 1. above, where I and J have different initial values, when I=1, J would be equal to 2, and when I=98, J would be equal to 99.

3. When the termination values are unequal, each index with the lesser termination value upon reaching termination takes on values of NULL for each index step of the remaining index. Thus in the example given, when I=98, J=99; but when I=99, J can no longer be incremented past its termination value of 99 and thus becomes NULL. The result of this action can be to create a 'bit bucket' into which operands are discarded rather than stored.

   The expression:

   ```
   CODO I=1,100.CCNT.J=2,99
   A(J)=B(I)
   ENDCD
   ```

   would store 98 values from array B, beginning at element 1, into the array A, beginning at element 2. A more meaningful example would be:

   ```
   CODO K=1,100;I=1,100.CCNT.J=2,99
   A(J,K)=B(I,K)+C(J,K)
   ENDCD
   ```

   which can generate object code for an ADD vector operation of length 100*100, but where two elements from each 100 in the J direction are not stored. This particular operation would result in the creation of a Control Vector by the compiler, with two zero bits in

each hundred. Another case is:

```
CODO K+1,100.CCNT.J=1,200,2
A(K)=B(J)
ENDCD
```

which would in effect compress out of the vector B every other element, placing the result in the vector A. In this example the compiler would form an Order Vector which would be applied in the Map Unit to perform a vector COMPRESS on the array B. The Order Vector would contain an alternating ones and zeros pattern, and could be generated at compile time, or at object time, since it is a fixed, non-data-dependent pattern.

As has been stated previously, the object of creating the CODO construct is to permit the insertion of a minimum number of lines of new code into existing programs to assist the compiler in the vectorization process. The examples given in this report are only a recommended staring point for subsequent programmability studies of the FMP.

LEVEL 2 Statement

The form of the LEVEL 2 statement is:

```
LEVEL 2 array name 1, array name 2, . . . ., array name n
```

All arrays specified in a LEVEL 2 statement are assigned to Backing Store. The only references permitted to these arrays are via the BUFFER IN/BUFFER OUT statements.

This feature permits the assignment of large data blocks to Backing Store by symbolic name. Array name 1 through array name n may appear in other FORTRAN declaratives (such as DIMENSION, INTEGER, etc.) or may be defined solely in the LEVEL 2 statement as:

```
LEVEL 2 A(100,100,100),B(100,100)
```

The compiler attempts to assign arrays greater than or equal to 32,768 words in length to an integral 32,768 word block starting address.

Additional Extensions

The CODO and LEVEL 2 statements discussed above and BUFFER IN, BUFFER OUT, UNIT which follow represent what is considered to be the minimum essential extensions to the FORTRAN language. They assume a reasonable extension of the compiling and optimizing capabilities of known FORTRAN compilers, such as the STAR FORTRAN compiler. The objective of minimizing extensions is, of course, to reduce development and testing time for the compiler, and retraining time for programmers. However, the programmers must be trained to understand the relationship of the CODO statement to object code efficiency, a process which will necessarily be somewhat long and agonizing.

Other suggestions are offered to assist in the programming of the FMP in FORTRAN, but are not as essential as those few previously mentioned. The other suggestions are:

- MACRO facility – Examination of the segment of the three-dimensional code in Section 2, Figure 2-2, shows that the entire segment has been converted to in-line code. Thus the

subroutines XXM and BTRI have been included directly and the consequent subroutine calls eliminated. This serves two purposes. First, in the case of XXM, unnecessary code and conditional branch statements are eliminated, since when used in the AMATRX segment, the conditions of the indices are known beforehand, and since the XXM routine is never processing data on the boundaries in this instance, the overall code can be replaced. The second purpose served is that by eliminating subroutine calls, the code can be blocked into CODO segments more efficiently. As in scalar code optimization, the compiler can better optimize code if it has a larger block of 'uninterrupted' code to deal with.

The scheduling of the GATHER operations, implied by the statements in Figure 2-2, lines 140 through 210, and lines 290 through 360 can be optimized easier over the whole block of CODO, ending in line 660, than could be optimized of the CODO ended at line 210 (as if a subroutine call had been made to XXM). In this case, the GATHER operations for Q(K,L,Y,J) could be initiated, immediately after the GATHER operations have been completed for X, Y, and Z, and can be accomplished concurrently with the calculation of the metric cross products in lines 140 through 250.

The in-line coding of large segments of computation places a burden on the programmer in both keypunching (inputting source statements) and maintaining congruence between each of the in-line expansions of what would otherwise be a subroutine. Specifically, if the subroutine XXM is kept in subroutine format, any changes in the calculations in XXM need be made only once within the subroutine. If the subroutine is included in-line at the point where it is called in the main program, then each version would have to be changed. A means for reducing this problem is the inclusion in the language support system of a powerful MACRO processor, which can recognize particular constructs, evaluate parameters, and generate the necessary lines of FORTRAN source code. The most desirable MACRO processor would be one which is imbedded in the language processor itself, since items such as the variable attributes and lengths are readily available. However, no such MACRO facility is prescribed as a standard for FORTRAN, and no compiler presently possesses such capability. To minimize development cost then, a MACRO preprocessor, based on already operational systems, should be provided. Two very powerful MACRO systems are available on commercial equipments; they are called ML-I* (ref. 1) and STAGE-2** (ref. 2). There are a host of other candidates available on non-CDC equipment.

If code development is to continue in a reasonably dynamic way through the lifetime of the NASF, then the value of such a MACRO facility is extremely high. However if the system code becomes rather static, then the manual labor involved in creation and maintenance of the code may not justify the inclusion, documentation, training and maintenance of a sophisticated MACRO facility. At this juncture, Control Data would highly recommend investigating and including such a MACRO facility (preferably one already in existence) for the FMP and the front-end processors, operating solely on the front-end processors.

Intrinsic Functions – Certain attributes of codes that may find their way onto the FMP require the handling of data-dependent vectorization. The FMP hardware provides the facility for manipulation of array data based on some selection criteria, and to some extent the CODO statements can cause the compiler to generate operations using these facilities. In other cases however, the programmer must be aware of the vector nature of a given conditionally-executed operation and should have direct access to that facility. This can be accomplished by defining a set of built-in, intrinsic functions which might be:

VCMPRSS(b,A) — Compress array A by bit vector b

VMRG(A,b,C) — Merge elements of array A and C under control of bit vector b

VMASK(A,b,C) — Produce a vector consisting of elements from A corresponding to one bits in b, and elements of C corresponding to zero elements in b

VSEARCH(A.EQ.B) — Compare elements of A and B, return a scalar index variable containing the position in the arrays at which the comparison is met. Any legal FORTRAN relational operator (.NE.,.GT., .LT.,.LE.,GE.,.NOT.) are permitted in the relational expression.

VCOMPARE — Compare elements of A and B forming vector of integers containing the index position in the array, where the relation is met.

VSELECT — Compare elements of A and B forming a bit vector, with one bits in each position wherein the relation is met.

The BIT attribute permitted by STAR FORTRAN, and the logical operators .AND.,.OR.,.XOR. would be used on bit strings to provide manipulation of the Order and Control Vectors, explicitly.

- Machine Language – Experience with programming portable modules in STAR FORTRAN has shown that use of the 'escape valve', introducing in-line machine code via the scheme called Q8mnemonics, has resulted in undecipherable code, which is difficult to optimize by a compiler since the compiler cannot control the resources of the various functional units as closely as when no explicit machine code is allowed. Unless the compiler development cannot utilize an existing compiler system as a base, and unless current optimization techniques prove to be useless for the FMP (a very unlikely event), the use of machine code escape mechanism should be prohibited and not implemented in the compiler. If events require the development of a brand new compiler with massive language changes, then it may be necessary to introduce this form to provide early access to the FMP facilities while the compiler is maturing.

Buffered Input and Output

Explicit input and output can be initiated by the FORTRAN programmer for data transfers between the Backing Store and the network input/output system, and between the Backing Store and Main Memory. The mechanism for controlling this input/output activity is the use of the FORTRAN BUFFER IN and BUFFER OUT statements.

The length of the buffer area in which the data is contained in memory should be an even number of multiple of blocks for all files. Ordering the data in this manner provides the most economical use of storage.

Any unit referenced in a BUFFER statement cannot be referenced in any other input or output statement; however, such units can be referenced in the unit positioning statements BACKSPACE, REWIND, and ENDFILE. Once buffered input/output is established for a logical unit in a FORTRAN program, all input and output for that unit must be buffered.

The ENCODE and DECODE statements are most frequently used to process the data read into a buffer, or to gather and place data in a buffer for transmission to external files. Status of the peripheral device involved should be checked by the UNIT function before the buffer operation is begun.

BUFFER IN Statement

The execution of the BUFFER IN statement transfers data from the unit specified, in the mode given, to Level 2 storage locations first to last.

Form

BUFFER IN(unit,mode)(first,last)

unit An integer constant or variable that represents the logical unit number.

mode — An integer constant or variable that specifies the recording mode of the data being read. The permitted values are:

0	7-track BCD
1	7-track or 9-track binary
2	CDC 64 character ASCII subset
4	Mass storage (disk)
5	Archive
6	Front-End Processor (FEP)

first — A Level 2 array reference defining the first location in the buffer into which data is to be transmitted. The transmission continues from that point to the location specified by parameter last. The array name used can be type character, integer, real, double precision, or complex.

last — A Level 2 array reference defining the location in the buffer into which the last data item is to be transmitted. The location designated by the parameter first must be less than or equal to the location designated by parameter last; both must refer to the same array. The array name used can be type character, integer, real, double precision, or complex.

The BUFFER IN statement initiates data transmission from the logical unit to the buffer. Before data in the buffer can be used, the status of the data transmission must be checked using the UNIT function.

Example

```
BUFFER IN (5,2) (X(1),X(10))
```

UNIT Function

The UNIT function is suitable for evaluation in an arithmetic IF statement that causes branching to appropriate statements as directed by the value returned. Failure to perform a unit status check renders unpredictable the input that is transferred to the buffer by the preceding BUFFER IN statement.

Form

```
UNIT (u)
```

u — An integer constant or variable that represents the logical unit number.

The function returns one of the following real values:

-1.0	Unit ready
0.0	Unit ready; end of file encountered
1.0	Unit ready; parity error encountered

BUFFER OUT Statement

The execution of the BUFFER OUT statement transfers data to the unit specified, in the mode given, from Level 2 storage locations first to last.

Form

BUFFER OUT (unit, mode)(first.last)

unit — An integer constant or variable that specifies the logical unit number.

mode — An integer constant or variable that specifies the mode in which the data record is to be written:

0	7-track BCD
1	7-track or 9-track binary
2	CDC 64 character ASCII subset
4	Mass storage (disk)
5	Archive
6	Front-End Processor (FEP)

first — A Level 2 array reference defining the first location in the buffer from which data is to be transmitted. The array name used can be type character, real, integer, double precision, or complex.

last — A Level 2 array reference defining the location in the buffer from which the last data item is to be transmitted. One logical record is written for each BUFFER OUT statement.

Example

```
BUFFER OUT (6,3) (X(1),X(10))
```

Extensions for Backing Store/Main Memory Transfers

For transfers between Backing Store and Main Memory, the BUFFER IN and BUFFER OUT statements are extended as follows:

BUFFER IN (Level 2 array reference) (first, last)

BUFFER OUT (Level 2 array reference) (first, last)

Level 2 array reference — An array name or array reference (subscripted array name) to an array declared to be in Level 2 memory (Backing Store).

first — A Level 1 array reference defining the first location in the buffer into which data is transmitted. The transmission continues from that point to the location specified by parameter 'last'. All transmissions are in integral number of 32,768-word blocks.

last — A Level 1 array reference defining the location in the buffer into which the last data item is to be transmitted. The location designated by the parameter 'first' must be less than or equal to the location designated by the parameter 'last'; both must refer to the same array.

The UNIT function is also extended as follows:

UNIT (level 2 array reference)

which returns the following real values:

-1.0 transmission to or from referenced array is complete

0.0 transmission to or from referenced array not complete

1.0 transmission to or from referenced array cannot be completed (Backing Store locked out, or data not present)

The compiler attempts to assign arrays in Backing Store and in Main Memory to large block (LB) boundaries (32,768 64-bit word segments). If the BUFFER IN, BUFFER OUT statements reference integral blocks, the compiler generates direct SWAP instructions. If the block being transferred does not align to a block boundary, or if less than an integral block is transferred, the compiler generates a SWAP to an intermediate Main Memory block, then generates 'in-line' code to move the sub-block (or partial block) to its appropriate Main Memory location. The same operation is performed in reverse for BUFFER OUT statements, with sub-blocks being moved to Main Memory intermediate blocks and then a Backing Store SWAP initiated.

Because of the compiler's attempts at assignment of arrays for optimum transfers, the programmer should be cautioned that arrays are not necessarily stored sequentially to one another by the compiler. Thus the statement:

```
DIMENSION A(100,100),B(100,100)
```

does not imply that B(1,1) immediately follows A(100,100) in actual memory.

The example:

```
DIMENSION Q(100,100),R(100,100)
LEVEL 2 QB(100,100,100),RB(100,100,100)
BUFFER IN (QB(1,1,1),Q(1,1),Q(100,100))
```

would move 10,000 elements beginning at the first block of QB to the array Q in Main Memory. To determine if the final transfer is completed the programmer may use the statement:

```
IF UNIT (QB(99,99,100)) 110,120,130
```

to branch to the appropriate statement depending on the condition of the transfers underway. Note that the FMP hardware maintains status information on SWAPS in 32,768-word blocks, thus for a BUFFER IN operation on block boundaries for an array X(100000) the UNIT statement:

```
IF UNIT (X(1))1,2,3
```

is equivalent in function to the statement:

```
IF UNIT (X(32768))1,2,3
```

since the hardware flag tested by the object code is identical in both cases.

The Specification

The philosophy governing the introduction of the aforementioned language extensions has been to minimize change in language or compiler. The next phase of this project must produce a full scale programming language specification which can be used to procure and implement an applications programming language for the FMP.

The first item that must be resolved in that subsequent phase is a choice of the FORTRAN base language, FORTRAN EXTENDED, FORTRAN 66 or FORTRAN 78. This decision will have to result from meetings between staff members from NASA and the RADL investigators, with schedule risk being a preeminent consideration.

The compiler specification therefore must await this same decision. The only specification possible at this time being the hand-compiled examples given in Section 2 and the description of probable compiler actions given in the preceding discussions of CODO and BUFFER IN/BUFFER OUT.

OPERATING SYSTEM FUNCTIONAL REQUIREMENTS – FLOW MODEL PROCESSOR SYSTEM

SYSTEM PHILOSOPHY

Three factors drive the architecture of the FMP operating system (FMPOS)

- Minimization of new software development,
- Reduction of overhead within the FMP CPU,
- Balance of system resources.

The development schedule for the FMP system precludes a massive development of software to support all of the functions commonly associated with general-purpose computing facilities. To achieve the level of total system stability, reliability, and availability implies that a substantially constrained set of functions be allocated to the FMP CPU operating system, and existing software be exploited in all attached processors to the maximum extent possible.

The main purpose of the FMP is to perform massive amounts of computation on highly vectorized mathematical codes. The objective of the total system installation, therefore, is to maximize the amount of time that the FMP is operating at its peak speeds. First and foremost, the language processor and related documentation must ensure that the actual computations are matched to the hardware architecture. Secondarily, as little FMP resource as possible should be tied up in the management of internal FMP functions such as memory allocation. The functional constraints on the FMP serve to reduce both the space (in main memory) and the time (usually using inefficient scalar code) required by the CPU-based portion of the operating system.

System balance is an important and obvious consideration as the power of the FMP could be quickly dissipated by bottlenecks in input or output or in the scheduling of system resources (such as disk space) by other supporting processors.

The preceeding imperatives point to the need for a system philosophy around which a system design can be formulated and upon which the total system implementation can be based. The approach taken for the FMP is an extension of the "distributed system philosophy" originally evolved for the Control Data STAR computer systems.

Distribution of Functions

The allocation of system functions should be governed by some basic guiding principles which can be used for both hardware and software implementations. A suggested set of such guidelines are offered here:

1. System resources consist of storage and processing facilities. Storage can be central memory, backing store memory, disk drives or magnetic tape devices. Processing resources could be the FMP CPU, miniprocessors handling system control, or other computers providing support functions.

2. Management of resources is performed by computers which can be programmed to make intelligent decisions, and to perform whatever control and management functions may be assigned to them.

3. The management of resources should be placed as close as possible (electronically, physically, and logically) to the resource being managed. Thus a disk management function should be allocated to a processor which may actually reside within the disk unit or in the disk controller which is normally intimately connected to the physical storage units.

4. All such resource management functions should be moved outward from the central computer toward the particular resource.

5. Form follows function; the hardware should be built to fit all of the functions which have been moved outwards to the resource, rather than to fit as many functions as possible into an existing unit. In place of the word "'build" here one could say "sized", since many standard computing elements can be used in distributed fashion, but the tendency to 'go the cheapest route' usually results in acquiring a processor too small, into which a partial list of management functions are then force-fit.

6. An intelligent processor should manage only its own resources and should be ignorant of (and thus unable to manage) other processors' attached resources.

7. A processor should maintain a list of functions which it is capable of performing. Any functional requests not in this list should be passed on (if the processor is a communications node) or not acknowledged (if the processor is part of a network). Thus no processor rejects requests unless they are patently illegal, and therefore no processor need know what functions other processors are capable of performing.

8. All communications between processing elements must be through a single, highly structured message system, with rigorous attention paid to message formats and protocols.

9. All the preceeding principles must be tempered with common sense and technological and economic realities.

The result of the application of this set of groundrules is manifested in visible system features such as processors whose sole responsibility is the management of files for all other processors in a given complex. This is the ultimate extension of the process whereby first a processor and software are created to manage the motion of a disk arm and the reading and writing of data bits on the magnetic media; thence the ability is added to that processor (which is nearly imbedded within the disk unit) to handle error detection, retry and some recovery of the data recorded on the disk; then further diagnostic ability, management of the disk space, and finally the management of the files on that particular disk are added.

A list of functions to be distributed to a multiplicity of processors then would include:

1. File management – Control of access, security, backup and error handling, space allocation.

2. Communications handling – Management of all remote access trunks, logon validation, recovery, scheduling of resources activated by the remote devices.

3. Trunk management – Control of the network that interconnects the collection of resources and processors.

4. Special processor control – Independent management of special resources such as the FMP graphics processors and archival storage coordinators.

Hardware Interconnection

The most flexible system organization would permit the interchange of data and control information between any set of processors and resources in the system. As the number and variety of processors grows, the practical methods of interconnection become taxed by physical limitations such as volume and

lengths of cables. The FMP system is based on a network trunk technique (reference section 3.5 of Functional Specification).

In this scheme all intelligent processors are connected together by one or more bit-serial trunks on which data can be transmitted, or control information interchanged. Each connection is via a programmable device controller (PDC) which is itself an intelligent processor. Management of the trunk (which is itself a resource) is distributed among all the PDCs on the trunk which deal with contention and scheduling of transmissions.

Each PDC is capable of providing for attachment to four different trunks, however, in the interest of system availability at least two PDCs will be used at each interconnection. Each of the PDCs will have access to at least two different trunks.

All system resources, disks, tapes, graphics, archival, communications, special FMP CPU and front-end processors, will be attached uniformly throughout the network, thus permitting the linking of any system component with any other.

In general, data transfers are direct from resource to resource without intervention (or store-and-forward) by other processors. Thus a high speed disk unit would transmit data directly to the FMP CPU, without the data being passed through any other processor (such as the front-end units). The major front-end processors charged with the file management responsibility in this case would validate the access to the particular disks, setup the software linkage (so that the FMP knows where the data is physically located), then step out of the way (logically) while the high-speed data transfers take place.

While the system is on-line, any device or processor can be logically, or even physically, removed from the network without disrupting operation, as long as that resource is not required for a particular computation during the time of removal.

Software Interconnection

The total interconnectability offered by the hardware aspects of the network trunk can be constrained by the software system to appear as a variety of traditional interconnection schemes (such as a STAR organized network). The choice of constraining the system interconnections must be based on:

1. Desire to eliminate a multiplicity of interface modules to be written; despite a generalized message structure, the act of linking a PDP-10 graphics computer to the FMP would require some interface logic different from that needed when linking to a disk or a front-end processor.

2. Format conversion – To produce the most cost-effective system it is desirable to adapt any number of existing, proven hardware devices to the purposes of the FMP. Raw data, not even internal arithmetic formats, are rarely identical, thus there exists a need for software to provide conversions. These impose overheads in space and time and also require programming and checkout resources, which may be in limited supply. The need to reduce the number and variety of this type module is great.

3. The need to restrict access to certain resources – For security or system efficiency reasons it may be desirable to limit certain interchanges. Thus the graphics processor has no need to speak directly to the FMP for any reason, and vice versa, despite the fact that they may be attached to a common trunk for purposes of attachment to the high-speed disk units.

The software must provide, via modifiable tables, a means for defining the apparent interconnection of all devices on the trunk. The network trunk provides an eight-bit address for each unit attached. At the basic hardware level, any device can have its address established by manually changing the setting of a series of keylocked switches. The PDC can have loaded into itself at system startup time a series of software addresses (or address-like structures). Finally, each attached processor will have its own higher-level addressee structure.

Messages, Structure and Discipline

To keep any system of cooperating but asynchronous processors from degenerating instantly into a state of electronic chaos, a rigid set of protocols must be defined and adhered to rigorously. The only (emphasize the word only) means of intercommunication is through a predefined set of system messages. There cannot be any sneak paths or extra wires used for that one special case. The rule is simply that if a function cannot be handled efficiently within the message system, then either the message system must be revised or the function abandoned. There can be no equivocation on this rule, lest the system totally collapse from special-casing.

When dealing with the concept of messages, several levels of system consciousness or message envelopes must be defined:

1. Trunk protocol – Each PDC, when communicating with another PDC on the trunk, builds a basic trunk protocol envelope around the data being transmitted, and decomposes the envelope from data received (Reference section 3.5 of the Functional Specification).

2. Processor Protocol – Each processor involved in interchanging information on the trunk places another level of envelope around the data being exchanged. Whereas the trunk messages involve hardware-oriented items such as hardware address codes and cyclic error checking, this second level is defined by the software portion of the operating system. The format and contents are addressed to the methods wherein the messages are stored, queued, routed, and decoded within and by each processor. Suggested formats and a list of message types to be implemented for the FMP are presented below.

3. The highest level messages are those exchanged between job-level programs executing in the major computer processors (FMP, front-end, graphics and archival store manager). These messages are primarily for control purposes rather than for the exchange of quantities of data. An example would be the chit-chat between the interactive graphics processor producing the displays, and a mesh generation and stretching program residing in the front-end processor during a session wherein the aerodynamacist is interactively modifying a mesh structure. Specification of this set of messages will have to await further phases of the FMP development project.

Message Formats and Types

User programs may issue messages which result in the performance of system functions. To issue a message, the user presets a 2- or more word block according to the Alpha and Beta conventions described below, and performs an Exit Force instruction (09) that transfers control to the operating system monitor mode.

Immediately following the exit force instruction in the instruction stream is either a 32-bit indirect or a 64-bit direct message pointer. Hexadecimal format of an indirect message pointer is:

00EE00rr

rr Register containing the address of the message.

The hexadecimal format of a direct message pointer is:

00FFaaaaaaaaaaa

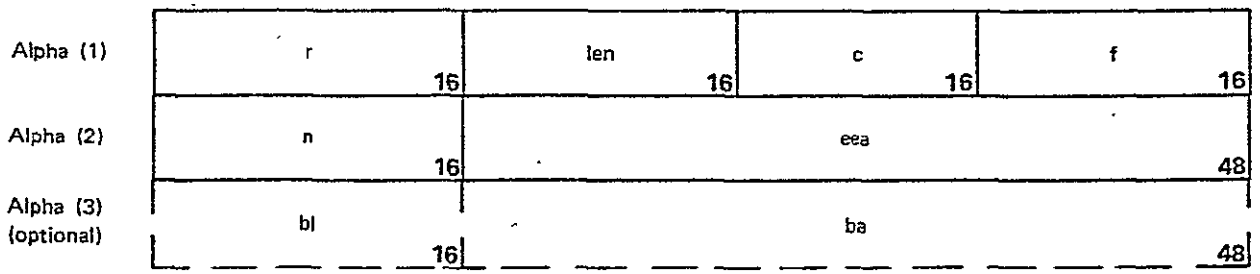
a's Address of the first full word of the message.

The message has a two-part standard format. The Alpha (first) portion specifies the function to be performed, length of parameter list, and where to proceed for error processing. The Alpha portion has the same general format for all messages.

The Beta (second) portion contains the parameters. The format of the Beta portion depends on the function, as described later for each function code. Alpha and Beta words must start on full-word boundaries and must exist in space which has read/write or write temporary access.

When a message is processed without error, the operating system returns control to the next half or full word immediately following the message pointer. Thus, calls can be chained by placing one message pointer directly behind another.

Alpha format:



r Hexadecimal response code returned by the operating system when message has been processed. If no error occurred, the response code is zero (exceptions: f=0013, f=0016 and f=0017). The significance of a non-zero response code varies as described for each function code.

len If len = FFFF, Alpha (3) contains the length and bit address of the Beta portion. Otherwise, Beta is assumed to begin at Alpha (3) and len is the length of the Beta portion.

c This field varies with the message; usually, it specifies function options or controls.

f Specifies function to be performed (hexadecimal message code).

n May specify option or control, may contain a parameter for the message, or may be a parameter returned during message processing.

eea Bit address that receives control if error occurs. This address must lie within the program issuing the message. If eea = 0, the error is considered fatal to the further execution of this user process.

bl If the Beta and Alpha portions are not contiguous (len = FFFF), this parameter indicates Beta length in full words.

ba If Beta and Alpha portions are not contiguous (len = FFFF), this parameter indicates address of Beta portion's first full word.

The terms controller and controllee have specific meaning relative to FMPOS. For example, a batch processor also controls actions of a user program; the former becomes the controller and the latter, controllee. This relationship between programs can exist in other ways as well: one program can initialize and/or direct the actions of another.

Since FMPOS is a file-oriented system, file management is an important aspect of the operating system. Although FMPOS takes a little direct responsibility for action on a given file, a set of user messages allows a fair degree of latitude in directing FMPOS processing for a given file. Standard messages also transmit information between programs operating in controller-controllee mode. The messages are calls to the system; they are shown in the following table by the alphabetical name of the message.

MESSAGE FUNCTION CODES

Message	Function Code†
CHANGE FILE NAME OR ACCOUNT NUMBER	0B
CLOSE FILE	05
CREATE FILE	01
DESTROY FILE	02
EXECUTE OPERATOR COMMAND††	21
EXECUTE PROGRAM FOR USER NUMBER††	22
EXPLICIT I/O	50
GET MESSAGE FROM CONTROLLEE	17
GET MESSAGE FROM CONTROLLER	16

Message	Function Code†
GET PACKLABEL AND PFI	11
GIVE FILE	08
GIVE TAPE ACCESS TO CONTROLLEE	0C
GIVE UP CPU UNTIL I/O COMPLETES	52
INITIALIZE CONTROLLEE CHAIN	1D
INITIALIZE OR DISCONNECT CONTROLLEE	1B
KEEP	28
LIST CONTROLLEE CHAIN	13
LIST FILE INDEX OR SYSTEM TABLE	09
MAP	04
MESSAGE CONTROL	18
MISCELLANEOUS	24
OPEN FILE	03
POOL FILE MANAGER	26
PROGRAM INTERRUPT	1C
RECALL	25
REDUCE FILE LENGTH	0A
REMOVE CONTROLLEE FROM MAIN MEMORY	19
RETURN FROM INTERRUPT	51
ROUTE AND FILE DISPOSITION	0D
SEND MESSAGE TO CONTROLLEE	15
SEND MESSAGE TO CONTROLLER	14
SEND MESSAGE TO OPERATOR	1A
SET ALL PERM FLAG††	2A
TERMINATE	06
UPDATE USER DIRECTORY††	23
USER/ACCOUNTING COMMUNICATION	0E

† Reserved for future use:

07 ADVISE
20 ABNORMAL TERMINATION INTERRUPT
27 LINK SYSTEM CALL
29 SEND MESSAGE TO DAYFILE
1E, 1F, E0-FF Reserved for installation use

†† Available to a privileged task only

THE FMP MONITOR

The FMP CPU hardware is being designed with a particular mode of system operation in mind. The distribution of functions to the other processors attached to the network trunk frees the FMP from many of the conventional operating system chores. Thus the hardware design permits a total of 65,536 64-bit words to be utilized by the monitor. Since there is no direct input or output to the main memory and since the user has dominion over all the remaining memory, there will be no (repeat – no!) operating system overlays. The absolute maximum is 65,536 words. Certain hardware instructions have been provided to assist the monitor in its resource management functions; other instructions have been consciously omitted to inhibit the desire to add one more feature to the system.

Allocation of FMP Resources

In keeping with the distributed system philosophy, the FMP monitor need only manage the storage available to it (Main Memory and Backing Store – the register file and vector buffers are the responsibility of the compiler system), and the computing resources (or which job is to be loaded and executed next).

Backing Store

The 256-million-word Backing Store is managed in units of 32,768-word blocks. All data transmissions are accomplished in that same size block, however, the input/output channel PCDs may decompose the blocks to smaller increments for transmission on the trunk.

In the initial configuration, there are 8192 blocks of 32,768 words that can be managed. Any program executing in monitor mode can address any block in the Backing Store. Programs operating in job mode can reference a contiguous band of Backing Store as established by the monitor. The monitor then must be able to provide the following facilities:

- Allocation of Backing Storage for the entire program and data base for a job being loaded from the network trunk. This allocation is based on the queued list of jobs submitted by the front-end processor for execution. Included in the queued information is the space requirements for the job execution.
- Setup of the base address register and field length register for the job in execution to enable that job to reference the Backing Store.
- Deallocation of storage as the completed (or aborted) job's data is rolled out onto the network trunk.
- Allocation of small blocks of Backing Store for on-line diagnostic storage, I/O lists (see I/O Handling, below), and system statistics buffers.
- Freezing of blocks in the Backing Storage when explicit input/output requests involve them, freeing the same blocks on completion of I/O actions.
- Developing accounting information for billings based on storage useage over time.
- On-line exercise (periodically) of the Backing Store map table (which interlocks the use of Backing Store) and any other facilities attached to the Backing Store, to verify that everything is still working.
- When running a series of small jobs that require small amounts of Backing Store, maintaining a table of space allocation for all such jobs.

Main Memory

Only one job is intended to reside in the Main Memory at a time, thus the monitor need provide no special facilities for memory allocation.

The management of Main Memory as a resource coincides exactly in form and content with the allocation of the computing resource, which follows.

Functional Units

The front-end processors are responsible for the organization, staging, and scheduling of jobs to be submitted to the FMP. Once a job is fully staged by the FEP (front-end processor) an FEP to FMP monitor message is transmitted on the trunk. This is a type 2 message, which gives the following data:

Job I.D. (generated by the FEP)
Backing Store and Main Memory requirements
I/O list for the staged job (see I/O Handling, below)
Time limit (if job exceeds the limit – abort –)
Relative priority
I/O list for files to be accessed with explicit I/O

The monitor allocates Backing Store for the I/O lists and queues the remainder of the message in a sixteen-job queue (maximum allowed is 256 jobs, but that appears to be excessive). When the job in progress completes, the monitor initiates the roll-out of that job and examines its job queues (including diagnostics that might be invoked on periodic schedules). The job with the highest priority that will fit the available memory (in the event that the FMP is operating in degraded memory mode) will be rolled in:

- Continguous block space must first be allocated to the job coming in. This may involve collection of disparate groups of blocks that have become diffused in the Backing Store during explicit I/O or small job executions.
- The file/files containing the job data to fill the block space are physically defined by the I/O lists. The lists for the incoming job are pointed to by the queued job request held in monitor's personal area in the 65,536 main memory block reserved for monitor. Monitor transmits this pointer to the PDCs on the I/O channel which then access the lists and perform the data loading functions. Prior to initiating the I/O action, monitor sets all affected blocks 'busy' in the Backing Store map table.
- As blocks are loaded by the PDC, it returns a response to the monitor which verifies that that portion of the operation was properly completed, and clears the map table busy flags. Upon completion of the last block, the monitor sets a ready flag in the job queue.
- When a job is in the ready state, monitor transmits a message to the FEPs indicating which job has been staged to the Backing Store, along with a time stamp. This permits the FEPs to maintain an audit trail of the progress of a given job.
- When the job in the CPU completes execution, its entire Main Memory contents are swapped to its Backing Store blocks. The job in ready state is then swapped into Main Memory, the Backing Store RA+FL (reference base address and field length) register set, the monitor interval timer set with the job time limit and an exchange operation initiated which puts the job into execution.

- As the exchange operation is initiated the monitor sends another time-stamped system message to the FEPs to alert them that the job is now executing.
- When the job completes and has been swapped to the Backing Store, monitor alerts the PDC, gives it the I/O list pointers, and thus initiates the roll-out operation after setting all the respective Backing Store blocks busy.
- On job completion, an end-of-job is transmitted to the FEPs.
- When the roll-out is completed, monitor sets all relevant blocks not busy, updates its own chart of available memory, sends a final roll-out time-stamped message to the FEPs, and examines its jobs waiting queue for the next job to be preloaded into the Backing Store.

PDC Communication with the Monitor

Upon completion of requested actions by the PDC, it loads 64 bits of software-defined status information into the channel status word. The monitor periodically pools each channel with an 02 instruction (Transmit (R) to Channel (S) and Channel (S) to (T)).

The software definition of the included bits in the status word provides for valid and error terminations of the Input/Output request, as well as other information. For example, the monitor might request that it be informed of the completion of each block transfer accomplished in a multi-block input/output exchange. The PDC would then respond by updating a status word for each transmission.

Since a block transfer at 200 megabits is completed in about 14 milliseconds, the polling rate of once each 100 microseconds would allow monitor a fairly refined scan of the progress of input/output it has requested.

User/Monitor Communications

All communication between the user job in execution and the FMP monitor is through a message structure identical to the format and style given in the section on Messages, Structure, and Discipline above. Two methods are used by the user job for pointing to the messages:

1. Direct – The user job executes an 09 (Exit Force) instruction. The 64-bit quantity immediately following the Exit Force instruction contains a hexadecimal FF in the leftmost eight bits (bits 0 through 7), indicating that a memory address is contained in bits 35-63 of that word. Monitor will fetch this word and use the address to acquire the message. All messages must be stored in Main Memory. Instruction execution will be continued following the 64-bit word carrying the direct address, after monitor has responded to the message.

2. Indirect – the user job executes an 09 instruction. The 32-bit quantity immediately following the Exit Force instruction contains all zeros in bits 0 through 8 indicating that the rightmost eight bits of the 32-bit quantity contain the register designator of the register containing the address of the message. Instruction execution continues following the 32-bit pointer quantity, after monitor has responded to the message.

After interpreting the message and taking appropriate action, the monitor executes an Exit Force (09) back to the job to restart it at the point it performed its job to monitor exchange (with its own Exit Force instruction).

Messages

A basic set of messages are required for assisting the executing job:

- Read N blocks from file XXX sequentially into Backing Store beginning at address AAAAAA from current file position. All transfers are in blocks of 32,768 words.
- Write N blocks to file XXX sequentially from Backing Store beginning at address AAAAAA from current file position. All transfers are in blocks of 32,768 words.
- Rewind file to beginning block.
- Skip forward file N blocks.
- Close file (note that the user may not open any files. Thus the Close operation essentially locks out the file from further use during this job execution.
- Give file to user UUUU with password PW (used to release an explicit I/O file to tasks on the front-end processors (FEPs).
- Reduce time limit for this job to TT seconds.
- Reduce Backing Store allocation for this job to/by BB blocks.
- Send message to user UUUU. A user job must be logged on within one of the other processors, and enabled to receive messages. The user I.D. is a 16-bit quantity with all I.D.s greater than 8000 (hexadecimal) reserved for system tasks, and all I.D.s 7FFF (hexadecimal) and below assigned by the various application programmers.
- Enable message receipt – all other users/users UUUU, YYYY, ZZZZ only. Execute message processing program at job address AAAAA, save current execution address in job register 01 (data flag branch address).

- Disable message receiving - all users.
- Set error processor address at job address AAAAA.
- Suspend job temporarily (roll out job until external actions reactivate job).
- Job complete.
- Job abort, dump following areas of job memory to system error file (normally contains error messages and error parameters).

Exception Handling

Monitor validates the format and syntax of all messages passed from the user. In the event that a message is invalid in these areas the job is aborted, the system error log updated, error messages are sent to the FEPs, and the next job is initiated.

If the function requested is not in the table of monitor capabilities it is automatically passed onto the network trunk (see Distribution of Functions under System Philosophy above). The message response area in monitor for this message is set with a real time value from the current clock. After a delay of PPPPP seconds, if no response is returned, the message is considered unachievable. In this instance the monitor then enters the job error processing program (if that exit has been set by the appropriate message), or aborts the job (if the job did not set an error exit address).

If the message is responded to but is not achievable due to system errors or a resource being "down", the monitor elects to execute the job error processor (if set up) or aborts the job.

In all cases the time of message transmission, response and error conditions are recorded in the master system log, and the system error log (maintained by the FEPs) is updated. In the event of a continued failure of the system to achieve a particular function or message type, the monitor will suspend operation and alert the FEPs and the rest of the system.

Monitor/System Communications

Monitor can communicate with the outside world via two avenues: direct control of the input/output PDCs (programmable device controllers) with word-oriented messages using the monitor mode 02 (Transmit (R) to Channel (S) and Channel (S) to (T)) instruction, or indirect exchange of messages using polling techniques via the Backing Store. Software in the PDC can interpret the contents of the 64-bit word exchanged by this means to perform certain functions. In some cases the rightmost 48 bits of this word will contain an address in Backing Store. In other cases the 64-bit data contains several fields for use by the PDC and monitor for communicating maintenance, diagnostic, degradation, setup, restart, and other internal functions. Three major types of messages are proposed:

- Immediate – The entire message is contained in the one 64-bit word exchanged by the 02 instruction.
- Direct – The function code is contained in the leftmost 16 bits of the exchange word; the address points to the actual message in Backing Store.
- Indirect – The function code and sequence of system commands are all contained in a list in Backing Store; the list contains within it the address pointer to the messages to be transmitted.

- Acknowledge (receipt of incoming monitor or job message, plus time stamp).
- Reject (cannot perform function requested because it is not in FMP resource table).
- All job mode messages are legal in monitor mode.
- Job suspended, rollout complete.
- Job aborted, dayfile and error log information at DDDDD and EEEE respectively.
- Transmit dayfile information from DDDDD.
- Transmit error log information from EEEE.
- Transmit maintenance log information from MMMM.
- Job complete, rollout complete.
- Where is file.
- Is file open.
- Degrade available Backing Store to NNN blocks.
- Check real time clock synchronization.
- Perform I/O operation from I/O list at IIII
- Load I/O list at IIII
- Reject because resource failed, or job addressed not in FMP
- Assign new job, queue information at QQQQQ

Exception Conditions

All incoming messages pass through the PDC which either plants them in Backing Store (in a block reserved for monitor) or sends them direct upon a 02 instruction poll from the monitor. If the monitor fails to poll after a given period of time, or the function requested is not performed by some set time, the PDC assumes the FMP to be disabled and alerts the FEPs.

If monitor is unable to complete a critical function, such as real time clock synchronization, or if messages or responses appear garbled, the PDC will allert the FEP. A Maintenance Control Unit function can then be initiated to determine the condition of the FMP, and bring everything to a halt if need be.

If monitor is unable to get completion of messages, it will first switch to alternate PDCs, alternate FEP addressees, and finally halt and alert the Maintenance Control Unit.

Input/Output Handling

Input and output is controlled by communications between the monitor and the PDC. Actual data transfers always take place from and to the Backing Store, under control of the PDC. The PDC provides addresses to the input/output interface within the Swap Unit, block counts, and function (read or write). Data exchange between the PDC and the Swap Unit is monitored directly by the PDC, as its own internal counter follows the 32-bit half-word transfers across the input/output trunk. When a 512-word block has been fully transferred to the PDC buffer, a trunk input/output operation is initiated (for output from the FMP). For input the trunk input/output fills a 512-word buffer in the PDC before the PDC to Swap Unit transfer is initiated.

For file transfers the PDC receives a pointer to the input/output list for that file. The input/output list contains the following information:

- Header word containing file identification
- Open status (read,write,read/write)
- Position pointer into the file map of current location
- First block of file
- Last block of file
- Unit number and logical block number
- Address of first and last entry in input/output list
- A variable length list of entries giving the disk unit
- Disk block address and number of consecutive blocks for this segment of the file

A file is then a collection of data that may be spread over a number of disks, in noncontiguous chunks. The file map is kept with the file on the disk and transferred to Backing Store by the FEP (Front-End Processor) which opened the file for the executing job. Disk unit addresses are 16 bits long with all addresses of 8000 (hexadecimal) or larger specifying that the disks are multiple units using multiple trunks and, thus, multiple PDCs for the parallel transfer of data. In the case where monitor detects a multiple disk drive file (alternate blocks on each drive) a PDC will be alerted for each trunk, and a separate file list pointed to for each PDC. The set of file lists, either one (for one disk transfer), two (for two parallel disks transfering alternate blocks), or four (a maximum of four disks can be transferring simultaneously), are separted into individual input/output lists which define the file. In degraded mode a single PDC can alternate through the lists, transferring first one block from one disk, the second from another, and so on.

File size is defined by the FEP during job assembly time, and may not be extended by the FMP job or its monitor.

Input and output of data can be carried on with other system components in the same manner as the disk system, however, the data transferred need not be in minimum units of 32,768 words, since many attached processors need not contain that much buffering. Addresses in the file list below 4000 (hexadecimal) designate other components on the trunk (such as the FEPs, or low-speed devices). The command word sent to the PDC from the monitor in such cases carries a count of words to be transferred rather than blocks of words.

Maintenance Interface

The Maintenance Control Unit (MCU) communicates with the FMP over any one or more designated channels of the network trunk. Any PDC can have the particular address and password loaded by its software at autoload time permitting it to accept MCU messages, and toggle the special maintenance control bits provided in the FMP CPU (see Section 3.6, FMP Functional Computer Specification). These control lines provide hardware level control and monitoring capability for any permitted processor on the trunk.

Special Lines

The FMP Functional Specification defines a set of lines (established by the PDC acting as the MCU interface) whose function is to control degradation, configuration, and activity of the FMP. Lines such as 'stop', 'disable instruction overlap' are needed for system failures and during scheduled maintenance. The FMP monitor must contain the capability to exercise the options defined by these lines. Until more detailed CPU design is completed these special functions must be limited to their STAR-100 counterparts discussed in Section 3.6.1.

Messages

In addition to the messages discussed in Monitor/System Communications above, the monitor can issue a set of privileged messages to the MCU processor, via the PDC. Included in this set of messages would be:

1. Disable instruction overlap – vector, map, swap, or scalar
2. Disable SECDED error check, leave syndrome bits unmodified (used during diagnostic operation)
3. Force SECDED error on trunk KK (diagnostics only)
4. Help! (Undefined ailment, illogical combination of events discovered by monitor.)
5. New job to be initiated, rotate assignment of pipelines.

Degradation

The only mode of degraded operation permitted for the FMP is a reduction of the Backing Store hardware to a minimum of 32-million words and a reduction in the Main Memory to a minimum of two-million words. This degraded mode permits the maintenance of memory units off-line, since they are powered and cooled in those minimal unit quantities. Memory configuration is specified by a set of control bits preset by the MCU in a message to the appropriate PDC when the FMP is in the master clear state only. For purposes of addressing changes when in degraded mode, the memory is first divided into upper and lower (with each half being able to be the lower half-memory in degraded mode). Thus the first level of degradation consists of cutting the available memory in half, changing the apparent physical addresses to take the healthy half of memory into the lower address space and locking out address references to the sick half memory.

In this mode the maintenance station can enable memory references by the monitor or selected PDCs to provide diagnostic facilities at various clock rates.

The Scalar Processor is capable of operating with the Vector and Map Units disabled, and even powered off.

THE SYSTEM FUNCTIONS

The basic approach in the FMP Operating System is to rely heavily on the utilization of functions already implemented in existing software and operating on existing computing hardware. To meet this objective in the time frame established for the FMP installation requires the minimum disruption and redesign of the standard operating system components. This is accomplished by three techniques:

a. Programming of the PDC to simulate interfaces already known by the existing software systems. Basic OS drivers can remain intact in most instances. Machines of alien architecture to each other but with massive entrenched software can be interfaced for relatively small cost.

b. Constraint of the number and complexity of functions required by the FMP, and making the FMP responsible for any functions with extremely fast response time requirements (such as the reading and writing of the major data base disk systems).

c. Providing most FMP services with job mode applications programs written in higher-level languages.

Input/Output for the FMP

Management of all files in the FMP system will be handled by the FEP. Data transfers between elements of the system and file resources will be conducted directly by those elements, however. The FEPs must supply the functions normally required of general-purpose computers:

1. Open file, verify access, set access mode (read, write, read/write).
2. Close file, dispose of files (destroy, archive, keep in place).
3. Allocate file, and file space.
4. Expand and contract file space.
5. Move files.
6. Search, and retrieve files.
7. Assemble files from subfiles.
8. Build file maps for FMP disk transfers.

These functions can be provided by most known operating systems, such as the CYBER NOS and NOS/BE systems. By judicious programming of the attached PDC it should be possible for the FEP operating system to deal with attached peripherals (for which software is already in place) in the same manner as if the peripherals were directly attached in today's customary manner.

Interface for the FMP can be provided by an FMP message handler operating in job mode. This message handler can perform file searches and retrievals using the structured access file software native to the FEPs, and transmit data to the FMP, or its high-speed disks, using only one specially developed trunk driver which would have to operate in the CYBER PPU.

Job Scheduling for FMP

The FEP complex must perform the scheduling of sequences of jobs to be executed on the FMP.

The decision on when and what to execute from the queue given to the FMP by the FEPs still remains for the FMP monitor as described in Allocation of FMP Resources above.

For purposes of scheduling, the high performance disk system is considered part of the memory resources that belong to the FMP and must be allocated by the FEP. Taking the FEPs' scheduling responsibilities for a job in order:

1. Assessment of available system resources (tabulating amount and location of high performance disk, low performance disk, central FMP memory, Backing Store, available terminals, archive, and jobs already queued for FMP).
2. Choice of next job to be assembled for the FMP – based on incoming requests for service
   a. priority,
   b. resources required,
   c. time limit (quick interactive or full run),
   d. availability of components of the job (programs and data).
3. Reservation of high performance disk space.
4. Layout of contiguous job space to be rolled into main FMP memory.
5. Layout of contiguous storage area for the job to be allocated in the Backing Store.
6. Building of file map for job and data files on the high-speed disks.
7. Creation of file entity with security and modes of permitted access.
8. Storage of file header, file map on selected disk.
9. Movement of program file from local storage to job file.
10. Movement of data base to job data files.
11. Closing of data and program files.
12. Transmission of queue request to FMP.
13. Logging of all the above activities.
14. On close file from FMP, evaluate the disposition code and perform required operation (note that only FEPs can destroy or otherwise dispose of files).
15. Maintain job status from submission to FMP to job completion.
16. Provide job mode programs for accounting.
17. At job mode completion, follow through on disposition codes.

Exception Handling for FMP

The front-end processors provide two functional entities which deal with system exceptions:

1. Maintenance Control Unit (MCU) processing.
2. Front-end system management.

Programs representing the MCU functions will be prepared for at least the FEP computers. These modules will be given the MCU privileged password for communications with the FMP and control of the special MCU lines (channels) within the FMP CPU. These functions are defined by the MCU lines available, and additional monitor communications defined by as yet to be completed design work.

The FEP computers are responsible for recording all errors and determining what disposition to make of the partial data and remaining program and job data that are salvaged from aborted jobs. Restart, recovery and retrench functions are programmed in the FEP, and the semi-automated decision to apply such strategies driven by FEP programs.

Input/Output Handling for Other Attached Processors

In order to maintain full control over all the system resources, the FEPs should perform all file management functions afforded the FMP. In some instances a particular processor may require a unique attachment to a unique peripheral which need not be attached to the trunk. In these cases management of the attached peripheral resource becomes the responsibility of the processor that "owns" the resource. In all other cases the FEP or the PDC attached to a resource acts as the resource manager. As in the FMP, the actual data transfers bypass the FEP and go directly between the processor and the requisitioned resource.

OPERATING SYSTEM STRUCTURE AND IMPLEMENTATION

Within the limits of the operating system implementations for the front-end processing systems already in existence, the FMP operating system should conform to certain architectural and implementation ground rules.

Programming Language

All new components of the operating system should be written in a higher-level language. The same higher-level language should be used regardless of the processor being programmed. The programmable device controllers presently are implemented in assembly language because they operate in real-time, and each machine cycle must be accounted for. This situation may be unavoidable, however some effort should be expended to see if major portions of PDC programs could be implemented in a higher-level language.

The choice of higher-level language is not as important as the decision to require the use of one at all, however, the language PASCAL is becoming a pseudo-standard software programming language which is implemented on a variety of hardware, and widely taught in computer science courses. At this point

PASCAL would appear to be the best choice since many language processors are being created and bootstrap systems produced for it.

A dialect (if not the entire language) of PASCAL should be used as the primary programming language for the PDCs wherever possible. The compiler system could reside on one particular mainframe producing system code for the other attached processors, including the FMP. Development of such a system, with attendant documentation, debugging, and system design control aids should be initiated as soon as possible.

Modularization

One of the outgrowths of the distributed system philosophy is an enforced modularization of the software. As functions are distributed outward from a central computing facility into a network of ever smaller computers, the functional portions become the entire program module for the distributed computers. By enforcing a set of message standards for communications between such programs, a rigid set of boundaries can be defined for each and every module in a system.

What remains is to break up all system functions into modules of like kind regardless of whether or not they actually reside together in a large central computer or in fragmented smaller machines. This means imposing message disciplines and constraints on intermodule communication, even within a common computer. Thus at some later date, a module could be moved to a different computer, with the identical messages being passed over the network instead of internally within the memory of a single computer.

A set of module implementation specifications must therefore be created and placed in this portion of any software specification, at a later phase in the project.

Configuration Flexibility

The FMP system must be capable of functioning even in severe states of degradation. This means that, as a minimum, the FMP in its most degraded state of memory, at least one FEP, sufficient disk space to queue one job, plus one interconnecting trunk must be available for completing flow model solutions. The system must also be able to sustain interactive communications, and to queue jobs on mass storage during interruptions of service by any of the other system components such as the FMP; archival storage subsystem, graphics subsystem, or high performance mass storage system.

The operating system must cope with any possible combinations of configurations arising dynamically in an operating day. Equipments must be able to be taken off-line, without restarting the operating system, and new equipments installed without reassembly, recompilation, or other massive remapping of the operating system (the exception to this would be the reassembly of a small portion of the operating system, say the disk driver, to accommodate a new type of disk system). In most cases of this sort, however, the PDC (Programmable Device Controller) is expected to insulate the bulk of the operating system from such specific hardware changes.

Extensibility

Two factors militate against the creation of an ideal system design and implementation on the first installation of the FMP complex:

1. The time and resources available to generate the first production system make it necessary to be ruthless about eliminating all but the most necessary production system functions from the first system implementations.

2. The operating system must be adapted to the actual production requirements as learned from operation of the FMP system over an extended period of time. Decisions such as the location and allocation of system functions according to the "Distributed System Philosophy" must be reconsidered as actual experience detects bottlenecks and resource imbalance.

It should be obvious from past computer system experience that these factors require a high degree of flexibility and adaptability in the operating system, as understanding of the FMP and its participation in the airframe design process matures. It should be possible to redesign, recompile, load and test experimental functional modules while the total system is on-line. Further, it should be possible to introduce new functional modules into the system, while the system is on the air in production mode, without fear of destroying the remainder of the system. Thus if an error occurs, it can only affect the modified or new modules and functions.

Without this feature it can be expected that, although the hardware availability might be high due to extensive reliability engineering and the basic software might be reliable for the same reason, new software development requires so much dedicated system time that actual customer availability is severely affected. This is of particular concern when it is realized that certain components in the system (such as the FMP) are one-of-a-kind, as well as the total configuration being unique. Thus final debugging and testing would have to be done on the actual FMP complex. This consideration alone is so important it must be stressed with utmost vigor, as experience on the STAR-100 systems has shown. Since few STARs are available for software checkout, the STAR data center has become a major test and integration facility, substantially reducing the system availability to general customer utilization.

The operating system architecture then must be defined with this goal in mind, and thence the manner and means of modularization and message implementation can be dictated.

RAM (RELIABILITY, AVAILABILITY AND MAINTAINABILITY)

The software system for a complex as large as envisioned for the FMP installation is a major factor in system reliability and availability as is the hardware. A detailed specification of the RAM requirements for each software subsystem must be developed for the FMP as objectives for the initial and final production operating systems, and be included in this portion of a software specification. Two items requiring special care and attention are documentation and stability.

Documentation

Operating systems with many independent nodes, and a variety of functional modules require extensive documentation in excess of the commonly used listings, flow charts, and module descriptions. There must be established a set of documentation standards which engage the following issues:

1. Extensive definition of all terminology in an alphabetic glossary (for example the term "message" must be rigorously defined and each of the fields in the message must be defined in terms of their meaning).

2. A master outline of the operating system documentation must be created with a place for every single piece of system documentation (including listings of the source language, job control setups to form the loaded operating system, and any other preparation details).

3. A set of required documentation rules to be used for imbedding thought as well as technique in the program listings is required. Thus management ground rules are needed such as "all term names must be fully spelled out in source language symbols, at the cost of more typing time at a terminal or more keypunch strokes" (phrases like "message length" are more desirable than "MSGLNG" for readability).

4. A set of programming ground rules which aid documentation and future comprehensibility must be established and enforced. The structured programming schemes aided by the structure of PASCAL are one major example of an enforced technique.

5. A theory of operation manual must be prepared for each component of the distributed system, as an overview of the total system.

6. A message dictionary (with every form and every function described) must be created for the whole system.

7. A master library (preferably automated to make updates timely) must be established for current, and past, versions of the source code, flow charts (or flow descriptions), and all versions of the software objectives, design and test objectives documentation.

Stability

A detailed breakdown of the stability requirements for each software module must be developed before the system implementation commences. That specification would then be placed here. The overall requirement is that the total software system (including operating system, compilers, and utilities) must have a mean time to failure no worse than the hardware system.

Obviously a perfect system would be desired, however practical experience shows that some degree of instability will remain in a system as long as new applications are submitted to it and, in particular, if new or modified functional modules of the operating system continue to be introduced throughout the lifetime of the system.

REFERENCES

1 Brown, P.J.: The ML/I Macro Processor. Communications of the ACM, vol. 10, no.10, Oct. 1967.

2 Waite, W.: A Language-Independent Macro Processor. Communications of the ACM, vol. 10, no. 7, July, 1967.

Section 5

APPENDIXES

APPENDIX A

FMP INSTRUCTION

SPECIFICATION

CDC [R] FLOW MODEL PROCESSOR

INSTRUCTION DESCRIPTIONS

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

1.0 SCOPE

This specification for the CDC FLOW MODEL PROCESSOR (FMP) is to be used in conjunction with the CDC STAR-100 Computer Specifications. It is assumed that the reader is familiar with the concepts and terminology described in those documents.

This is NOT a reference manual for user's groups. This document is written expressly for logic designers and diagnostic programmers.

2.0 APPLICABLE DOCUMENTS

10354637 . CDC FLOW MODEL PROCESSOR Functional Computer Specification

3.0 PERFORMANCE REQUIREMENTS

3.1 General Description

3.1.1 Instruction Formats and Types

3.1.1.1 Instruction Formats - all fields are 8 bits unless otherwise specified.

3.1.1.1.1 N/A

3.1.1.1.2 N/A

3.1.1.1.3 N/A

3.1.1.1.4 Format 4

F	R	S	T
Function	Source 1	Source 2	Destination

3.1.1.1.5 Format 5

F	R	
Function	Destination	I48

3.1.1.1.6 Format 6

F	R	
Function	Destination	I16

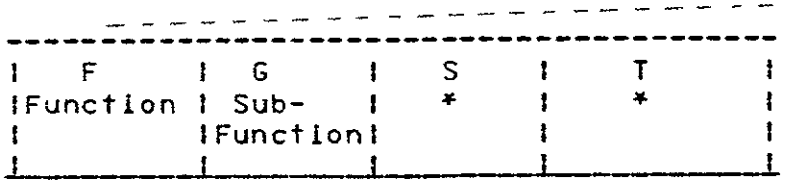
3.1.1.1.7 Format 7

F	R	S	T
Function	*	*	*

*Described where used

3.1.1.1.8 N/A

3.1.1.1.9 Format 9

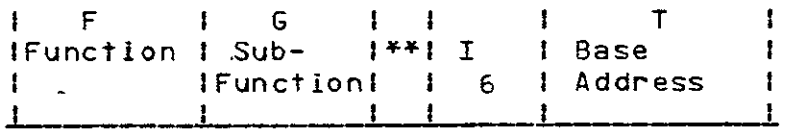


3.1.1.1.10 Format A

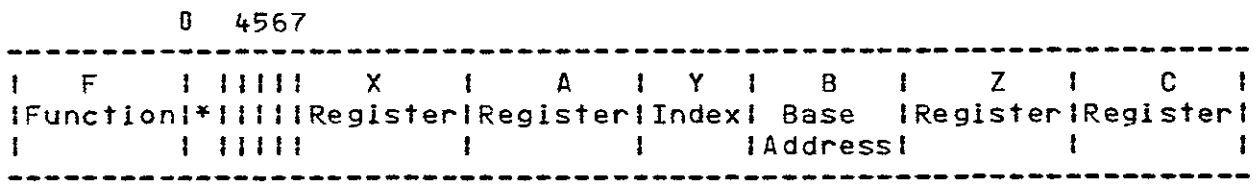
* Described where used

3.1.1.1.11 Format B

** Unused areas must be cleared to zeros



3.1.1.1.12 Format C



* Unused area must be cleared to zeros.

RADL

3.1.1.1.13 Format D

F Function	*	PC	Parcel 1
	4	4	16

* Unused area must be cleared to zeros.

3.1.1.1.14 Subformats

3.1.1.1.14.1 Format D1 Parcel 16 Bits

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15				
A	B	C	D	E

3.1.1.1.14.2 Format D2 Parcel 64 Bits

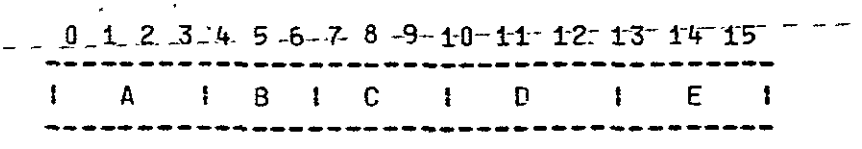
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16->31					
A	B	C	D	Z	E

32->59 60 61 62 63		
F	G	

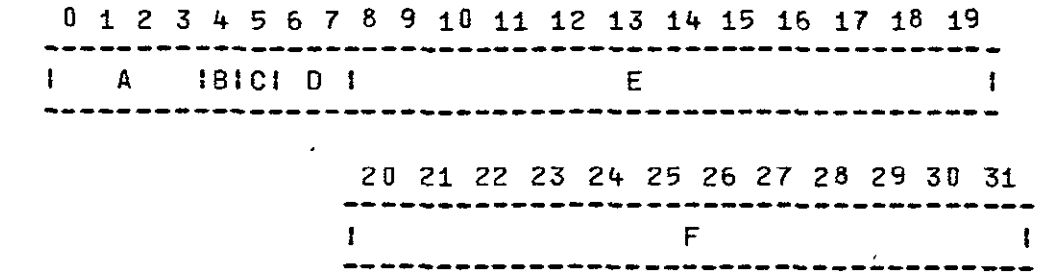
3.1.1.1.14.3 Format D3 Parcel 16 Bits

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15		
A	B	C

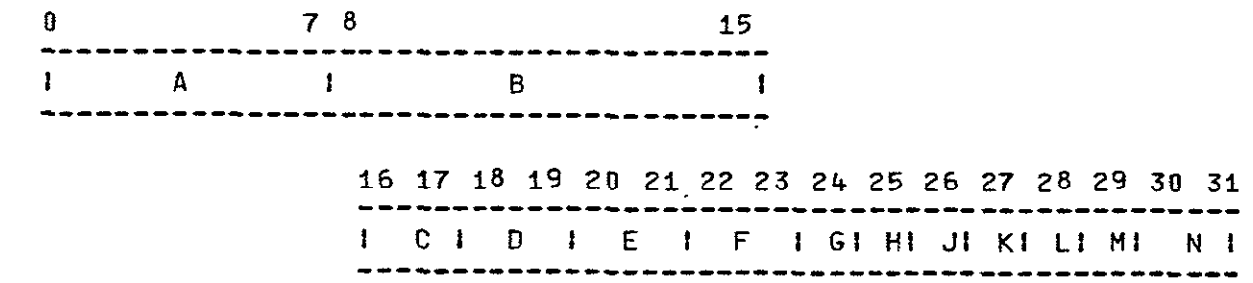
3.1.1.1.14.4 Format D4 Parcel 16 Bits



3.1.1.1.14.5 Format D5 Parcel 32 Bits



3.1.1.1.15 Format E



3.1.1.2 Instruction Types

3.1.1.2.1 Register Instructions (RG)

In the register instructions, all operand sources and all result destinations are registers. R, S, and T each designate the contents of one of 256 registers.

A register may be used to hold one or both source operands as well as the result. Special case: if register 00 is designated as a source or result register, see Section 3.1.7.

Unless stated differently in the instruction description in all register-to-register operations, the contents of the source registers are unchanged and the destination register is cleared before the result is transferred into it.

3.1.1.2.2 Index Instructions (IN)

The index instructions are used primarily in performing numerical calculations on field lengths and addresses.

The term, replace, means replace only the specified bits. The phrase, replace the right-most 48 bits ..., implies that the left-most 16 bits are not altered.

3.1.1.2.3 Branch Instructions (BR)

Branch conditions may be determined by examining single bits, a 48-bit index, 32-bit floating-point operands or 64-bit floating-point operands. A special branch is provided to enter and leave the monitor program. All item counts in branch instructions are in half-words.

3.1.1.2.4 Stream Instructions (SM)

Stream instructions operate on ordered sets of data, executing in either the Swap, Map, Buffer, or Vector Units. A set of stream instructions consists of a 32-bit header packet (see format D, 3.1.1.1.13) and a variable number of 32-bit packets containing parcels of data to be transmitted from the Scalar Processor to one of the three stream units (Vector, Map, Swap). Packets are fixed length (32-bits) while parcels are 16-bit, 32-bit, or 64-bit length. The header and its associated packets of instructions constitute a form of high level micro-code for the particular function.

Referring to format D under section 3.1.1.1.13, the stream designators are defined as follows:

F - Eight-bit instruction code

PC - Four-bit packet count specifying the number of 32-bit packets following the header packet. Note that a count of zero implies that the entire stream instruction is contained in the first 32-bit packet containing the header.

The various subformats for the microinstruction parcels are given in section 3.1.1.1.14, formats D1 through D5. The fields take on meanings dependent upon their use in a given parcel, for a given unit. In general, the field designators are used as follows:

A - Subfunction field (for example, R1 SETUP).

B - Reference mode (immediate for addresses imbedded in the instruction, or indirect for addresses in the specified registers in the Scalar Processor register file).

C - Word size (32-bit or 64-bit).

(continued)

3.1.1.2.4 (Cont.)

D - Extension code, source field, or length field depending on the instruction.

E - Register file pointer, length field, base address, or source field depending on the instruction.

F - Memory address in Map and Buffer instructions, source field in Vector Unit instructions.

G - Lower address bits (shift count) for Map instructions, round flag in Vector Unit instructions.

H,J - Complement flag for B and D trunks in Vector Unit.

K - Null field.

L,M,N - Vector Unit result bus select.

Z - Unused

3.1.1.2.5 N/A

3.1.1.2.6 N/A

3.1.1.2.7 N/A

3.1.1.2.8 N/A

3.1.1.2.9 Monitor Instructions (MN)

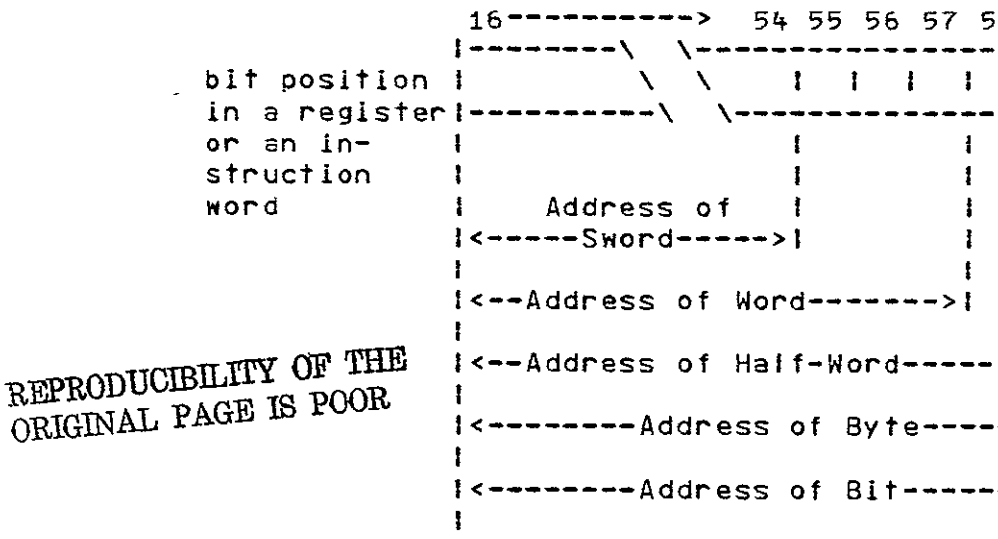
Monitor instructions perform as described only when in monitor mode. When not in monitor mode, the monitor instructions perform as an illegal instruction would (see Section 3.1.4.2.2).

3.1.1.2.10 Non-Typical Instruction (NT)

The format and operation of these instructions are completely described under the individual instruction write-ups.

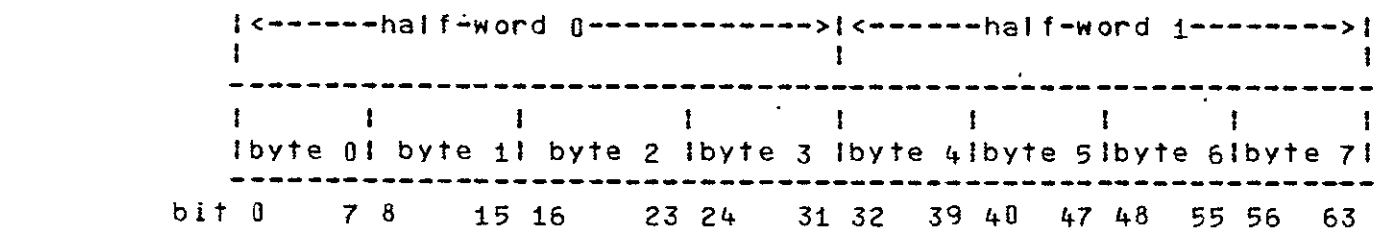
3.1.2 Addressing

Groups of bits in an address should be thought of as addressing various units of storage as illustrated in the chart below.



Within a word, bits, bytes, and half-words are always numbered from left to right. The lowest addressed bit, byte, or half-word is always the left-most bit, byte, or half-word in the word.

All addresses are 48-bit quantities and contain enough information to reference a specific bit. Depending on the usage of an address, a certain number of the right-most bits in the address are ignored. For example, if a byte is being read, the right-most three bits of the address being used to reference it are ignored. Depending on the instruction, operands are counted on a bit, byte, half-word or word basis.



The above figure illustrates the relative location of each bit, byte and half-word within a 64-bit word.

(continued)

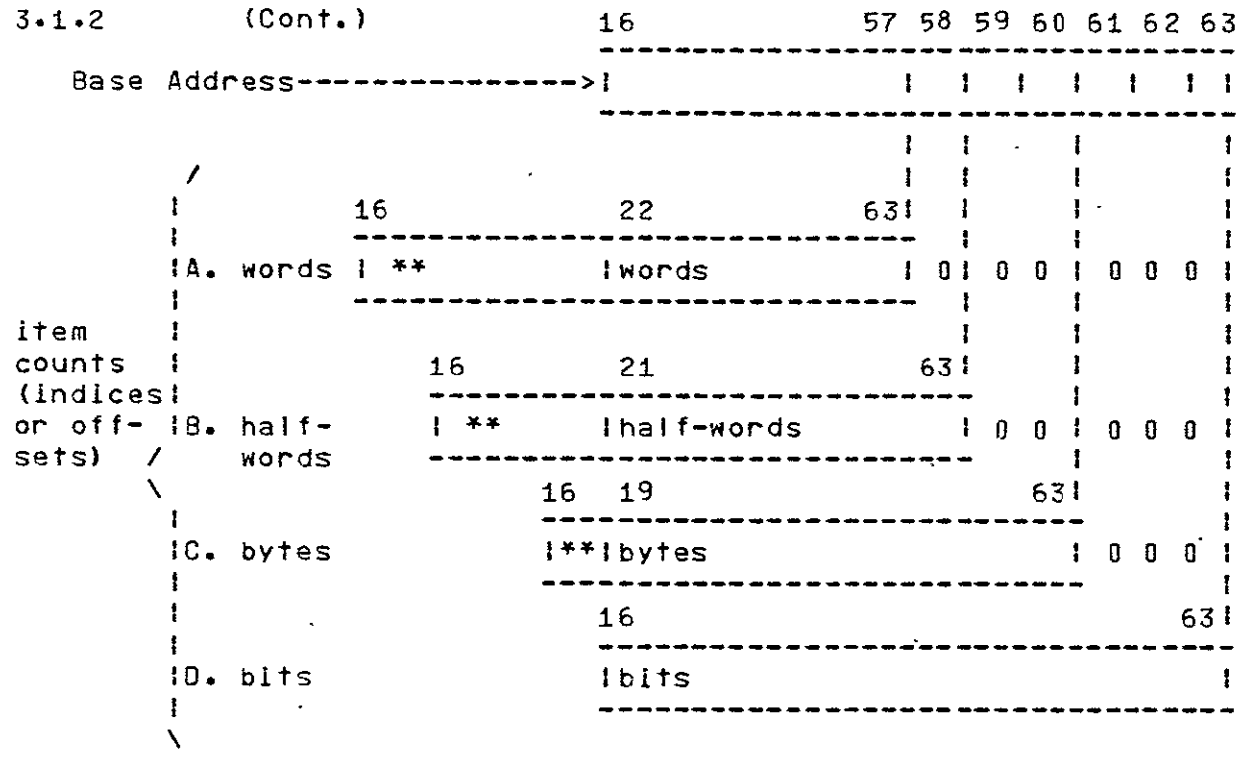
3.1.2 (Cont.)

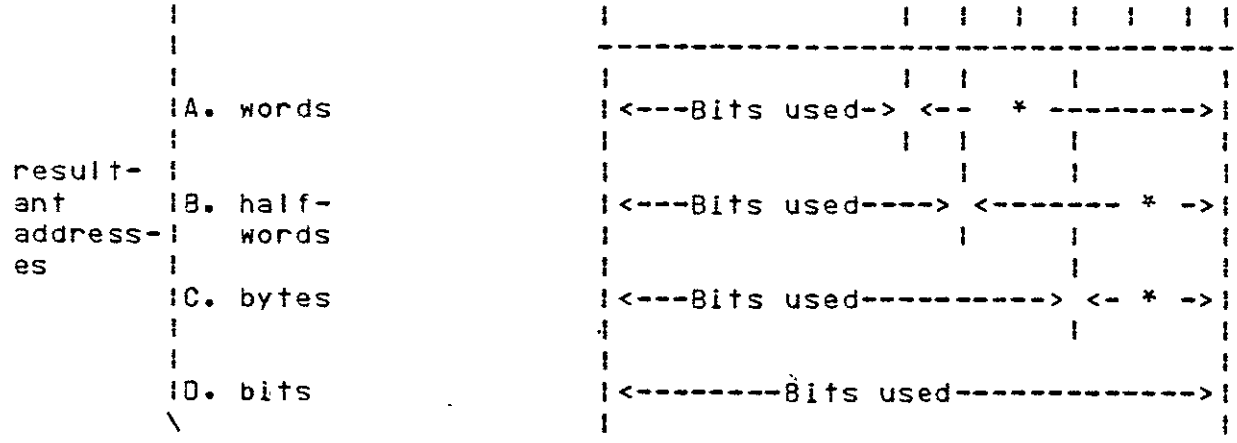
If it is necessary to add addresses and item counts (indices or offsets), the item count is shifted left end off until it is properly aligned with the address. Binary zeros are attached to the right end of the quantity being shifted.

The result of the addition always addresses a quantity having the same unit as the item count. for instance, if a byte count is added to any address, the result references a byte. This means that the right-most three bits of the address will be ignored. The following chart summarizes the process of adding an item count to an address and shows which bits are ignored in the resulting address.

(continued)

3.1.2 (Cont.)





* These bits in the resultant address are ignored.

** These bits in the index or offset are shifted off and do not enter the address calculation.

3.1.2.1 Memory Hierarchy Addressing

There are four levels of memory accessible to the programmer: Register File, Vector Unit Buffer (VUB), Main Memory and Backing Store. Each of these memories can be addressed by instructions in two different ways: direct and indirect.

3.1.2.1.1 Direct Addressing

Memory addresses can be contained within the instruction itself, and are therefore called direct addresses. In the case of the Register File, such direct addresses are called register designators, each designator being assigned a name such as R, S, or T. A direct register file reference in an instruction can access any one of 256 registers (64-bit or 32-bit).

Vector Unit Buffers can contain from one to four thousand words each. Thus a field of twelve bits is established (see formats D1 through D5) for the insertion of the buffer address.

Main memory addresses permit accessing up to 128 million 64-bit words, thus 27 bits are established as the direct address field for this format.

Backing store references always access data in 32,768 64-bit word blocks. Twenty-seven bits of direct address field are allocated to permit referencing up to 128 million of these blocks, or 4.4×10^{12} words of data.

The actual amount of physical memory present is determined by the specific machine configuration. Memory not actually in existence causes a data flag branch to occur at the time of reference.

3.1.2.1.2 Indirect Addressing

The memory addresses can also appear in 64-bit registers in the Register File. Thus an instruction can reference memory indirectly by giving the appropriate register file address, which points to the register containing the desired memory address. The allocation of address bits in the designated registers are as follows:

- Register file address -- rightmost 8 bits of the indirect register (bits 56-63). This allocation is used only for the SWAP (7D) instruction.
- Vector buffer unit addresses -- all addresses are bit addresses, thus the rightmost 21 bits of the indirect register are used (bits 43-63).
- Main memory addresses -- all addresses are bit addresses, thus the rightmost 33 bits of the indirect register are used (bits 31-63).
- Backing store memory -- all addresses are bit addresses, thus the entire 48-bit address is used.

3.1.2.1.3 Illegal Addresses

Main memory addresses 0 through 100000_{16} are reserved for the operating system. Any reference to this address range by a job mode program results in a job mode illegal abort of the program in execution.

Main memory addresses 0 through 4000_{16} are reserved for the storage of the monitor's register file. Any reference to this area by a monitor mode memory access will cause a monitor mode illegal abort.

3.1.2.2 Instruction Addressing

Instructions are addressed on full-word and half-word boundaries. The instruction address counter will, therefore, be incremented by a half-word after executing a 32-bit instruction and by a full word after executing a 64-bit instruction. This allows instructions to be packed contiguously in storage. The following chart illustrates the various ways instructions may be packed within 64-bit words.

bit position

0 31	32 63
32-bit inst. *	64-bit inst. upper
64-bit inst. lower	64-bit inst. upper
64-bit inst. lower	32-bit inst. *
64-bit instruction	
32-bit inst. *	32-bit inst. *
*These could also be 32-bit packets of stream instructions.	

Note that a branch is possible to any of the instructions. The lower 5 bits in any branch address will always be interpreted as zeros.

3.1.3. Termination Rules

For instructions which terminate upon exhausting the length of a data field, data string or vector: if that item is exhausted prior to the first operand fetch, the instruction becomes a no op; no data is fetched and no data flags are altered.

3.1.3.1 Stream Instruction Termination

Stream instructions terminate when the result vector is exhausted. Source vectors which are exhausted before the result vector is exhausted are extended, as required, with the operand designated in the D field (extend code).

3.1.3.2 (N/A)

3.1.3.3 (N/A)

3.1.3.4 (N/A)

3.1.4 Definitions and Rules

3.1.4.1 Overlap of Operand and Result Fields

If the result field overlaps a source field such that elements of the result are stored in the source field before elements in this portion of the source field are read, undefined results may occur. That is, the source elements may be the original elements or they may be the newly-stored elements. The instruction's results may become undefined. Note that some specific instructions prohibit any overlap of source and destination fields. This restriction is included in the appropriate instruction descriptions.

3.1.4.2 Self-Modifying Programs, Undefined Instructions and Undefined Operands

3.1.4.2.1 Self-Modifying Programs [A2.0]

As a general rule, self-modifying programs are not allowed. See Appendix A2.0 for further details.

3.1.4.2.2 Illegal Instructions

An instruction with an unused function code is termed an illegal instruction and causes the following:

A. If in monitor mode, an automatic branch to the address specified by the contents of absolute register 4 is executed.

B. If in job mode, an exchange to monitor mode is performed with execution beginning at the address specified by the contents of absolute register 3.

3.1.4.2.3 Undefined Instructions

The instructions with a defined F code but which either have undefined bits set or specify an undefined operation cause undefined results.

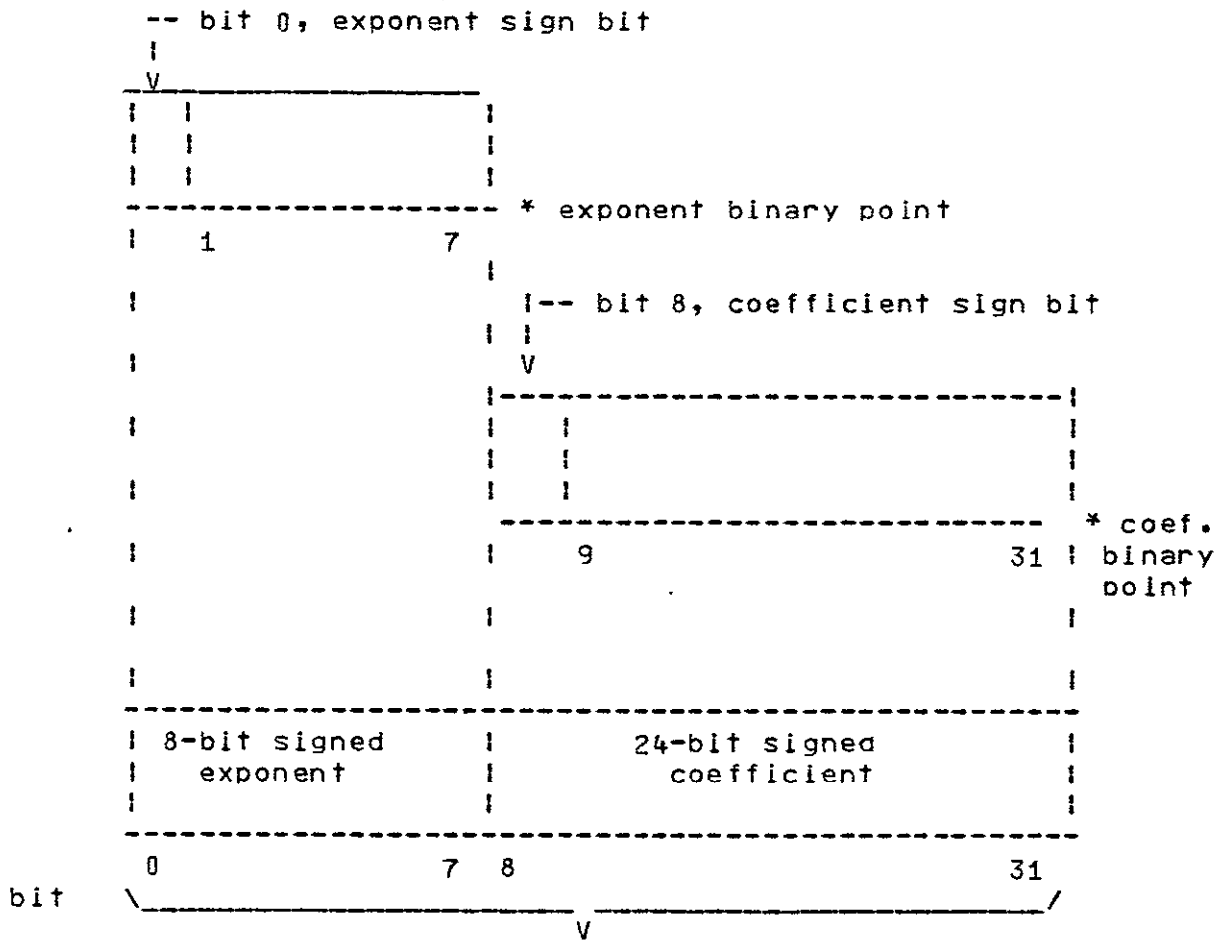
3.1.4.2.4 (N/A)

3.1.4.2.5 No op Instructions

The instructions that are defined as No op (no operation) instructions do not fetch data and do not alter data flags.

3.1.4.3 Floating-Point Format

3.1.4.3.1 32-Bit Floating-Point Format



32-bit floating-point number

There are two 32-bit half-words in every 64-bit word. A 32-bit floating-point number occupies a half-word.

A zero is a positive sign bit and a one is a negative sign bit for both the exponent and the coefficient.

Both the exponent and the coefficient are expressed as two's complement signed integers. Numbers are of the form (c) 2x where c is the 24-bit signed coefficient, x is the 8-bit signed exponent, and the base is 2.

(continued)

3.1.4.3.1 (Cont.)

The range of useful coefficients is from 800000 to $7FFFFF_{16}$.

This represents numbers of the range $-(2^{23})$ through $+(2^{23}-1)$.

The range of useful exponents is from 90_{16} to $6F_{16}$ which is from minus 112_{10} to plus 111_{10}. The values of 70_{16} through $8F_{16}$ all fall into a special end case range as defined by the following table.

X is any hexadecimal digit.

Element	Representation
Machine Zero	$8XXXXXXX_{16}$
Indefinite	$7XXXXXXX_{16}$

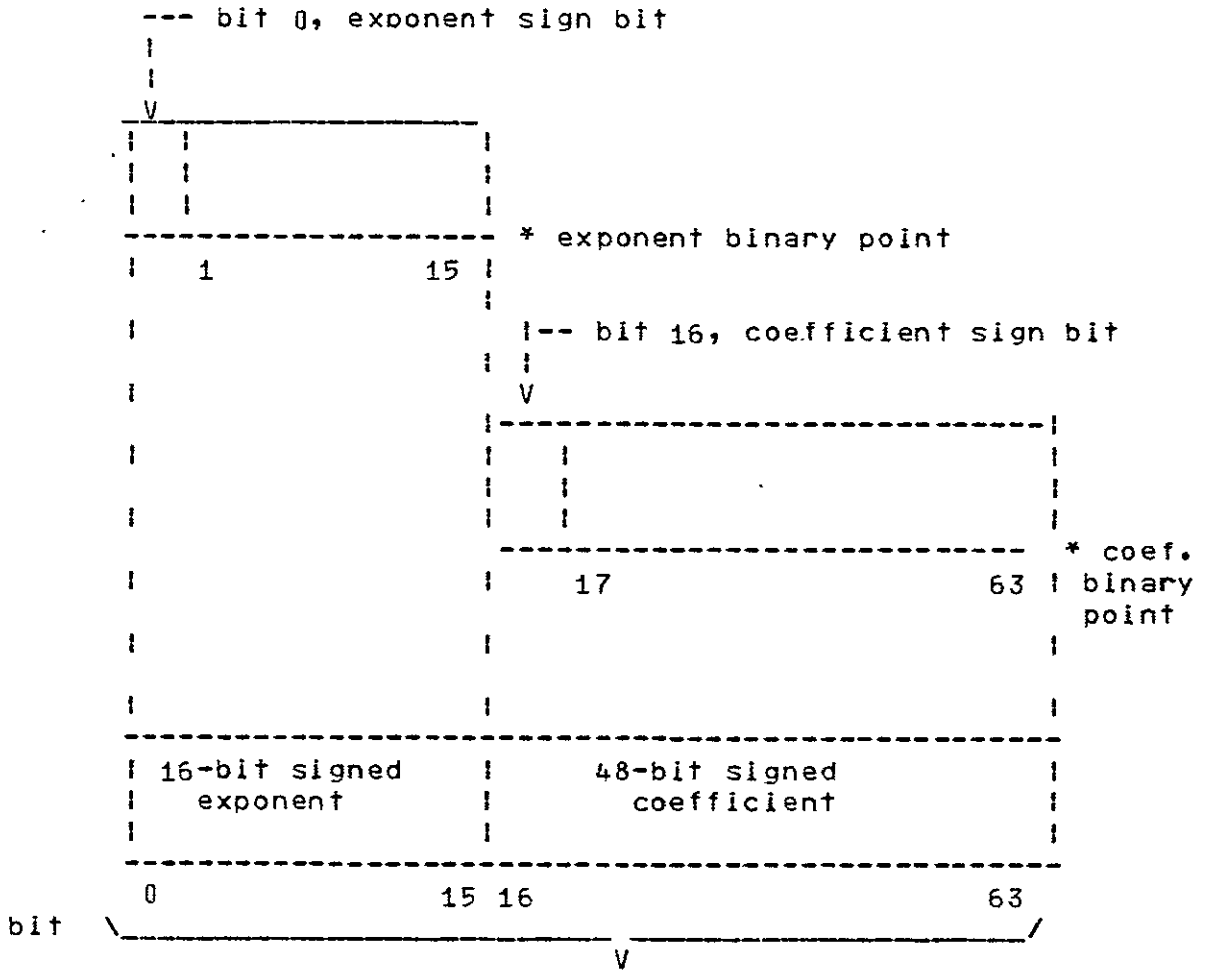
Examples of 32-bit floating-point format represented in base 16.

+1	00	000001
+1 normalized	EA	400000
-1	00	FFFFFF
-1 normalized	E9	800000
$+256_{10}$	00	000100

A floating-point number is normalized if the coefficient sign bit is different from the next bit to the right. This condition implies that the coefficient has been shifted to the left as far as possible. Note that an all zero coefficient requires special attention for normalized operations.

R A D L

3.1.4.3.2 64-bit Floating-Point Format



64-bit floating-point number

A 64-bit floating-point number is contained in a 64-bit word.

A zero is a positive sign bit and a one is a negative sign bit for both the exponent and the coefficient.

Both the exponent and the coefficient are expressed as two's complement signed integers. Numbers are of the form (c) 2x where c is the 48-bit signed coefficient, x is the 16-bit signed exponent, and the base is 2.

(continued)

3.1.4.3.2 (Cont.)

The range of useful coefficients is from 8000 0000 0000 0000 to 7FFF FFFF FFFF FFFF. This represents numbers of the range $-(2^{16}-1)^{47}$ through $+(2^{16}-1)^{47}$.

The range of useful exponents is from 9000 to 6FFF (16) which is from minus 28,672 (10) to plus 28,671 (10). The values of 7000 (16) through 8FFF (16) all fall into a special end case range as defined by the following table. X is any hexadecimal digit.

Element	Representation
Machine Zero	8XXXXXXXXXXXXXXX (16)
Indefinite	7XXXXXXXXXXXXXXX (16)

Examples of floating-point format represented in base 16

+1	0000 0000 0000 0001
+1 normalized	FFD2 4000 0000 0000
-1	0000 FFFF FFFF FFFF
-1 normalized	FFD1 8000 0000 0000
$+256_{10}$	0000 0000 0000 0100

A floating-point number is normalized if the coefficient sign bit is different from the next bit to the right. This condition implies that the coefficient has been shifted to the left as far as possible. Note that an all zero coefficient requires special attention for normalized operations.

3.1.4.4 End Cases

If indefinite is used as an operand in a floating-point instruction, both the upper and the lower results are indefinite.

For the cases listed below, 0 represents machine zero and N represents an operand which is neither machine zero nor indefinite.

0 ± 0 = 0	0 * 0 = 0	0 / 0 = Indefinite
0 ± N = ± N	0 * N = 0	0 / N = 0
N ± 0 = N	N * 0 = 0	N / 0 = Indefinite

3.1.4.5 Floating-Point Compare Rules

Several of the instructions compare two floating-point operands for:

a.	equality	(r) = (s)
b.	non-equality	(r) <> (s)
c.	greater than or equal to	(r) ≥ (s)
d.	less than	(r) < (s)

For these examples, the first operand is represented by (r) and the second operand by (s).

3.1.4.5.1 One or Both Operands Indefinite

If one operand is indefinite, no compare condition is met since indefinite is not: greater than, less than, equal to, nor not equal to any other operand.

If both operands are indefinite, the (r) = (s) and the (r) ≥ (s) conditions are met since indefinite is defined equal to indefinite.

3.1.4.5.2 Neither Operand Indefinite but One or Both Operands Machine Zero

Any non-indefinite, non-machine zero operand with a positive, non-zero, coefficient is strictly greater than machine zero.

Any non-indefinite, non-machine zero operand with a negative coefficient is strictly less than machine zero.

Machine zero is equal only to itself and any number having a finite exponent and an all zero coefficient.

3.1.4.5.3 Neither Operand Indefinite Nor Machine Zero

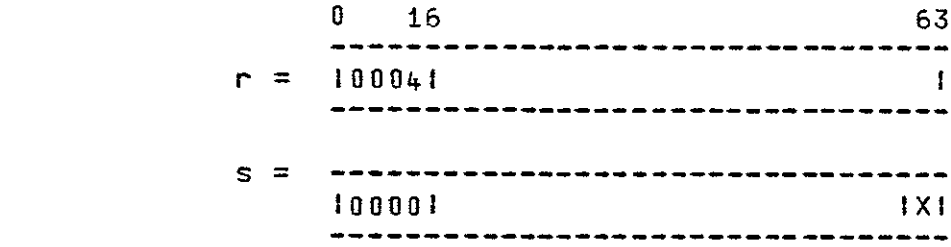
A. If the signs of the coefficients of the two operands are unlike, the operands are unequal and the operand with the positive coefficient is the larger of the two.
B. If the signs of the two coefficients are alike, a floating-point subtract upper is performed; operand r minus operand s.

Condition met criteria are analyzed as follows:

a. If the upper 48 bits of the result coefficient are all zeros (r) = (s)
b. If the upper 48 bits of the result coefficient are not all zeros (r) <> (s)
c. If the result coefficient is positive (r) ≥ (s)
d. If the result coefficient is negative (r) < (s)

The above criteria (a and b) for equality and non-equality do not guarantee that if r = s, that s = r when the following is true:

a. The operands have unequal exponents.
b. "1" bits exist in any of the right-most bit positions of the coefficient which will be shifted off the right during alignment of the smaller exponent. For example:



Exponent difference = 4

If x = 0 then r = s implies s = r

If x <> 0 then if r = s, s <> r
or if s = r, r <> s

(continued)

3.1.4.5.3 (Cont.)

The order of events of the floating-point subtract upper is first to complement the subtrahend, then align the coefficient associated with the smaller exponent and finally to perform a floating-point add operation. The following is an example of r = s but s <> r.

Operand r =	0100	0000	0000	1001	
s =	0104	0000	0000	0100	
Complement s	0104	FFFF	FFFF	FF00	
Align r	0104	0000	0000	0100	1
Add aligned r and complemented s	0104	0000	0000	0000	1

Since the upper 48 bits of the result coefficient are all zeros, the pair of operands are considered equal. However, if the operands are interchanged, the following happens:

Operand r =	0104	0000	0000	0100	
s =	0100	0000	0000	1001	
Complement s	0100	FFFF	FFFF	EFFF	
Align s	0104	FFFF	FFFF	FEFF	F
Add r and complemented, aligned s	0104	0000	0000	0100	
	0104	FFFF	FFFF	FEFF	F
	0104	FFFF	FFFF	FFFF	F

Since the upper 48 bits of the result coefficient are not all zeros, the pair of operands are considered unequal.

3.1.4.6 Upper and Lower Results

The floating-point add, subtract and multiply instructions generate a result coefficient twice the length of the source operands' coefficients. The left and right halves of this result are called the upper result (U) and the lower result (L), respectively.

The sign bit of the lower result's coefficient is not affected in a lower operation and remains at zero in two's complement arithmetic. The other bits of the lower coefficient receive no special treatment. Remember that a lower result is not meaningful alone, but it must be used in conjunction with its associated upper result.

Sections 3.1.4.6.1 - 3.1.4.6.4 are written for 64-bit operands. For 32-bit operands, substitute 47 for 95, 46 for 94, 23 for 47, and 22 for 46 where the latter numbers appear.

3.1.4.6.1 Right Normalization

When the result coefficient overflows its register, a right shift of one place is necessary. In this case, the entire 95-bit result is shifted right one place with sign extension and one is added to the exponent. This operation is known as right-normalization and it is done, when necessary, even if normalization is not explicitly specified by the instruction. This may cause exponent overflow; if so, the result is set to indefinite and data flag bit 42 may be set.

3.1.4.6.2 Floating-Point Add

Regardless of their signs, both operands' coefficients are extended to 94 bits in length, not including sign, by adding 47 zeros to the right of their binary points.

The exponents of the two operands are compared and the 94-bit coefficient of the operand having the smaller exponent is effectively shifted right one bit and its exponent increased by one, successively until

(continued)

3.1.4.6.2 (Cont.)

the two exponents are equal. The sign of the shifted coefficient is extended from the left to the right during the shift. Negative coefficients approach a minus one and positive coefficients approach zero as they are shifted.

The add is a 94-bit operation, not including sign. Right normalization takes place, if necessary. The coefficient for the U result is the left-most 47 bits and the coefficient for the L result is the right-most 47 bits of the 94-bit result.

The exponent for the U result is equal to the larger of the two operand exponents. Right-normalization will increase this value by one, if it occurred.

The exponent for the L result is 47_{10} less than the U result's exponent for all cases except three:

a. Right-normalization causes the U exponent to overflow; the U result is set to indefinite; the L exponent will be $6FD1_{16}$ (59_{16} in the 32-bit case).

b. If the U result's exponent minus 47_{10} causes exponent underflow, machine zero is stored as the L result.

c. If either or both operands were indefinite, the U and L results are indefinite.

3.1.4.6.3 Floating-Point Subtract [A12.0]

The floating-point subtract operation is performed by complementing the coefficient of the subtrahend and performing a floating-point addition operation. The complementation is a 48-bit, two's complement operation and is performed before the operands are extended to 94 bits.

The hardware used for Floating Add or Subtract operations has an extra (or extended) coefficient sign bit. This means that the complementation of an 8000 coefficient is handled without the right shift of one and increase of the exponent by one as used elsewhere. This will cause a result (although not mathematically incorrect) which may differ from the result obtained when a right shift of one with increase of one is used, when the following conditions are met:

1. The operand of the pair having the large exponent (OR either of the two operands if their exponents are equal) must have a coefficient of 8000 ---

2. This operation must require this same operand to be complemented due to
   a. being the subtrahend in a subtract operation OR
   b. sign control in either a subtract or an add operation ---

3. The "other" operand must have a negative coefficient.

(continued)

R A D L

3.1.4.6.3 (Cont.)

Example I A - B

A 60 F F F 0 0 0
B 64 8 0 0 0 0 0

		CDC FMP (Extra Sign Bit)	Instruction Specification
Complement B	B	-64 (1) 8 0 0 0 0 0	64 8 0 0 0 0 0
	$\overline{B}$	->64 (0) 8 0 0 0 0 0	65 4 0 0 0 0 0
Align operand with smaller exponent		-60 (1) F F F 0 0 0	-60 F F F 0 0 0
		->64 (1) F F F F 0 0	->65 F F F F 8 0
Add A plus complement of B	A	64 (1) F F F F 0 0	65 F F F F 8 0
	$+\overline{B}$	64 (0) 8 0 0 0 0 0	65 4 0 0 0 0 0
		6 (0) 7 F F F 0 0	65 3 F F F 8 0
		64 7 F F F 0 0	65 3 F F F 8 0

Example II A - B

A 50 F F F 0 0 0
B 6F 8 0 0 0 0 0

		CDC FMP (Extra Sign Bit)	Instruction Specification
Complement B	B	-6F (1) 8 0 0 0 0 0	6F 8 0 0 0 0 0
	$\overline{B}$	->6F (0) 8 0 0 0 0 0	70 4 0 0 0 0 0
Align operand with smaller exponent		-50 (1) F F F 0 0 0	50 F F F 0 0 0
		->6F (1) F F F F F F	70 F F F F F F
Add A plus complement of B	A	6F (1) F F F F F F	70 F F F F F F
	$+\overline{B}$	6F (0) 8 0 0 0 0 0	70 4 0 0 0 0 0
		6F (0) 7 F F F F F	70 3 F F F F F

(continued)

3.1.4.6.3 (Cont.)

If this operation is a Subtract Upper, the specified result is indefinite (with the appropriate data flags) while the CDC FMP result did not overflow. If this operation were a Subtract Normalized, note the following:

	CDC FMP	Instruction Specification
Result of Subtract Upper	6F (0) 7 F F F F F	70 3 F F F F F
Normalize the Upper Result shifting zeros in from the right	6F 7 F F F F F	6F 7 F F F F E

Note that the subtract operation is not always commutative. In other words it is not always true that (A-B) = -(B-A). This characteristic will be observed if the following is true of A and B:

a. The exponents of A and B are not equal.

b. "1" bits exist in any of the right most bit positions of the coefficient which will be shifted off the right during alignment of the smaller exponent.

Example of (A-B) <> -(B-A):

```
A =  0104   6FCB   807E   89F2
B =  0100   6FAC   3F5D   A5FA <--
                                 |
                                 |
```

These two 1 bits will be shifted off during exponent alignment.

(continued)

RADL

3.1.4.6.3 (Cont.)

Complement B:

-B =	0100	9053	C0A2	5A06	

Align B:

-B =	0104	F905	3C0A	25A0	6

A-B:

A =	0104	6FCB	807E	89F2	
-B =	0104	F905	3C0A	25A0	6
	0104	68D0	BC88	AF92	6

A-B =	0104	68D0	BC88	AF92	

Align B:

B =	0104	06FA	C3F5	DA5F	A

Complement A:

-A =	0104	9034	7F81	760E	

-(B-A):

B =	0104	06FA	C3F5	DA5F	A
-A =	0104	9034	7F81	760E	
	0104	972F	4377	506D	A
-(B-A)=	0104	68D0	BC88	AF93	

This differs from A-B in the last bit position.

3.1.4.6.4 Results of the Floating-Point Multiply Instruction

When two floating-point numbers are multiplied, the lower result retains the 47 least significant product bits generated. The sign bit of the lower result is always set to zero and the exponent of the lower result is the sum of the two source operands' exponents with the exceptions listed below:

The upper result retains the 47 product bits immediately to the left of the bits retained by the lower product. The sign of the upper product's coefficient follows the normal rules of algebra. The exponent of the upper result is the sum of the two source operands' exponents plus 47 with the following exceptions:

a. The sum of the source operands' exponents (plus 47_{10}, if upper result) exceed $6FFF_{16}$ for which case the result exponent is set to indefinite.

b. The sum of the source operands' exponents (plus 47_{10}, if upper result) is less than 9000_{16} for which case the result exponent is set to machine zero.

c. Either or both operands are indefinite for which case the result exponent is set to indefinite.

d. Neither operand is indefinite but either or both operands are machine zero, for which case the result exponent is set to machine zero.

If either operand has a coefficient of 8000 0000 0000 and an exponent of X, the operand will be treated as though its coefficient were C000 0000 0000 and its exponent were X+1.

3.1.4.6.5 The Floating-Point Divide Instruction

The quotient from the divide operation is the result of dividing the prenormalized, integer coefficient of the divisor into the integer coefficient of the dividend generating a 47-bit quotient (23-bit quotient for 32-bit divide). If either operand has a

(continued)

3.1.4.6.5 (Cont.)

coefficient of 8000 0000 0000, the operand will be handled as though its coefficient were C000 0000 0000 and its exponent increased by one. When the divide hardware normalizes the divisor coefficient, the number of places shifted left is added to the exponent of the quotient as defined below.

The exponent of the result will be given by the following equation:

$$\text{Exponent of Quotient} = (\text{Exponent of Dividend}) - (\text{Exponent of Divisor}) - (46_{10} - NC)$$

where NC is the number of places shifted left to prenormalize the divisor. For the 32-bit divide operation 22_{10} is subtracted rather than 46_{10}.

The right-most bit of the quotient is neither rounded nor adjusted. The remainder is not retained. The sign of the quotient's coefficient follows the normal rules of algebra.

3.1.4.6.6 Normalized Upper Results

The normalized add and subtract instructions generate an intermediate result identical to the final result of the Add U and the Subtract U instructions. Normalization of the intermediate, 48-bit result then takes place as follows:

The 48-bit coefficient is shifted left one bit and its exponent is decreased by one, successively, until the sign bit and the bit immediately to the right of the sign bit are different. During this shift, zeros are attached to the right end of the 48-bit coefficient. If reducing the exponent by one causes exponent underflow, the result of the normalization operation is defined as machine zero.

3.1.4.6.7 (N/A)

3.1.4.7 (N/A)

3.1.4.8 (N/A)

3.1.4.9 (N/A)

3.1.4.10 (N/A)

3.1.4.11 Operand Size Definitions

The following definitions are implied throughout the specification.

Word - A 64-bit quantity, the address of the left-most bit always being a multiple of 64_{10}.

Half-word - A 32-bit quantity, the address of the left-most bit always being a multiple of 32_{10}.

Byte - An 8-bit quantity, the address of the left-most bit always being a multiple of 8_{10}.

Digit - A 4-bit binary coded decimal number or sign. One digit per byte in zoned format and two digits per byte in packed BCD format.

Sword - 512 bits (or 8 64-bit words).

3.1.5 Item Count (field lengths, offsets, indices, etc.)

All field lengths, offsets, indices, shift counts, etc., are item counts which specify a number of bits, digits, bytes, half-words or words.

Where an item count other than an index is contained in a 48-bit field, there shall be at least 32 consecutive and identical sign bits. Sign bits must always be extended to the left to fill the 16-bit or 48-bit field containing it.
The item count unit is specified by the instruction title line code (see arrow).

Example

```
                          |
                          V
3.2.1.67        42    4   32   RG ADD N; (R)+(S) TO (T)
```

The 32 indicates that field lengths and indices are expressed in 32-bit half-words. Any deviation from this method of specifying the units for the various item counts would be indicated in the instruction description or in the description of the instruction type. The instruction type refers to RG (register), SM (stream), etc.

An index may be either positive or negative in sign. The maximum magnitude of an index is a function of its usage. The index is shifted to the left end-off zero/three/five/six places before the addition to the base address when the unit for the index is bits/bytes/half-words/words. Digits are not used as a unit for indices.

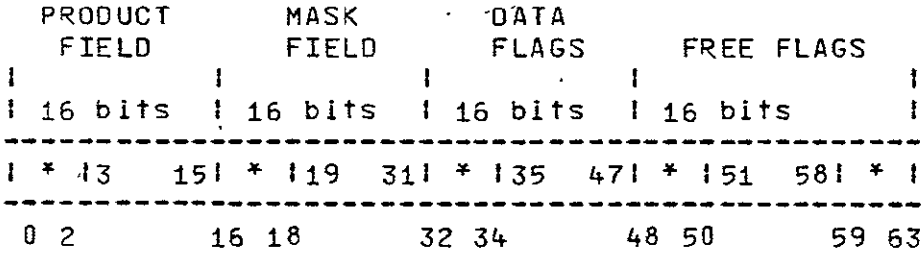
A field length must be positive in sign and have a magnitude of less than 2^{16}; the use of a negative field length causes that length to become strictly undefined. Offsets are subtracted from the field length in stream instructions, but note that for a negative offset, this amounts to increasing the length specification since subtracting a negative quantity is addition.

3.1.6 Data Flag Branch Register [A7.0]

3.1.6.1 General Description

The data flag register is designed to give the programmer an automatic branch to a special routine for certain operands, results, conditions, etc., without his having to pay the time penalty of explicitly checking these conditions in his program. If a condition which has been previously selected to cause an automatic branch occurs during an instruction, the instruction is completed, the address of the next instruction which would have been executed is stored into the address portion of register 01 and a branch is made to the address contained in register 02. The state of the data flags in the invisible package is defined only if the program was interrupted between instructions.

3.1.6.2 Register Description



*Bits 0 through 2, 16 through 18, 32 through 34, 48 through 50, and 59 through 63 of the data flag register are undefined. Any attempt to sample, set or clear these bits is meaningless and the result of any instruction trying to do so is undefined.

An additional register providing bits 64 through 127 has been added for the expanded vector capabilities. Bit assignments, and location in the invisible package, have not been made as yet.

3.1.6.2.1 Data Flag Bits

Data flags 35-47 indicate conditions that have occurred. Bits 35-47 are cleared only by the Data Flag Register Bit Branch and Alter, and the Data Flag Register Load/Store instructions.

3.1.6.2.2 Mask Bits

A mask bit is associated with each of the data flags. The mask bits have the function of selecting the conditions for which the programmer wishes an automatic data flag branch.

It is important to note that the associated mask bit need NOT be set in order to set a data flag bit. The mask function is solely one of enabling a particular data flag to cause a bit to set in the product field. The order in which the mask bit and its associated data flag bit are set is immaterial, as the result is the same; that is, their associated product bit is set.

3.1.6.2.3 Product Bits

Each product bit is the dynamic logical product of a data flag bit and its associated mask bit. Data flag branches are performed when there is at least one <u>one</u> in the product register and the data flag branch enable bit is set.

3.1.6.2.4 Data Flag Branch Enable Bit

The data flag branch enable bit, bit 52, must be set for an automatic data flag branch (DFB) to occur. Bit 52 is automatically cleared by the hardware when a DFB takes place. It must be reset with a Data Flag Register Bit Branch and Alter or a Data Flag Register Load/Store instruction to re-enable the DFB.

3.1.6.2.5 Data Flag Register Bit Assignments

```
Product Bit
|
|  Mask Bit
|  |
|  | -Data Bit
|  | |
V  V V
3-19-35
```

Soft Interrupt. Monitor software can set bit 35 of a job's Data Flag Branch register while the register is stored in the job's invisible package. If, after exchanging back to job mode, bit 35 and its corresponding mask bit (bit 19) are set, a normal data flag branch occurs following completion of the current instruction.

4-20-36

Job Interval Timer

5-21-37

N/A

(continued)

3.1.6.2.5 (Cont.)

6-22-38

N/A

7-23-39

The binary result exceeds the range of $\pm (2^{47}-1).10$.

8-24-40

Bit 40 is the inclusive OR of bits 37, 38 and 39. Bit 24 masks bit 40. Bit 8 is the logical product of bits 24 and 40.

(continued)

3.1.6.2.5 (Cont.)

9-25-41

Floating-point divide fault: The divisor has an all zero coefficient or the divisor as read from the register file or from central storage is machine zero. If the divisor and/or the dividend is indefinite, no divide fault exists. If a divisor causes a divide fault, the quotient is set to indefinite. The exponent overflow and result machine zero data faults are not set by a divide whose divisor caused a divide fault.

10-26-42

Exponent overflow: The exponent of the result is larger than $6FFF_{16}$ ($6F_{16}$ for 32-bit arithmetic). Results are not checked for exponent overflow until after the exponent adjustment for normalization or significance has taken place. In the adjust exponent instructions, if a left shift exceeds the number of places required for normalization, this data flag is set. Exponent overflow causes the result to be set to indefinite; therefore, the indefinite flag will always be set on an exponent overflow. This exponent overflow data flag is not set if either source operand from central storage or the register file is indefinite or by a divide instruction whose divisor causes a divide fault.

11-27-43

Result Machine Zero: The exponent of the result returned to Main Memory or to the Register File is less than 9000_{16} (90_{16} for 32-bit arithmetic). Result Machine Zero may be caused by exponent underflow or by one or more of the input operands being machine zero. The Result Machine Zero data flag bit is not set by a divide whose divisor causes a divide fault.

(continued)

3.1.6.2.5 (Cont.)

12-28-44

Bit 44 is the inclusive OR of bits 41, 42 and 43. Bit 28 masks bit 44. Bit 12 is the logical product of bits 28 and 44.

13-29-45

A negative source operand was encountered in a square root instruction. The square root of the absolute value of the operand is formed; and the two's complement of this square root is stored as the result.

14-30-46

An indefinite result was placed into central storage or into the Register File.... or .... either or both operands of a floating-point compare were indefinite.

An indefinite result may be caused by one or both operands of a floating-point arithmetic operation being indefinite or by the occurrence of either a divide fault or an exponent overflow.

15-31-47

Breakpoint: See section 3.2.1.5.

3.1.6.2.6 Free Data Flags

Bit 51 is the dynamic inclusive OR of the product field. This bit is set if any of bits 4 through 15 are set. Bit 51 cannot be cleared directly; bits 4 through 15 must be cleared to accomplish this.

Bit 52 is the data flag branch enable bit. If bit 52 is a one and bit 51 becomes a one (or vice versa) a data flag branch occurs at the end of the current instruction. See 3.1.6.3 for additional information. Bit 52 is automatically cleared by the execution of a data flag branch.

(continued)

3.1.6.2.6 (Cont.)

Bits 53, 54 and 55
There are no product or mask bits associated with bits 53, 54, and 55. Bits 53, 54, and 55 are cleared out automatically during the initial phases of the instructions (unless the instruction is a no op -- see Section 3.1.3) which may set any of them. Thus, if pertinent, these bits must be sampled before executing another instruction which would clear their previous state. The setting of bits 53, 54, and 55 does not cause a data flag branch.

Bit 56 - A CPU gate associated with the Maintenance Station monitoring counters (See Functional Computer Specification listed in Section 2.0).

Bit 57 - A CPU gate associated with the Maintenance Station monitoring counters (See Functional Computer Specification listed in Section 2.0).

Bit 58 - N/A

(continued)

3.1.6.2.6 (Cont.)

OP CODE	DATA FLAG BITS									53 54	OP CODE	DATA FLAG BITS									53 54
	V	37	38	39	41	42	43	45	46	47 55		V	37	38	39	41	42	43	45	46	47 55
00											20										
01											21										
02											22										
03											23										
04									X		24										
05											25										
06											26										
07											27										
08											28										
09											29										
0A											2A										
0B											2B										
0C											2C										
0D											2D										
0E											2E										
0F											2F										
10			X								30										
11											31										
12											32										
13											33										
14											34										
15											35										
16											36										
17											37										
18											38										
19											39										
1A											3A										
1B											3B										
1C											3C										
1D											3D										
1E											3E										
1F											3F										

(continued)

3.1.6.2.6 (Cont.)

OP CODE V	DATA FLAG BITS 37	38	39	41	42	43	45	46	47	53 54 55
40				X	X		X			
41				X	X		X			
42				X	X		X			
43										
44				X	X		X			
45				X	X		X			
46				X	X		X			
47										
48				X	X		X			
49				X	X		X			
4A										
4B				X	X		X			
4C			X	X	X		X			
4D										
4E										
4F			X	X	X		X			
50							X			
51							X			
52							X			
53					X	X	X			
54				X	X		X			
55				X			X			
56										
57										
58										
59				X	X		X			
5A										
5B										
5C					X		X			
5D					X		X			
5E										
5F										

OP CODE V	DATA FLAG BITS 37	38	39	41	42	43	45	46	47	53 54 55
60				X	X		X			
61				X	X		X			
62				X	X		X			
63										
64				X	X		X			
65				X	X		X			
66				X	X		X			
67										
68				X	X		X			
69				X	X		X			
6A										
6B				X	X		X			
6C			X	X	X		X			
6D										
6E										
6F			X	X	X		X			
70							X			
71							X			
72							X			
73					X	X	X			
74				X	X		X			
75				X			X			
76				X	X		X			
77				X	X		X			
78										
79				X	X		X			
7A										
7B										
7C										
7D										
7E										
7F										

(continued)

RADL

3.1.6.2.6 (Cont.)

OP COD	DATA FLAG BITS 37 38 39 41 42 43 45 46 47 53 54 55	OP CODE	DATA FLAG BITS 37 38 39 41 42 43 45 46 47 53 54 55
80		A0	
81		A1	
82		A2	
83		A3	
84		A4	
85		A5	
86		A6	
87		A7	
88		A8	
89		A9	
8A		AA	
8B		AB	
8C		AC	
8D		AD	
8E		AE	
8F		AF	
90		B0	
91		B1	
92		B2	
93		B3	
94		B4	
95		B5	
96		B6	
97		B7	
98		B8	
99		B9	
9A		BA	
9B		BB	
9C		BC	
9D		BD	
9E		BE	
9F		BF	

(continued)

R A D L

3.1.6.2.6 (Cont.)

OP CODE	DATA FLAG BITS V	37	38	39	41	42	43	45	46	47	53 54 55
C0											
C1											
C2											
C3											
C4											
C5											
C6											
C7											
C8											
C9											
CA											
CB											
CC											
CD											
CE											
CF											
D0											
D1											
D2											
D3											
D4											
D5											
D6											
D7											
D8											
D9											
DA											
DB											
DC											
DD											
DE											
DF											
E0											
E1											
E2											
E3											
E4											
E5											
E6											
E7											
E8											
E9											
EA											
EB											
EC											
ED											
EE											
EF											
F0											
F1											
F2											
F3											
F4											
F5											
F6											
F7											
F8											
F9											
FA											
FB											
FC											
FD											
FE											
FF											

3.1.6.3 Data Flag Branch (DFB) [A7.0]

If a bit in the mask field is set and its associated masked data flag bit is set, the associated bit in the product field becomes a one. Bit 51 in the free flag field also becomes a one since it is the dynamic inclusive OR of bits 4 through 15 of the product field.

If bit 51 is a one from above and if bit 52 is also set (this is the DFB enable bit), an automatic DFB occurs. The DFB takes place sometime following the termination of the instruction which caused the DFB condition to exist. The execution of the DFB sets the bit address of the next instruction into the right-most 48 bits of register 01 and a branch is made to the bit address contained in the right-most 48 bits of register 92. The DFB enable bit in the flag mask register (bit 52) is automatically cleared at this time. The left-most 16 bits of register 01 are cleared to zero by a DFB.

Programmer Note:

DFB's are disabled when bit 52 is cleared. But if bit 52 is reset before eliminating all the DFB conditions, another DFB will occur which will change the return address in register 01 and the machine may wind up in a "tight loop" if proper caution is not taken. Sampling bit 51 for a zero before setting bit 52 will prevent this situation for all cases except those involving the job interval timer. When using the job interval timer, it should be remembered that the setting of bit 36 in the DFR occurs asynchronously with respect to instruction execution once the job interval timer is loaded. Thus the time may set bit 36 after the check of bit 51 and before the branch to the contents of register 01. One method of handling this situation is to examine the contents of register 01 upon entering the routine for handling data flag branches. If register 01 indicates that the branch occurred outside the DFB routine, then register 01 could be copied to a temporary location.

(continued)

3.1.6.3 (Cont.)

If register 01 indicated that the branch had occurred within the OFB routine, then register 01 would not be copied to the temporary location. At the conclusion of the OFB routine, a branch would always be taken to the contents of the temporary location.

A simpler method is to combine the setting of bit 52 and the branch to the contents of register 01 into a single 33 instruction (33603401).

3.1.7 Register File

For register operations, the 8-bit instruction designators directly address the 256 $_{10}$ registers of the Register File. During program execution (monitor or job), these registers reside in the Register File. When an exchange operation occurs, the registers are stored into 256 $_{10}$ memory locations beginning at bit address zero if in monitor mode and bit address 4000 $_{16}$ if in job mode. The registers may not be referenced as memory by their associated monitor or job program. The only exceptions to this rule are the B7 and BA instructions with G-bit 7 set. (The B7 and BA instructions are illegal in the CDC FMP).

(continued)

3.1.7 (Cont.)

Figure 1 shows a map of the Register File and the relationship between the register, its storage address for monitor mode and its 8-bit designator. The number on the right represents the bit address and the number on the left is the value of the 8-bit designator for the 64-bit register case. The number inside the register represents the value of the 8-bit designator for the 32-bit operand case. Note that any reference to 32-bit register one is undefined.

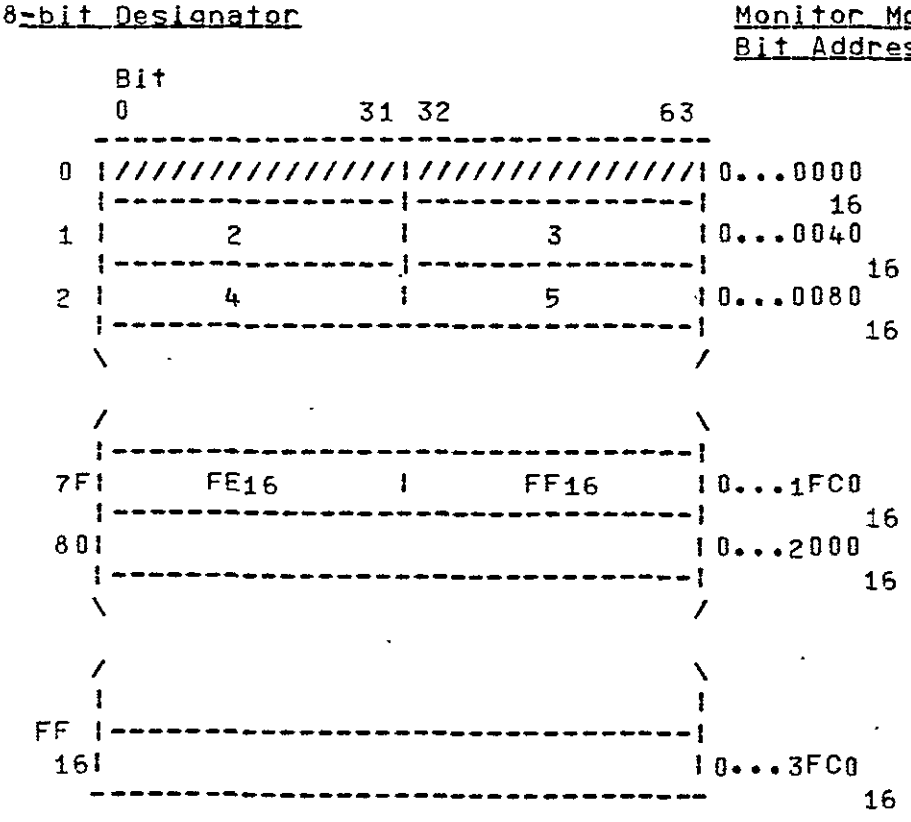


Figure 1. Register File

(continued)

3.1.7 (Cont.)

Register File Restrictions

A. Register Zero (Job or Monitor Mode)

1. During an exchange operation the contents of the trace register and the appropriate memory location for register zero are exchanged (swapped).

Monitor to Job:

	Before Exchange	After Exchange
Absolute Address Zero	A	C
Trace Register	C	A

Job to Monitor:

	Before Exchange	After Exchange
Absolute Address Zero	A	A
Trace Register	C	A

(continued)

3.1.7 (Cont.)

During a 7D (Swap) instruction involving register zero as part of the register field, note a required peculiarity. Although the current contents of the trace register are sent to the appropriate memory location for register zero, the current contents of the trace register are not altered.

	Contents Before 7D	Contents After 7D
Memory location for register zero	A	B
Trace register	B	B

2. Register zero when referenced by a designator will provide machine zero as an operand except when used as a source register for a base address or other description for a stream instruction, in which case register zero will appear to contain 64-zero bits. The use of a zero address may cause the instruction to be treated as an illegal instruction as defined in Section 3.1.10. The use of a zero field length may cause the instruction to become undefined such as the 3B instruction. If register zero is specified as the destination register, the instruction typically performs normally with data flags being set, if warranted, but no data is stored. Some instructions become undefined if register zero is specified as a destination register.

The following tables are intended to define what operand is obtained when register zero is specified for a source operand. To simplify this chart, specifying of register zero as a

(continued)

3.1.7 (Cont.)

destination register has been ignored. A blank in the chart indicates where it is either not possible to specify register zero or it may only be specified as a destination register. The designators R, S, T, G, X, A, Y, B, Z and C are used for convenience although they do not apply to all instructions. Utilization of the following symbols is made.

Symbol	Result When Register Zero is Referenced for an Operand
M	Machine zero is provided. 8000 0000 0000 0000 64-bit mode 8000 0000 $_{16}$ 32-bit mode
A	All zero is provided.
Z	All zero in the used portion. In this instance the left-most bit is not used thus machine zero and all zeros are indistinguishable.
N	Instruction performs as a no op.
C	No control vector is used.
O	A mask of all ones is provided.

(continued)

3.1.7 (Cont.)

Op Code	Instruction Designator R	S	T
04	Z		
09		Z	Z
0A	Z		
0E	Z	Z	
10	M		
11	Z		
12	Z	Z	
13	Z	Z	Z

Op Code	Instruction Designator R	S	T
2B	M	Z	
2C	M	M	
2D	M	M	
2E	M	M	
2F		Z	Z
30	M		
31	Z	Z	Z
32		Z	Z
33			Z
34	M	Z	
35	Z	Z	Z
36		Z	Z
37			
38	M		
3A	Z		
3B	Z		
3C	Z	Z	
3D	Z	Z	
3F	Z		

(continued)

3.1.7 (Cont.)

Op Code	Instruction Designators R	S	T	Op Code	Instruction Designators R	S	T
40	M	M		60	M	M	
41	M	M		61	M	M	
42	M	M		62	M	M	
				63	M	Z	
44	M	M		64	M	M	
45	M	M		65	M	M	
46	M	M		66	M	M	
				67	M	Z	
48	M	M		68	M	M	
49	M	M		69	M	M	
4B	M	M		6B	M	M	
4C	M	M		6C	M	M	
4D				6D	M	Z	
4E	Z			6E	M	Z	
4F	M	M		6F	M	M	
50	M			70	M		
51	M			71	M		
52	M			72	M		
53	M			73	M		
54	M	Z		74	M	Z	
55	M	M		75	M	Z	
56	Z	Z	Z	76	M		
				77	M		
58	M			78	M		
59	M			79	M		
5A	M			7A	M		
5B	Z	Z		7B	Z	Z	
5C	M			7C	M		
5D	M			7D	A	*	A
5E	Z	Z		7E	Z	Z	
5F	Z	Z	M	7F	Z	Z	M

*See Section 3.1.7.A.1

(continued)

3.1.7 (Cont.)

Op Code	Instruction Designators G	X	A	Y	B	Z	C
B0		Z	M	Z	Z	Z	
B1		Z	M	Z	Z	Z	
B2		Z	M	Z	Z	Z	
B3		Z	M	Z	Z	Z	
B4		Z	M	Z	Z	Z	
B5		Z	M	Z	Z	Z	
B6	Z						
9D							
9E							
9F							

Op Code	Instruction Designators G	X	A	Y	B	Z	C
B0		Z	M	Z	Z	Z	
B1		Z	M	Z	Z	Z	
B2		Z	M	Z	Z	Z	
B3		Z	M	Z	Z	Z	
B4		Z	M	Z	Z	Z	
B5		Z	M	Z	Z	Z	
B6	Z						

(continued)

3.1.7 (Cont.)

Op Code	G	X	Instruction Designators A	Y	B	Z	C
CE		Z					

Op Code	G	X	Instruction Designators A	Y	B	Z	C
					1		

(continued)

3.1.7 (Cont.)

B. 64-bit registers one and two (32-bit registers 2 through 5)

If data flag branches are being used, 64-bit registers one and two must be reserved exclusively for that use. Register one is the data flag branch exit address and register two holds the data flag branch entry address.

C. Monitor's 64-bit registers 0-F (32-bit registers $0-1F_{16}$)

Registers zero, one and two have the restrictions listed in A and B above. Registers 3 through 7 are used for the illegal instruction, exit force, and external interrupt entry points.

D. 32-bit register one (right-most half of 64-bit register 0)

Any reference to 32-bit register one is undefined.

3.1.8 Real-Time Counters

3.1.8.1 Free Running Clock

This clock consists of a free-running 47-bit counter and a positive sign bit for a total of 48 bits. It can be stored into register T using a "Transmit Real-Time Clock to T" (39) instruction. This counter increments at a one MHz rate.

3.1.8.2 Monitor Interval Timer

The monitor interval timer is a 24-bit timer that decrements at a one MHz rate.

This timer can be loaded from register R using the "Transmit (R) to Monitor Interval Timer" (0A) instruction, when the computer is in monitor mode. The timer can be activated by loading it with anything but all zeros. Once it is activated, it will decrement until it reaches zero or is deactivated. When the timer is decremented to zero, it will cause an external interrupt on channel 16 which must be processed like any other external interrupt.

The timer is deactivated by the following methods:

1. Master clear
2. Loading with all zeros
3. Decremented to all zeros (when it is decremented to all zeros and caused an external interrupt, it will be inactive until loaded with some value other than zero).

3.1.8.3 Job Interval Timer

The job interval timer is a 24-bit counter decrementing at a one MHz rate.

This clock can be loaded (in job mode) only from register R using a 3A (Transmit R to Job Interval Timer) instruction. Once loaded, the timer continues to decrement until either an exchange to monitor mode occurs, the timer decrements to zero, or the timer is loaded with a value of zero. If an exchange to monitor mode occurs, the decrementing of the job interval timer is stopped and the current contents of the timer are stored in the invisible package. When the execution of that job is resumed, the job interval timer is loaded from the invisible package and resumes decrementing.

When the timer decrements to zero, bit 36 of the data flag branch register will be set. Thus, if the corresponding mask bit is set, a data flag branch would then occur during the next RNI.

The timer may be deactivated by loading it with a value of zero. This does not cause bit 36 of the data flag branch register to be set. Master clear will also deactivate the job interval timer.

The timer is deactivated by the following methods:

1. Master clear

2. Loading with a value of zero

3. Decrementing to zero

The contents of the job interval timer may be sampled by use of the 37 instruction (Transmit Job Interval Timer to T). This does not deactivate the counter.

3.1.9 N/A

3.1.10 Exchange Operations and Invisible Package [A6.0]

The purpose of the exchange is to change the prime role of the CPU from monitor mode to job mode or from job mode to monitor mode.

The exchange operation from monitor to a job is always accomplished with an exit force instruction. This causes the contents of the invisible package to be loaded into the appropriate registers; the mode to be changed from monitor to job enabling interrupts; and execution to begin as specified by the invisible package. Note that this may be the restarting of a previously interrupted program.

The Exit Force instruction and the channel interrupt are the two normal ways of getting from a job in job mode to the monitor program in monitor mode. Attempting to execute a monitor-type instruction in job mode or by attempting to execute an undefined op-code comprise the third way into the monitor. Except for the starting point in the monitor program, the operation performed in getting to the monitor are identical for the three. Sufficient information to restart this job is stored into the invisible package and the mode is changed from job to monitor. The monitor program is executed starting at the absolute address contained in the right-most 48 bits of the monitor's register 3, 5, or 6.

(continued)

3.1.10 (Cont.)

Method of getting to the Monitor	Monitor register, the contents of which is used to set P
1. Attempt to perform an illegal instruction or a monitor-type instruction in job mode	Register 3
2. Attempt to perform an illegal instruction in monitor mode	Register 4
3. Exit force	Register 5
4. External interrupt	Register 6

The right-most ten bits of the absolute starting address of the invisible package must be zeros.

The monitor must set up an invisible package for each job. There is NO invisible package for the monitor program itself.

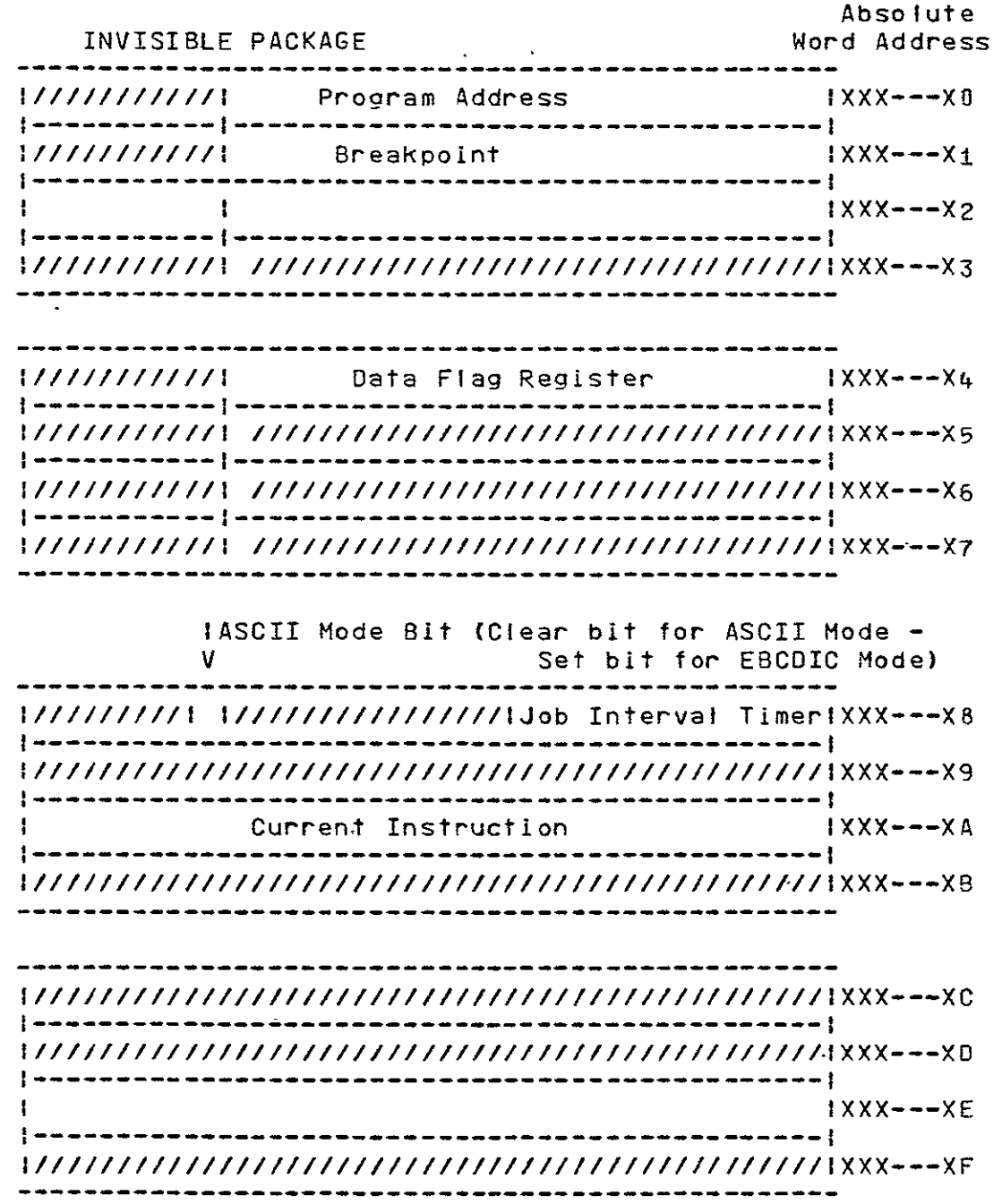
To start a job initially, the monitor must clear the entire invisible package area except for the program address areas.

For a more detailed description of the exchange operation, see the applicable computer specification as listed in Section 2.0.

(continued)

R A D L

3.1.10 (Cont.)



The computer returns the information in the non-crosshatched areas except as noted in Appendix A6.0. For specific detail in the cross-hatched areas see the applicable machine specification as listed in Section 2.0.

3.2 Performance Characteristics

3.2.1 Instruction Descriptions

The instruction titles (3.2.1.1 - 3.2.1.256) are written in the following format:

3.2.1.XXX AA B CC DD NAME OF INSTRUCTION [AX]

where AA = the function code (00-FF$_{16}$)

B = the format types, 1-E

CC = the number of bits in the operand

1	single bit
32	half-words
64	words
E	either 32 or 64-bit
B	both 32 and 64-bit
NA	operand size not applicable

DD = the instruction type

	Blank Undefined
BR	Branch
IN	Index
MN	Monitor
NT	Non-Typical
RG	Register
SM	Stream

[AX] = The section in the Appendix which gives further information.

3.2.1.1 00 4 NA MN IDLE

When in monitor mode, enable the external interrupts and idle until an external interrupt occurs. The R, S and T designators are undefined and must be set to zero.

3.2.1.2 01 4 64 NT TRANSMIT (R) TO BACKING STORE MAP REGISTER AND CURRENT BACKING STORE MAP REGISTER TO (T); SET AND CLEAR BUSY FLAGS PER (S)

The Backing Store contains 8192 blocks, each of 32,768 64-bit words. All or any portion of this Backing Store can be assigned to the user presently residing in the CPU by setting the Block Base Address (BBA) and Block Field Length (BFL) in the backing store map register. When in job mode, all backing store addresses sent to the Swap Unit have the BBA added to their values to form an absolute backing store address. All monitor mode and I/O references are made as absolute references without the addition of the BBA.

The BBA is contained in bits 48 through 63 of register R, while the BFL is contained in bits 32 through 47 of register R. Register T at the completion of this insruction contains the current values of BBA and BFL transmitted from the backing store map register in bits 32 through 63, while the upper bits (0 through 31) contain the block number and number of contiguous blocks that have been set busy in the Backing Store (as a result of an I/O operation, SWAP operation or monitor mode force busy operation). Bits 0 through 15 contain the number of contiguous blocks while bits 16 through 31 contain the block number of the first block found busy in the Backing Store, beginning at the block number found in bits 16 through 31 of register R.

If the contents of bits 0 through 15 or bits 32 through 47 of register S are non-zero the instruction also force-sets or force-clears groups of block busy flags in the Backing Store as follows:

- Bits 0-15 = number of blocks to be forced busy in the Backing Store
- Bits 16-31 = block number of first block of group to be set busy

(continued)

3.2.1.2 (Cont.)

- o Bits 32-47 = number of blocks to be forced not busy in the Backing Store
- o Bits 48-63 = block number of first block of group to be forced not busy

If a force busy and force not busy is attempted on the same block, or blocks, the result is undefined.

Note that the busy flags for each block can be set or cleared by the I/O channel and monitor by software, and by the Swap Unit during BSWAP transfers. If a BSWAP operation from job mode specifies a busy block, the BSWAP operation terminates, setting data flag 96.

The full execution of this instruction as described is possible only when it is executed in monitor mode. If the instruction is executed in job mode, only the block busy information is transferred to bits 0 through 31 of register T with bits 16 through 31 of register R specifying the beginning block number in finding the first busy block.

3.2.1.3 02 4 64 MN TRANSMIT (R) TO CHANNEL (S) AND CHANNEL (S) TO (T)

Register S contains the number of an I/O channel. The contents of 64-bit register R are transmitted to the specified I/O channel (between 0 and 15), at the same time the specified I/O channel transmits a 64-bit quantity to be stored in register T. The data being exchanged consists of control information passed between the monitor mode program and the I/O channel intelligent processor (PDC). The meaning of any combination of bits in these exchanged control words is solely defined by the software protocols established for the monitor and the PDC.

3.2.1.4 03 ILLEGAL

3.2.1.5 04 4 64 NT BREAKPOINT-MAINTENANCE

The breakpoint instruction transfers R to the breakpoint register. The breakpoint register is used as a maintenance and program debugging aid.

(continued)

3.2.1.5 (Cont.)

Note: Breakpoint will not be sensed on any scalar memory references in the CDC FMP.

	Usage Bits	Breakpoint Address	
0	8 9 15	16 58	59 63

Bits 0-8 and 59-63 are not used.

The breakpoint address is compared with various addresses such as the current instruction address, READ 1 and READ 2 operand addresses, etc. If the breakpoint address matches one of these addresses and the proper usage bit is set, bit 47 of the data flag branch register is set indicating a breakpoint. Any combination of usage bit is permissible; therefore the breakpoint address can be checked against any or all of the addresses listed below. The breakpoint register is part of the invisible package of a job.

Breakpoint Usage Bits

Bits 9-15 are breakpoint usage bits where if:

a. Bit 9 is set, breakpoint on half-word contents of the program address register (P) just after the execution of the instruction at that location.

b. Bit 10 is set, breakpoint on the READ 1 operand address for stream, or the read operand on random addressing instructions.

c. Bit 11 is set, breakpoint on the READ 2 operand address for a stream instruction.

d. Bit 12 is set, breakpoint on the WRITE 1 address for a stream instruction or the write operand on a random addressing instruction.

e. Bit 13 is set, breakpoint on the READ 3 control vector or operand address (mask) for a stream instruction.

f. Bit 14 is set, breakpoint on the READ 1 order vector address.

(continued)

3.2.1.5 (Cont.)

g. Bit 15 is set, breakpoint on the READ 2 order vector address.

Breakpoint Compares

1. When in job mode, addresses are compared with breakpoint.

2. When in monitor mode, absolute addresses are compared with breakpoint. Since the monitor program does not have an invisible package, the breakpoint register must be set up each time the monitor program is entered. The breakpoint register is automatically cleared to zero during the exchange to the monitor.

3. Program address compares are made on half-word boundaries, and all other compares are made on sword boundaries.

Data flag: bit 47

3.2.1.6 05 ILLEGAL

3.2.1.7 06 7 NA MN FAULT TEST - MAINTENANCE [A11.0]

This instruction is used to complement checkword bits on the scalar write bus in order that the read SECDED circuitry may be checked out. It can also be used to disable the error correction circuitry on all read buses. This allows data to be passed through the SECDED hardware without any correction taking place.

This instruction is always enabled during monitor mode. In job mode it becomes a no op unless bit 13 of word 8 in the job's invisible package is set.

The modes are set up by executing this instruction with a "1" in the appropriate R designator bit and are cleared by executing the instruction with a "0" in the same bit location.

The R designator bits are defined below:

R DESIGNATOR BIT

8	Disable error correction on all Read buses.
9-15	Checkword bits to be complemented.

Programmer Note: These bits must be set to zero before any monitor to job exchange operation. If these bits are not set to zero via an 06 instruction, the connection network could produce invalid data on the read and invalid data could be written into memory.

The S and T designators are undefined.

A description of each of these faults can be found in specification 10354637, CDC FMP Functional Computer Specification.

SECDED FAULTS

The test is initiated by executing an 06 instruction with bits 9 through 15 selected of the P designator to complement the respective checkword bits of half-words 0, 1, 2, and 3 on the write scalar bus to Main Memory. By appropriate selection of data bits and complementation of checkword bits when

(continued)

3.2.1.7 (Cont.)

writing in memory, one should be able to generate SECDED faults on all read buses. This should allow complete checking of the read SECDED hardware and also the fault recording hardware for type and address of the fault.

The forced complementing of the checkword bits is discontinued by executing an 06 instruction with bits 9 through 15 of the CDC FMP.

This description explains the way the 06 instruction is executed on the CDC FMP.

3.2.1.8 07 ILLEGAL

3.2.1.9 08 4 NA MN INPUT/OUTPUT PER R

When in monitor mode: Activate the channel flag designated by the R designator and exit to the next sequential instruction. If the R designator specifies a non-existent channel, the operation of this instruction is undefined.

The S and T designators are undefined and must be set to zero.

3.2.1.10 09 4 64 BR EXIT FORCE

From a job to the monitor: Exchange to the monitor program. A hardware branch is then taken to the address defined by the right-most 48 bits of the monitor's register S. For this case, the R, S and T designators are undefined and must be set to zero.

From the monitor to a job: Exchange to the job whose invisible package is located starting at the absolute bit address contained in register T. For this case, the R designator is undefined and must be set to zero. If either the S designator or the contents of register S are equal to zero, the job's register file and the monitor's register file are identical.

R A D L

3.2.1.11 0A 4 64 MN TRANSMIT (R) TO MONITOR INTERVAL TIMER

When in monitor mode, transmit bits 40 through 63 of 64-bit register R to the monitor interval timer (see Section 3.1.8). The left-most 40 bits of register R are ignored. The S and T designators are undefined and must be set to zero.

3.2.1.12 0B ILLEGAL

3.2.1.13 0C ILLEGAL

3.2.1.14 0D ILLEGAL

3.2.1.15 0E 4 64 MN TRANSLATE EXTERNAL INTERRUPT (A9.0)

Each bit in the external interrupt register (EIR) is associated with an external I/O channel, or the monitor interval timer.

External Interrupt Register Bit	Assignment
0	I/O Channel 0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	I/O Channel 15
16	Monitor Interval Timer

Translate the lowest numbered bit set in the EIR into its associated four-bit code and transmit this code to the right-most four bits of register T. The left-most 60 bits of register T are cleared to zero.

Examine the EIR and if only one bit is set, the branch condition is met. The branch, if taken, is to (S) + (R) where (S) is an index in half-words and (R) is the base address.

The exit, be it a branch or not, clears the bit (and only that bit) in the EIR corresponding to the channel designator which was transmitted to register T.

If the T and S designators are equal, the interrupting channel designator will also be the branch index.

Bit zero of the EIR will never be set as it is reserved for maintenance purposes.

If no bit in the EIR is set, this instruction sets T to all zeros and no branch is taken.

3.2.1.16 0F ILLEGAL

3.2.1.17 10 A 64 RG CONVERT BCD TO BINARY, FIXED LENGTH

Convert the packed BCD number in register R to a signed (two's complement) binary number and place the result into the right-most 48 bits of register T. The conversion is undefined for binary results greater than $2^{47}-1$ or less than $(2^{47}-1)$; thus the largest decimal number that may be converted is ±140,737,488,355,327. The ASCII/EBCDIC sign code for the BCD number is in bits 60-63 of register R.

Data flag bit 39 will be set for numbers outside this range.

If the input number is not a valid BCD number, the results are undefined. Bits 0-15 of register T will be cleared to zero.

3.2.1.18 11 A 64 RG CONVERT BINARY TO BCD, FIXED LENGTH

Convert the right-most 48 bits (two's complement binary number) of register R to a packed BCD number and place the result in register T. The result is a number having 15 digits (4 bits per digit plus the sign in th lower bits - bits 60-63). The binary range is $\pm (2^{47}-1)$. During job mode, the sign bits generated are conditioned by the ASCII/EBCDIC bit in the job's invisible package. During monitor mode, only ASCII codes will be generated.

3.2.1.19 12 7 64 NT LOAD BYTE; (T) PER (S), (R)
3.2.1.20 13 7 64 NT STORE BYTE; (T) PER (S), (R)

Load/store a byte from/into the address specified by (R) + (S), where (R) is the base address and (S) is an item count of bytes, into/from bits 56 through 63 of register T. The remaining bits of T are cleared on a load and ignored on a store.

3.2.1.21 14 ILLEGAL

3.2.1.22 15 ILLEGAL

3.2.1.23 16 ILLEGAL

3.2.1.24 17 ILLEGAL

3.2.1.25 18 ILLEGAL

3.2.1.26 19 ILLEGAL

3.2.1.27 1A ILLEGAL

3.2.1.28 1B ILLEGAL

3.2.1.29 1C ILLEGAL

3.2.1.30 1D ILLEGAL

3.2.1.31 1E ILLEGAL

3.2.1.32 1F ILLEGAL

3.2.1.33 20 7 64 RG SHIFT (R) AND (R+1) PER S TO (T) AND (T+1)

This instruction shifts the 128-bit operand formed by catenating the contents of register R and register R+1 (bit 0 of register R+1 follows bit 63 of register R) and stores the results into the register designated by T and the next sequential register (T+1). The S designator specifies the type and amount of shift. If the S designator is in the range from 0 through 7F (0 through 127), the 128-bit operand is shifted left end-around the specified number of places. If the S designator is in the range from FF through 81 (-1 through -127), the 128-bit operand is shifted right with sign extension. For this case, bit zero of the operand

(continued)

3.2.1.33 (Cont.)

from register R is considered to be the sign bit of the shifted operand. The number of right shifts is equal to the two's complement of the S designator. If for example, S is equal to FE$_{16}$, the operand shifts right two places. If the S designator is greater that 7F$_{16}$ or less than 81$_{16}$, the results of this instruction are undefined. The R designator must specify an even register number. If the R designator is equal to zero, register zero will provide machine zero. This instruction does not test for machine zero or indefinite or set any data flags.

3.2.1.34 21 7 64 RG SHIFT (R) AND (R+1) PER (S) TO (T) AND (T+1)

This instruction shifts the 128-bit operand formed by catenating the contents of register R and register R+1 (bit 0 of register R+1 follows bit 63 of register R) and stores the results into the register designated by T and the next sequential register (T+1). The contents of the register designated by S determine the type and amount of shift. If the right-most byte of register S is in the range from 0 through 7F$_{16}$ (0 through 127$_{10}$), the 128-bit operand is shifted left end-around the specified number of places. If the right-most byte of register S is in the range from FF$_{16}$ through 81$_{16}$ (-1 through -127$_{10}$), the 128-bit operand is shifted right with sign extension. For this case, bit zero of the operand from register R is considered to be the sign bit of the shifted operand. The number of right shifts is equal to the two's complement of the right-most byte of register S. If the right-most byte of register S is greater than 7F$_{16}$ or less than 81$_{16}$, the results of this instruction are undefined. The left-most seven bytes of register S are ignored.

The R designator must specify an even register number. If the R designator is equal to zero, register zero will provide machine zero. This instruction does not cause a test for machine zero or indefinite or set any data flags.

3.2.1.35 22 ILLEGAL

3.2.1.36 23 ILLEGAL

3.2.1.37 24 ILLEGAL

3.2.1.38 25 ILLEGAL

3.2.1.39 26 ILLEGAL

3.2.1.40 27 ILLEGAL

3.2.1.41 28 ILLEGAL

3.2.1.42 29 ILLEGAL

3.2.1.43 2A ILLEGAL

3.2.1.44 2B 4 64 RG ADD TO LENGTH FIELD

Add bits 00 through 15 of register R to bits 48 through 63 of S and store the result in bits 00 through 15 of register T. Bits 16 through 63 of register R are transferred to bits 16 through 63 of register T.

3.2.1.45 2C 4 64 RG LOGICAL EXCLUSIVE OR (R),(S), TO (T)
3.2.1.46 2D 4 64 RG LOGICAL AND (R),(S), TO (T)
3.2.1.47 2E 4 64 RG LOGICAL INCLUSIVE OR (R),(S), TO (T)

These instructions perform the indicated logical functions listed below. The function occurs bit by bit on the 64-bit operands contained in the registers designated by R and S. The result in each case is stored in the register designated by T.

R	S	EXCLUSIVE OR R,S	AND R,S	INCLUSIVE OR R,S
0	0	0	0	0
0	0	1	0	1
1	0	1	0	1
1	1	0	1	1

If the R or S designators equal zero, register zero will contain machine zero.

3.2.1.48 2F 9 1 BR REGISTER BIT BRANCH AND ALTER

This instruction examines bit 63 of register T. As specified by the G designator a branch is made to the address contained in the right-most 48 bits of register S. The branch is made according to G bits 0 and 1 as follows:

G0	G1	
0	0	do not branch
0	1	branch unconditionally
1	0	branch if the object bit was a one
1	1	branch if the object bit was a zero

After the branch decision has been made and regardless of what that decision was, the object bit is altered according to G bits 2 and 3 as follows:

G2	G3	
0	0	do not alter the object bit
0	1	toggle the object bit to the other state
1	0	set the object bit to a one
1	1	clear the object bit to a zero.

3.2.1.49 30 7 64 RG SHIFT (R) PER S TO (T)

This instruction shifts the 64-bit operand from the register designated by R and stores the result into the register designated by T. The S designator specifies the type and amount of the shift. If the S designator is in the range from 0 through 3F$_{16}$ (0 through 63$_{10}$), the operand from register R is shifted left end-around the specified number of places and then stored in register T. If the S designator is in the range from FF$_{16}$ through C1$_{16}$ (-1 through -63$_{10}$), the operand from register T is shifted right with sign extension and then stored into register T. For this case, bit zero of the operand from register R is considered to be the sign bit of the shifted operand. The number of right shifts is equal to the two's complement of the S designator. If for example, S is equal to FE$_{16}$, the operand from register R shifts right two places. If the S designator is greater than 3F$_{16}$ or less than C1$_{16}$, the results of this instruction are undefined.

If the R designator is equal to zero, register zero will provide machine zero. This instruction does not test for machine zero or indefinite or set any data flags.

3.2.1.50 31 7 64 BR INCREASE(R) AND BRANCH IF(R) <> 0

Increment the contents of the right-most 48 bits of register R by one. The upper 16 bits of register R are not altered and arithmetic overflow is ignored.

If the result from above is 48 zeros, go to the next sequential instruction. If the 48-bit result from above is non-zero, branch to (S) + (T) where (S) is an item count of half-words and (T) is the base address. The resulting address for the branch is undefined if the R designator is equal to either the S designator or the T designator.

R A D L

3.2.1.51 32 9 1 BR BIT BRANCH AND ALTER

Register S contains the address of the object bit. This instruction reads up the word containing the object bit and examines the bit. The branch is then made according to G bits 0 and 1:

G0	G1	
0	0	do not branch
0	1	branch unconditionally
1	0	branch if the object bit was a one
1	1	branch if the object bit was a zero

After the branch decision has been made and regardless of what that decision was, the object bit is altered according to G bits 2 and 3 as follows:

G2	G3	
0	0	do not alter the object bit
0	1	toggle the object bit to the other state
1	0	set the object bit to a one
1	1	clear the object bit to a zero

NOTE: If G0 and G2 and G3 = 0, do not reference the object bit at all

If (G0 = 1) and (G2 and G3 = 0) read, but do not write the object bit

G bit 5 = 0 Register T contains the branch address

G bit 5 = $\overline{1}$ Branch address is formed by
G bit 6 = 0 1 adding the T designator, used as a half-word item count to the program address register

G bit 5 = $\overline{1}$ Branch address is formed by
G bit 6 = 1 1 subtracting the T designator, used as a half-word item count, from the program address register

3.2.1.52 33 B 1 BR DATA FLAG REGISTER BIT BRANCH AND ALTER

I is a six-bit designator specifying an object bit in the data flag register.

The object bit in the data flag register is examined and the decision to branch is made according to G bits 0 and 1.

G0	G1	
0	0	do not branch
0	1	branch unconditionally
1	0	branch if the object bit was a one
1	1	branch if the object bit was a zero

After the branch decision has been made and regardless of what that decision was, the object bit is altered according to G bits 2 and 3 as follows:

G2	G3	
0	0	do not alter the object bit
0	1	toggle the object bit to the other state
1	0	set the object bit to a one
1	1	clear the object bit to a zero

Programmer Note: It is meaningless to try to alter bits in the product field (bits 0-15) since the product field is strictly a function of the appropriate data flag and flag mask bits.

Since the 33 instruction begins execution without waiting until the machine has completed all operations, the data flag bits may set on any minor cycle during execution of the 33 instruction. Therefore, the object bit is sampled 2 minor cycles after the 33 instruction is loaded into IR0. This sampled object bit, rather than the actual object bit, is used to control the decision to branch, and the altering of the actual object bit in the data flag register. Consequently, any data flag bits that set after the object bit is sampled will not affect the decision to branch. Also, if the sampled object bit is a zero, any data flag bits that set afterwards will not be cleared nor toggled to a zero.

(continued)

3.2.1.52 (Cont.)

G bit 5 = 0 Register T contains the branch address.

G bit 5 = 1 | Branch address is formed by
G bit 6 = 0 | adding the T designator, used as an item count, in half-words, to the program address register

G bit 5 = 1 | Branch address is formed by
G bit 6 = 1 | subtracting the T designator, used as an item count, in half-words, from the program address register.

3.2.1.53 34 4 64 RG SHIFT(R) PER (S) TO (T)

This instruction shifts the 64-bit operand from the register designated by R and stores the result into the register designated by T. The register designated by S specifies the type and amount of the shift. If the right-most byte of register S is in the the range from 0 through $3F_{16}$ (0 through 63_{10}), the operand from register R is shifted left end-around the specified number of places and then stored into register T. If the right-most byte of register S is in the range from FF_{16} through $C1_{16}$ (-1 through -63_{10}), The operand from register R is shifted right with sign extension and then stored into register T. For this case, bit zero of the operand from register R is considered to be the sign bit of the shifted operand. The number of right shifts is equal to the two's complement of the right-most byte of register S. If the right-most byte of register S is greater than $3F_{16}$ or less than $C1_{16}$, the results of this instruction are undefined. The left-most seven bytes of register S are ignored.

If the R designator is equal to zero, register zero will provide machine zero. This instruction does not cause a test for machine zero or indefinite or set any data flags.

3.2.1.54 35 7 64 BR DECREASE (R) AND BRANCH IF (R) <> 0

Decrement the contents of the right-most 48 bits of register R by one. The upper 16 bits of register R are not altered and arithmetic overflow is ignored.

If the result from above is 48 zeros, go to the next sequential instruction. If the 48-bit result from above is non-zero, branch to (S) + (T) where (S) is an item count of half-words and (T) is the base address. The resulting address for the branch is undefined if the R designator is equal to either the S designator or the T designator.

3.2.1.55 36 7 64 BR BRANCH AND SET(R)TO NEXT INSTRUCTION

After storing the address of the next sequential instruction into register R, branch to (S) + (T) where (S) is an item count of half-words and (T) is the base address. Bits 0 through 15 of register R are forced to zeros. Bits 59 through 63 of register R are undefined. If the R designator is equal to the S designator the results of this instruction are undefined.

NOTE: If S=0, and R=T, this instruction sets register R to the half-word address of the next instruction and the program continues at the next instruction. This is a way to sample the program address register (P).

3.2.1.56 37 A 64 NT TRANSMIT JOB INTERVAL TIMER TO (T)

Transmit the contents of the job interval timer into bits 40-63 of register T. Bits 0-39 are cleared to zero. The R and S designators are undefined and must be set to zero. This instruction does not deactivate the time.

When executed in monitor mode, the operation of this instruction is undefined.

3.2.1.57 38 A 64 IN TRANSMIT (R BITS 00-15) TO (T BITS 00-15)

Replace the left-most 16 bits of register T with the left-most 16 bits of register R.

3.2.1.58 39 A 64 NT TRANSMIT REAL-TIME CLOCK TO(T)

Transmit the contents of the real-time clock to bits 16 through 63 of register T. Bits 00 through 15 are cleared. R and S must be zero.

3.2.1.59 3A A 64 NT TRANSMIT(R)TO JOB INTERVAL TIMER

When executed in job mode, this instruction transmits bits 40 through 63 of 64-bit register R to the job interval timer. S and T must be zero. (See Sections 3.1.6.3 and 3.1.8.3).

When executed in monitor mode, this instruction performs as a no op.

3.2.1.60 3B A 64 BR DATA FLAG REGISTER LOAD/STORE

Transfer the contents of register R to the data flag register and the original contents of the data flag register to register T. The S designator is undefined and must be set to zero. The R and T designators may be the same and this will swap data flag packages.

NOTE: An immediate data flag branch results at the termination of this instruction if the new contents of the data flag register meet the appropriate conditions.

3.2.1.61 3C 4 32 NT HALF-WORD INDEX MULTIPLY(R)*(S) TO (T)

The right-most 24 bits of registers R and S contain signed, two's complement integers. Their product is formed and stored into the right-most 24 bits of register T. The left-most 8 bits of register T are cleared to zeros.

If the product or either operand exceeds the value, $\pm(2^{23}-1)$ the result is undefined.

3.2.1.62 3D 4 64 NT INDEX MULTIPLY (R)*(S) TO (T)

The right-most 48 bits of registers R and S contain signed, two's complement integers. Their product is formed and stored into the right-most 48 bits of register T. The left-most 16 bits of register T are cleared to zeros.

If the product or either operand exceeds the value, $\pm(2^{47}-1)$ the result is undefined.

3.2.1.63 3E 6 64 IN ENTER(R) WITH I (16 BITS)

Clear register R and transfer the right-most 16 bits of this instruction to the right-most 48 bits of register R (the sign of the 16-bit immediate operand is extended through bit 16).

3.2.1.64 3F 6 64 IN INCREASE(R) BY I (16 BITS)

Replace the right-most 48 bits of register R by the sum of those bits and the right-most 16 bits of this instruction (the sign of the 16-bit immediate operand is extended through bit 16 for the addition). Arithmetic overflow is ignored.

3.2.1.65	40	4	32	RG	ADD U; (R)+(S) TO (T)
3.2.1.66	41	4	32	RG	ADD L; (R)+(S) TO (T)
3.2.1.67	42	4	32	RG	ADD N; (R)+(S) TO (T)
3.2.1.68	43				ILLEGAL
3.2.1.69	44	4	32	RG	SUB U; (R)-(S) TO (T)
3.2.1.70	45	4	32	RG	SUB L; (R)-(S) TO (T)
3.2.1.71	46	4	32	RG	SUB N; (R)-(S) TO (T)
3.2.1.72	47				ILLEGAL
3.2.1.73	48	4	32	RG	MPY U; (R)*(S) TO (T)
3.2.1.74	49	4	32	RG	MPY L; (R)*(S) TO (T)
3.2.1.75	4A				ILLEGAL
3.2.1.76	4B	4	32	RG	MPY S; (R)*(S) TO (T)
3.2.1.77	4C	4	32	RG	DIV U; (R)/(S) TO (T)

These instructions perform the indicated floating-point arithmetic operation on the 32-bit floating-point operands contained in the registers designated by R and S. The result in each case is stored in the register designated by T.

(continued)

3.2.1.77 (Cont.)

U signifies that the upper result of the operation is returned; L signifies the lower result; S signifies the significant result; and N signifies the normalized upper result.

Data flags: bits 41, 42, 43 and 46

3.2.1.78 4D 6 32 IN HALF-WORD ENTER R WITH I(16 BITS)

Clear register R and transfer the right-most 16 bits of this instruction to the right-most 24 bits of register R (the sign of the 16-bit immediate operand is extended through bit 8).

3.2.1.79 4E 6 32 IN HALF-WORD INCREASE R BY I(16 BITS)

Replace the right-most 24 bits of register R by the sum of those bits and the right-most 16 bits of this instruction (the sign of the 16-bit immediate operand is extended through bit 8 for the addition). Arithmetic overflow is ignored.

3.2.1.80 4F 4 32 RG DIV S; (R)/(S) TO (T)

This instruction performs a divide significant operation on the 32-bit floating-point operands contained in the registers designated by R and S. The result is stored in the register designated by T.

Data flags: bits 41, 42, 43 and 46

3.2.1.81 50 A 32 RG TRUNCATE; (R) TO (T)

Transmit to destination register T the nearest integer whose magnitude is less than or equal to the 32-bit floating-point operand in origin register R. This integer is represented as an unnormalized 32-bit floating-point number having a positive exponent.

If the exponent of the source operand is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

(continued)

3.2.1.81 (Cont.)

If the exponent of the source operand is negative, the magnitude of the coefficient is shifted right end off, and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Zeros are extended on the left during the shift. If the coefficient of the source operand is positive, the shifted coefficient with zero exponent is entered into the destination register. If the coefficient of the source operand is negative, the two's complement of the shifted coefficient with zero exponent is entered into the destination register.

If machine zero is used as an operand, 32 zeros are returned as a result.

Data flag: bit 46

3.2.1.82 51 A 32 RG FLOOR; (R) TO (T)

Transmit to destination register T the nearest integer less than or equal to the 32-bit floating-point operand in origin register R. This integer is represented as an unnormalized 32-bit floating-point number having a positive exponent.

If the source operand's exponent is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

If the exponent of the source operand is negative, the coefficient is shifted right end off and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Sign bits are extended on the left during the shift. The shifted coefficient with zero exponent is entered into the destination register.

If machine zero is used as an operand, 32 zeros are returned as a result.

Data flag: bit 46

3.2.1.83 52 A 32 RG CEILING;(R) TO (T)

Transmit to destination register T the nearest integer greater than or equal to the 32-bit floating-point operand in origin register R. This integer is represented as an unnormalized 32-bit floating-point number having a positive exponent.

(continued)

3.2.1.83 (Cont.)

If the source operand's exponent is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

If the exponent of the source operand is negative, the two's complement of the coefficient is shifted right end off and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Sign bits are extended on the left during the shift. The two's complement of the shifted coefficient with zero exponent is entered into the destination register.

If machine zero is used as an operand, 32 zeros are returned as a result.

Data flag: bit 46

3.2.1.84 53 A 32 RG SIGNIFICANT SQUARE ROOT; (R) TO (T)

Transmit to 32-bit register T the square root of a 32-bit floating-point operand in register R.

Data flags: bits 43, 45 and 46

3.2.1.85 54 4 32 RG ADJUST SIGNIFICANCE; (R) PER (S) TO (T)

Adjust the significance of the floating-point operand in register R and transmit it to result register T.

A signed, two's complement, integer is contained in the right-most 24 bits of register S. The absolute value of this integer is a shift count.

If the shift count is positive, shift the operand's coefficient left the number of places specified by the shift count or by the number of shifts needed to normalize the coefficient, whichever is smaller. In either case, the exponent of the operand is reduced by one for each place actually shifted. An all zero coefficient will be shifted left the number of places specified.

(continued)

3.2.1.85 (Cont.)

If the shift count is negative, shift the operand's coefficient right the number of places specified by the shift count and increase the exponent of the operand by one for each place shifted. If R is indefinite, T will be indefinite and data flag bit 46 is set. If R is machine zero, T will be machine zero and data flag bit 43 will be set.

This instruction is undefined if the absolute value of the shift count is greater than 2^{10}. Note that the addition of the shift count can cause either exponent overflow or exponent underflow.

Data flags: bits 42, 43 and 46

3.2.1.86 55 4 32 RG ADJUST EXPONENT; (R) PER (S) TO (T)

Transmit the adjusted operand from register R to result register T. The exponent of the result is set equal to the exponent of the operand in register S. The coefficient of the result is formed by shifting the coefficient of the operand from register R.

The shift count used is the difference between the exponents in registers R and S. If the exponent in register R is greater/less than the exponent in register S, the shift is to the left/right, respectively. For zero coefficients in register R, the exponent from register S is copied to register T with an all-zero coefficient.

If a left shift exceeds the number of places required for normalization, the result is set to indefinite, and data flag bit 42 is set. If either or both operands are indefinite or machine zero, the result is set to indefinite. In this case, data flag bit 46 is set and data flag bit 42 is not set.

Data flags: bits 42 and 46

3.2.1.87 56 7 64 SM BSWAP: R-->S or S-->T

Move data from the Backing Store (specified by source field R) to Main Memory (specified by S), or move data from Main Memory to the Backing Store (specified by T). When moving data from the Backing Store to Main Memory the T field must be zero; when moving data from Main Memory to the Backing Store the R field must be zero.

Bits 8 to 15 of register S specify the number of blocks to be transferred from one memory to another; one block is 32,768 64-bit words. Only an integral number of blocks may be transferred. A value of zero for transfer length makes the instruction a No op. The maximum length transfer is 255 blocks.

Bits 0 to 7, 16 to 34, and 43 to 63 of register S are unused. Bits 35 to 42 specify the block base address for the start of the transfer from/to Main Memory.

Bits 0 to 13, 16 to 29, and 43 to 63 of registers R and T are unused. Bits 14 and 15 are used only in monitor mode to manipulate the backing store block busy flags. A one in bit 14 means the blocks of Backing Store accessed by the BSWAP instruction will remain busy after completion of the swap (normally the Backing Store blocks are made busy at the issue of the BSWAP instruction and the busy flags are cleared block by block as the data transfer completes. A one in bit 15 is essentially an override of busy flags for blocks accessed by the BSWAP instruction. This allows monitor mode to lock down blocks by making (and leaving) them busy, yet allows monitor access to them. In job mode bits 14 and 15 are not used and a BSWAP proceeds as though both bits were zero -- blocks must not be busy at the start and are left not busy at completion.

Bits 30 to 42 of registers R and T specify the block base address for the start of the transfer from or to the Backing Store, respectively.

If the length of the transfer is such that the Swap Unit attempts to read or write past the end of either memory, instruction results are undefined.

(continued)

3.2.1.87 (Cont.)

Examples of 56 useage:

Register 10 --- 0010100001000000

Register 11 --- 0000100020000000

56111000 Move 10_{16} blocks of data from the Backing Store beginning at address 200000000 to Main Memory beginning at address 10000000.

56001011 Move 10_{16} blocks of data from Main Memory beginning at address 10000000 to the Backing Store beginning at address 200000000.

3.2.1.88 57 ILLEGAL

3.2.1.89 58 A 32 RG TRANSMIT; (R) TO (T)

Transmit the operand in 32-bit register R to 32-bit register T.

3.2.1.90 59 A 32 RG ABSOLUTE; (R) TO (T)

Transmit the absolute value of the 32-bit floating-point operand in register R to register T.

3.2.1.91 5A A 32 RG EXP.; (R) TO (T)

Transmit the exponent from the left-most 8 bit positions of the origin register R to the right-most 8 bit positions of destination register T. The sign of the exponent is extended through bit 8 of destination register T, the left-most 8 bits of the destination register are cleared to zeros.

3.2.1.92 5B 4 32 RG PACK; (R), (S) TO (T)

Transmit a 32-bit floating-point number to the destination register T. The exponent of the number is obtained from the right-most 8 bit positions of register R and the coefficient is obtained from the right-most 24 bit positions of register S.

3.2.1.93 5C A B RG EXTEND; 32-BIT(R) TO 64-BIT(T)

Extend the floating-point number from 32-bit register R into a 64-bit floating-point number and transmit the result to 64-bit register T. The value of the resulting 16-bit exponent is 24 less than that of the origin operand's exponent. The coefficient is obtained by transmitting the right-most 24 bits of the origin register into bits 16 through 39 of the destination register. The right-most 24 bits of the destination register are cleared to zero.

If R is indefinite, T will be indefinite and data flag bit 46 will be set. If R is machine zero, T will be machine zero and data flag bit 43 will be set.

Data flag: bit 43 and 46

3.2.1.94 5D A B RG INDEX EXTEND; 32-BIT(R) TO 64-BIT(T)

Extend the floating-point number from 32-bit register R into a 64-bit floating-point number and transmit the result to 64-bit register T. The value of the resulting 16-bit exponent is the same as the origin operand's exponent. The coefficient is obtained by transmitting the right-most 24 bits of the origin register into bits 40 through 63 of the destination register. Bits 16 through 39 of the destination register are set to the sign of the origin coefficient.

If R is indefinite, T will be indefinite and data flag bit 46 will be set. If R is machine zero, T will be machine zero and data flag bit 43 will be set.

Data flag: bit 43 and 46

3.2.1.95 5E 7 32 NT LOAD; (T) PER (S), (R)
3.2.1.96 5F 7 32 NT STORE; (T) PER (S), (R)

Load/store 32-bit register T from/into the address specified by (R) + (S) where (R) is the base address and (S) is an item count of half-words. Note that S and R are 64-bit registers and that the item count is shifted left five places before the addition. Overflow from this addition is ignored, if it occurs.

RADL

3.2.1.97 60 4 64 RG ADD U; (R)+(S) TO (T)
3.2.1.98 61 4 64 RG ADD L; (R)+(S) TO (T)
3.2.1.99 62 4 64 RG ADD N; (R)+(S) TO (T)

These instructions perform the indicated floating-point arithmetic operation on the 64-bit floating-point operands contained in the registers designated by R and S. The result in each case is stored in the register designated by T.

U signifies that the upper result of the operation is returned; L signifies the lower result; and N signifies the normalized upper result.

Data flags: bits 42, 43 and 46

3.2.1.100 63 4 64 RG ADD ADDRESS; (R)+(S) TO (T)

This instruction adds bits 16 through 63 of register R to bits 16 through 63 of register S and stores the result in bits 16 through 63 of register T. Bits 16 through 63 are treated as 48-bit, positive, unsigned integers. Arithmetic overflow is ignored. Bits 0 through 15 of register R are transferred without modification to bits 0 through 15 of register T.

3.2.1.101 64 4 64 RG SUB U; (R)-(S) TO (T)
3.2.1.102 65 4 64 RG SUB L; (R)-(S) TO (T)
3.2.1.103 66 4 64 RG SUB N; (R)-(S) TO (T)

These instructions perform the indicated floating-point arithmetic operation on the 64-bit floating-point operands contained in the registers designated by R and S. The result in each case is stored in the register designated by T.

U signifies that the upper result of the operation is returned; L signifies the lower result; and N signifies the normalized upper result.

Data flags: bits 42, 43 an 46

3.2.1.104 67 4 64 RG SUB ADDRESS; (R)-(S) TO (T)

This instruction subtracts bits 16 through 63 of register S from bits 16 through 63 of register R and stores the result in bits 16 through 63 of register T. Bits 16 through 63 are treated as 48-bit, positive unsigned integers. Arithmetic overflow is ignored. Bits 0 through 15 of register R are transferred without modification to bits 0 through 15 of register T.

3.2.1.105	68	4	64	RG	MPY U; (R)*(S) TO (T)
3.2.1.106	69	4	64	RG	MPY L; (R)*(S) TO (T)
3.2.1.107	6A				ILLEGAL
3.2.1.108	6B	4	64	RG	MPY S; (R)*(S) TO (T)
3.2.1.109	6C	4	64	RG	DIV U; (R)/(S) TO (T)

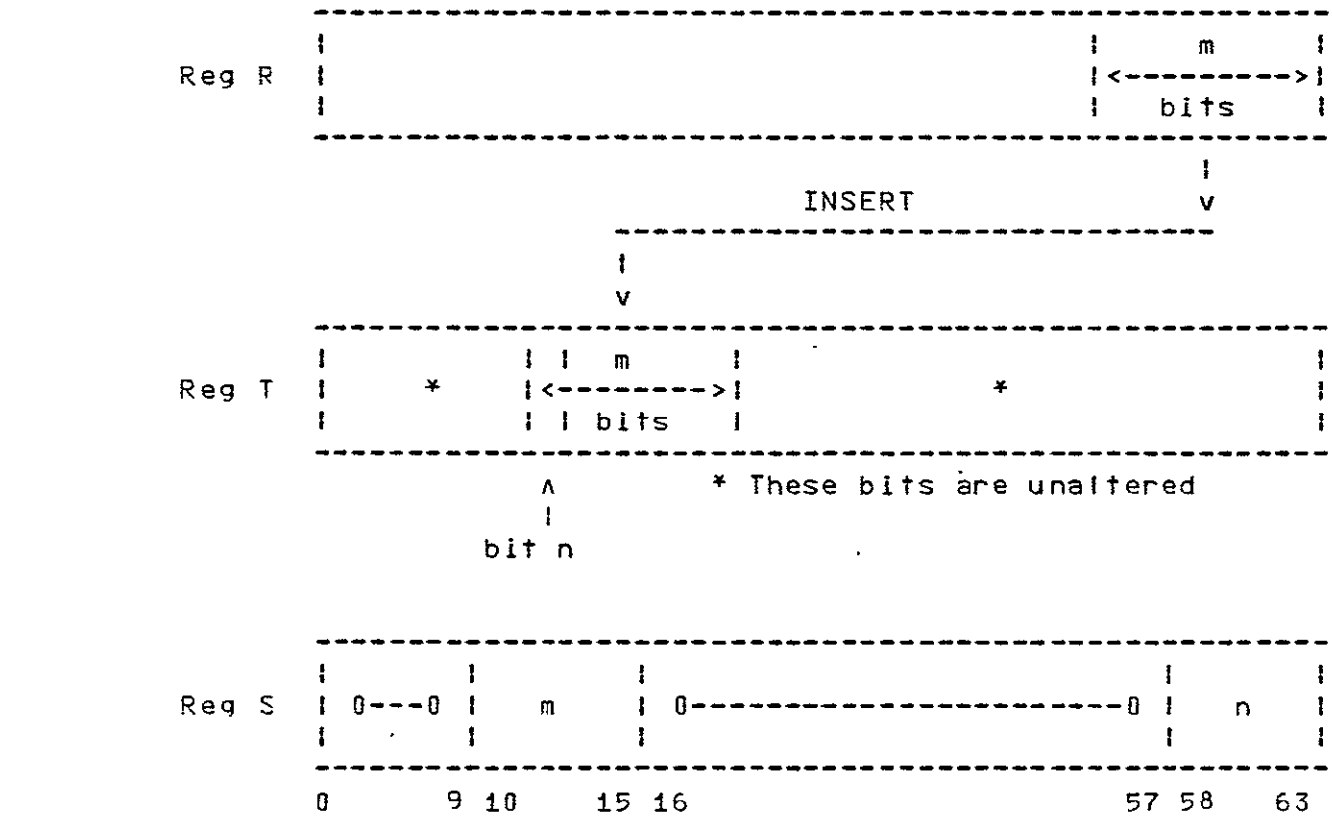
These instructions perform the indicated floating-point arithmetic operation on the 64-bit floating-point operands contained in the registers designated by R and S. The result in each case is stored in the register designated by T.

U signifies that the upper result of the operation is returned; L signifies the lower result; S signifies the significant result.

Data flags: bits 41, 42, 43 and 46

3.2.1.110	6D	4	64	RG	INSERT BITS; (R) TO (T) PER (S)

This instruction inserts the right-most bits of the register designated by R into the register designated by T.



(continued)

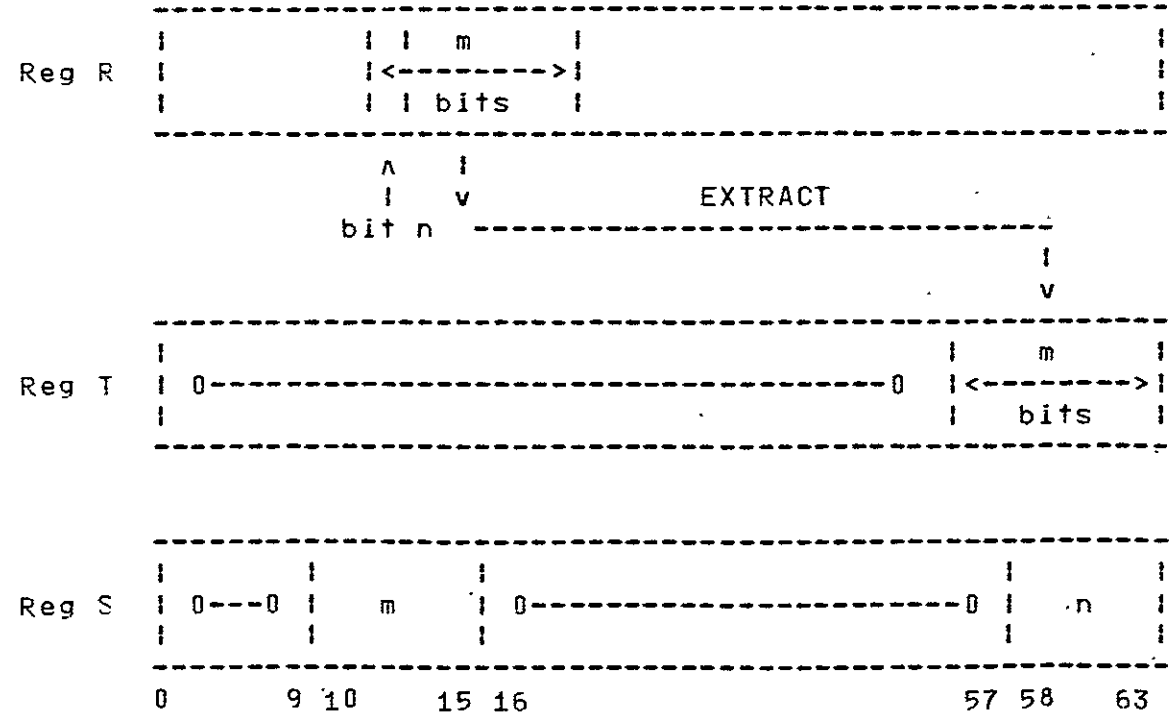
3.2.1.110 (Cont.)

Bits 10 through 15 of register S contain the number (m) of right-most bits to be inserted. The right-most 6 bits of register S specify the the bit number (n) in register T where the leftmost bit of the inserted data will be placed. Bits 0 through 9 and 16 through 57 of register S are undefined and must be set to zero.

If the R designator is equal to zero, then register zero will provide machine zero. If m plus n is greater than 64_{10}, or if m is equal to zero, the results of this instruction are undefined.

3.2.1.111 6E 4 64 RG EXTRACT BITS; (R) TO (T) PER (S)

This instruction extracts bits from register R and stores them into the right-most portion of register T. Register T is cleared before receiving the extracted bits.



(continued)

3.2.1.111 (Cont.)

Bits 10 through 15 of register S contain the number (m) of bits to be extracted from register R. The right-most 6 bits of register S specify the left-most bit number of the extracted bits. Bits 0 through 9 and 16 through 57 of register S are undefined and must be set to zero.

If the R designator is equal to zero, register zero will provide machine zero. If m plus n is greater than 64_{10}, or if m is equal to zero, the results of this instruction are undefined.

3.2.1.112 6F 4 64 RG DIV S; (R)/(S) TO (T)

This instruction performs a Divide Significant operation on the 64-bit floating-point operands contained in the registers designated by R and S. The result is stored in the register designated by T.

Data flags: bits 41, 42, 43 and 46

3.2.1.113 70 A 64 RG TRUNCATE; (R) TO (T)

Transmit to destination register T the nearest integer whose magnitude is less than or equal to the magnitude of the 64-bit floating-point operand in origin register R. The integer is represented as an unnormalized 64-bit floating-point number having a positive exponent.

If the exponent of the source operand is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

If the exponent of the source operand is negative, the magnitude of the coefficient is shifted right end off and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Zeros are extended on the left during the shift. If the coefficient of the source operand is positive, the shifted coefficient with zero exponent is entered into the destination register. If the coefficient of the source operand is negative, the two's complement of the shifted coefficient with zero exponent is entered into the destination register.

(continued)

3.2.1.113 (Cont.)

If a machine zero is used as an operand, 64 zeros are returned as a result.

Data flag: bit 46

3.2.1.114 71 A 64 RG FLOOR; (R) TO (T)

Transmit to destination register T the nearest integer less than or equal to the 64-bit floating-point operand in origin register R. This integer is represented as an unnormalized 64-bit floating-point number having a positive exponent.

If the source operand's exponent is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

If the exponent of the source operand is negative, the coefficient is shifted right end off and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Sign bits are extended on the left during the shift. The shifted coefficient with zero exponent is entered into the destination register.

If a machine zero is used as an operand, 64 zeros are returned as a result.

Data flag: bit 46

3.2.1.115 72 A 64 RG CEILING; (R) TO (T)

Transmit to destination register T the nearest integer greater than or equal to the 64-bit floating-point operand in origin register R. This integer is represented as an unnormalized 64-bit floating-point number having a positive exponent.

If the source operand's exponent is positive (greater than or equal to zero), the operand is transmitted directly to the destination register.

If the exponent of the source operand is negative, the two's complement of the coefficient is shifted right end off and the exponent is increased by one for each bit position shifted until the exponent becomes zero. Sign bits are extended on the left during the shift. The two's complement of the shifted coefficient with zero exponent is entered into the destination register.

(continued)

3.2.1.115 (Cont.)

If machine zero is used as an operand, 64 zeros are returned as a result.

Data flag: bit 46.

3.2.1.116 73 A 64 RG SIGNIFICANT SQUARE ROOT: (R) TO (T)

Transmit to register T the square root of the 64-bit floating-point operand in register R.

Data flags: bits 43, 45 and 46

3.2.1.117 74 4 64 RG ADJUST SIGNIFICANCE; (R) PER (S) TO (T)

Adjust the significance of the floating-point operand in register R and transmit it to result register T.

A signed, two's complement integer is contained in the right-most 48 bits of register S. The absolute value of this integer is a shift count. The left-most 16 bits of register S are ignored.

If the shift count is positive, shift the operand's coefficient left the number of places specified by the shift count or by the number of shifts needed to normalize the coefficient, whichever is smaller. In either case, the exponent of the operand is reduced by one for each place actually shifted. An all zero coefficient will be shifted left the number of places specified.

If the shift count is negative, shift the operand's coefficient right the number of places specified by the shift count and increase the exponent of the operand by one for each place shifted.

This instruction is undefined if the absolute value of the shift count is greater than 47_{10}. Note that the addition of shift count can cause either exponent overflow or exponent underflow.

If R is indefinite, T will be definite and data flag bit 46 will be set. If R is machine zero, T will be machine zero and data flag bit 43 will be set.

Data flags: bits 42, 43 and 46

R A D L

3.2.1.118 75 4 64 RG ADJUST EXPONENT; (R) PER (S) TO (T)

Transmit the adjusted operand from register R to result register T. The exponent of the result is set equal to the exponent of the operand in register S. The result is formed by shifting the coefficient of the operand from register R.

The shift count used is the difference between the exponents is register R and S. If the exponent in register R is greater/less than the exponent in register S, the shift is to the left/right, respectively. For zero coefficients in register R, the exponent from register S is copied to register T with an all-zero coefficient.

If a left shift exceeds the number of places required for normalization, the result is set to indefinite and data flag 42 is set. If either or both operands are indefinite or machine zero, the result is set to indefinite. In this case, data flag bit 46 is set and data flag bit 42 is not set.

R A D L

3.2.1.119 76 .A B RG CONTRACT; 64-BIT (R) TO 32-BIT (T)

Contract the 64-bit floating-point number from register R into a 32-bit floating-point number and transmit the result to 32-bit register T.

Input Exponent	Result
7FFF : 7000	Result Indefinite Indefinite Data Flag 46
6FFF : 0058	Result Indefinite Data Flag 42, 46
0057 : : : FF78	Result exponent 24 larger than input exponent Copy left-most 24 bits of input coefficient
FF77 : 8000	Result machine zero Data Flag 43

The 24-bit result coefficient is copied from the left-most 24 bits of the 48-bit source coefficient (bits 16 through 39). This has the effect of contracting all negative source coefficients, whose absolute values (neglecting the exponent) were less than or equal to 2^{24}, to a minus one.

Data flags: bits 42, 43 and 46

R A D L

3.2.1.120 77 A B RG ROUNDED CONTRACT; 64-BIT (R) TO 32-BIT (T)

Perform a rounded contract operation on the 64-bit floating-point number in register R and transmit the 32-bit floating-point result to 32-bit register T. A positive one is added to the origin operand in bit position 40. If overflow occurs the exponent is increased by one and the coefficient is shifted right one place. The left-most 24 bits of this 48-bit sum are then transmitted to the 24-bit coefficient portion of register T. Each non-endcase result element's 8-bit exponent is 24_{10} (25_{10} is overflow occurred) greater than the corresponding source element's exponent.

Data flags: bits 42, 43 and 46

3.2.1.121 78 A 64 RG TRANSMIT; (R) TO (T)

Transmit the 64-bit operand in register R to register T.

3.2.1.122 79 A 64 RG ABSOLUTE; (R) TO (T)

Transmit the absolute value of the 64-bit floating-point operand in register R to register T.

Data flags: bits 42, 43 and 46

3.2.1.123 7A A 64 RG EXP.; (R) TO (T)

Transmit the exponent from the left-most 16 bit positions of origin register R to the right-most 16 bit positions of destination register T. The sign of the exponent is extended through bit 16 of destination register T. The left-most 16 bits of the destination register are cleared to zeros.

3.2.1.124 7B 4 64 RG PACK; (R), (S) TO (T)

Transmit a 64-bit floating-point number to destination register T. The exponent of the number is obtained from the right-most 16 bit positions of register R, and the coefficient is obtained from the right-most 48 bit positions of register S.

3.2.1.125 7C A 64 RG LENGTH; (R) TO (T)

Transmit the left-most 16 bit positions of origin register R to the right-most 16 bit positions of destination register T. The left-most 48 bits of the destination register are cleared to zeros.

3.2.1.126 7D 7 64 NT SWAP; S----->T AND R----->S

Move to destination field T, a portion of the Register File beginning at the 64-bit register specified by the right-most eight bits of register S. Transmit source field R to the Register File beginning at the 64-bit register specified by the right-most eight bits of register S.

The left-most 16 bits of register R and T specify the field length in words for the source and destination fields, respectively. The field lengths of the source and destination fields may be different but each must be even. A zero field length indicates no transfer for that field. Any transfer of words into or out of the Register File that becomes exhausted of registers (i.e., beyond the bounds of the Register File), causes the instruction to become undefined.

The right-most 48 bits of registers R and T specify the base address of the source and destination fields, respectively. These addresses must specify an even 64-bit word in Main Memory. Bits 57 through 63 of register R and T are undefined and must be set to zero. Overlap of the source and destination fields is allowed only if the base addresses for both fields are equal.

Registers R, S, or T, may be in the range of the registers being swapped.

The starting register in the file specified by the right-most eight bits of register S must be an even register or this instruction will be treated as an undefined instruction. For additional material see Section 3.1.7 on the Register File.

3.2.1.127	7E	7	64	NT	LOAD; (T) PER (S), (R)
3.2.1.128	7F	7	64	NT	STORE; (T) PER (S), (R)

Load/store 64-bit register T from/into the address specified by (R) + (S) where (R) is the base address and (S) is an item count of words.

3.2.1.129	80	ILLEGAL
3.2.1.130	81	ILLEGAL
3.2.1.131	82	ILLEGAL
3.2.1.132	83	ILLEGAL
3.2.1.133	84	ILLEGAL
3.2.1.134	85	ILLEGAL
3.2.1.135	86	ILLEGAL
3.2.1.136	87	ILLEGAL
3.2.1.137	88	ILLEGAL
3.2.1.138	89	ILLEGAL
3.2.1.139	8A	ILLEGAL
3.2.1.140	8B	ILLEGAL
3.2.1.141	8C	ILLEGAL
3.2.1.142	8D	ILLEGAL
3.2.1.143	8E	ILLEGAL
3.2.1.144	8F	ILLEGAL
3.2.1.145	90	ILLEGAL
3.2.1.146	91	ILLEGAL
3.2.1.147	92	ILLEGAL
3.2.1.148	93	ILLEGAL
3.2.1.149	94	ILLEGAL

3.2.1.150 95 ILLEGAL

3.2.1.151 96 ILLEGAL

3.2.1.152 97 ILLEGAL

3.2.1.153 98 ILLEGAL

3.2.1.154 99 ILLEGAL

3.2.1.155 9A ILLEGAL

3.2.1.156 9B ILLEGAL

3.2.1.157 9C ILLEGAL

3.2.1.158 9D D E SM STREAM MAP

Depending on the value of the four-bit subfunction A, the Map Unit performs the functions described.

Subformat D1 Register File Reference Mode (16-bit parcel)

Field	Code	Operation
A	1	READ 1 Setup
	2	READ 2 Setup
	3	READ 3 Setup
	4	WRITE 1 Setup (W1A) source from READ 1
	5	WRITE-1 Setup (W1B) source from READ 2
	6	WRITE 1 Setup (W1C) source from Comoress/Mask/Merge Net
	7	WRITE 1 Setup (W1D) source from Gather/Scatter Net
	8	WRITE 1 Setup (W1E) source from Vector Unit
B	0	Register File Reference Mode (Subformat D1)
C	0	64-bit Mode
	1	32-bit Mode

(continued)

3.2.1.158 (Cont.)

D (for A code = 1-3, READ Setup)		
	0	Extend indefinite on input vector
	1	Extend floating-point zero on input vector
	2	Extend floating-point one on input vector
	3	Repeat input vector (for 64-bit mode, lowest three bits of field length must be zero; for 32-bit mode, lowest four bits of field length must be zero; if field length is zero, operand is broadcast)
D (for A code = 4-8, WRITE Setup)		
	0	Order vector operates on ones
	1	Order vector operates on zeros
	2, 3	Not defined
E	00-FF	Register file designator

Subformat D2 Immediate Reference Mode (64-bit parcel)

Field	Code	Operation
A	1	READ 1 Setup
	2	READ 2 Setup
	3	READ 3 Setup
	4	WRITE 1 Setup (W1A) source from READ 1
	5	WRITE 1 Setup (W1B) source from READ 2
	6	WRITE 1 Setup (W1C) source from Compress/Mask/Merge Net
	7	WRITE 1 Setup (W1D) source from Gather/Scatter Net
	8	WRITE 1 Setup (W1E) source from Vector Unit
B	1	Immediate Reference Mode (Subformat D2)
C	0	64-bit Mode
	1	32-bit Mode

(continued)

3.2.1.158 (Cont.)

D (for A code = 1-3, READ Setup)

0	Extend indefinite on input vector
1	Extend floating-point zero on input vector
2	Extend floating-point one on input vector
3	Repeat input vector (for 64-bit mode, lowest three bits of field length must be zero; for 32-bit mode, lowest four bits of field length must be zero; if field length is zero, operand is broadcast).

D (for A code = 4-8, WRITE Setup)

0	Order vector operates on ones
1	Order vector operates on zeros
2, 3	Not defined

E — 16-bit field length

F — 28-bit sword address

G — Lower address bits (used for shift count)

Z — Unused

Subformat 04 S1, S2 Connection Setup (64-bit parcel)

Field	Code	Operation
A	9	S1, S2 Connection setup
B, D		Destination code (B for S1, D for S2)
	0	No change
	1	Vector Unit
	2	Buffer Unit
	3	Both Vector Unit and Buffer Unit
	4	Internal to Map Unit

(continued)

3.2.1.158 (Cont.)

C, E — Source Code (C for S1, E for S2)

0	No change
1	R1
2	R2
3	Compress/Mask
4	Merge
5	Gather

Subformat 03 Map Unit Functions (16-bit parcel)

Field	Code	Operation
A	0	No op
	A	Map Unit functions
	B	Clear
B (For A code = A only, otherwise ignored)		
	1	Gather
	2	Scatter
	3	Compress
	4	Mask
	5	Merge
	6	Form Order Vector

C (For A code = A) C field bits specify the following:

2^7 = 1, order vector test greater than

2^6 = 1, order vector test not equal

2^5 = 1, 32-bit operands (=0, 64-bit)

2^4 = 1, move records (=0, move words)

2^3 = 1, order vector operates on zeros (=0, on ones)

2^2, 2^1, 2^0 not defined

(continued)

3.2.1.158 (Cont.)

C (For A code = B) C field bits each specify a unit to be cleared as follows:

2^7	READ 1
2^6	READ 2
2^5	READ 3
2^4	WRITE 1
2^3	Compress/Mask/Merge
2^2	Gather/Scatter
2^1	S1
2^0	S2

(continued)

3.2.1.158 (Cont.)

With a string of subfunction parcels taken from the previously listed functions, one can perform a number of operations within the Map Unit, or link the Map Unit to the Vector Unit to provide operand streams or to take result operands from the Vector Unit. Some examples of this linkage are:

Vector Transmit 10_{16} 64-bit words from memory address 10000000_{16} to address 1000000.

Assume that the register file is setup as follows:

Register 10= 0010|0000|0000|0000

Register 11= 0010|0000|0100|0000

and the instruction buffer holds the following: (All quantities in hexadecimal)

Total command: 9D01101040128028

Header (indirect mode for both vector references)

9D01 9D directs the operation to the Map Unit, the 1 indicates that one 32-bit packet follows the header packet.

First parcel: 1010 Function code 1 in first four bits indicates that this is a READ 1 setup. The next four bits are zero, indicating that:

B field=0 -- indirect reference mode
C field=0 -- 64-bit mode operation
D field=0 -- Extend A field (if shorter than C field) with indefinites.

The next 8 bits=10_{16}, the register designator pointing to register 10.

(continued)

3.2.1.158 (Cont.)

Second parcel: 4012 Function code 4 in first four bits indicates a WRITE 1 setup with source from READ 1, B, C, D fields all zero indicating 64-bit mode, indirect reference to the register file, and extension mode is ignored, the remaining 8 bits contain the register designator 12 pointing to the register containing the starting address and length of the output stream to be written from WRITE 1.

Third parcel: B028 Function code B indicates CLEAR operation; 28 specifies that R3 is to be disconnected (no order vector) and Compress/Mask/Merge is to be cleared (from any previous operation).

The same function could be programmed using the instruction itself to contain the field lengths and base addresses:

9D04B028180001010000000480000100100000O
which could be broken down into the following fields:

Header 9D04 Function sent to Map Unit, 4 32-bit packets follow to describe the operation.

First parcel: B028 Clear R3 and Compress/Mask/Merge

(continued)

3.2.1.158 (Cont.)

Second parcel: 1800001010000000
First four bits = 1 meaning a READ 1 setup, next four bits = 8 meaning that this is an immediate instruction (memory address and length to come directly from the instruction itself), 64-bit mode, extend indefinites; the next 8 bits are unused for the 9D instructions; the next 16 bits contain a 0010 or field length of 16_{10} elements; the remaining 32 bits contain the bit base address of the source field.

Third parcel: 4800001001000000
First four bits = 4 meaning a WRITE 1 setup with source from READ 1, the next four bits = 8, meaning that B=1 or the memory address and length are contained in the instruction, the next eight bits are unused, and the remaining 48 bits contain the 16-bit field length and 32-bit base address of the destination field.

Note that one field could be described by an immediate parcel and the other by an indirect reference to the Register File thus a 9D021010480000001001000000 would have an indirect reference to register 10 for the source field while the destination field address of 1000000 would be contained in the instruction itself.

(continued)

3.2.1.158 (Cont.)

A gather operation which would retrieve every fifth element from memory beginning at address 10000000 would require an additional parcel to be sent to the Map Unit to direct the gather operation. In this case, let register 05 contain: 0000100000500000 and location 500000 contain: 0000100000000005. Then the instruction would appear as:

9D02101033057012A1000000

which can be broken down as follows

header 9D02	--	Send function to Map Unit, 2 32-bit packets follow
Parcel 1: 1010	--	Setup READ 1 with register 10 (base address of vector)
Parcel 2: 3305	--	Setup READ 3 from register 5 (pointer to increment), repeat vector (increment)
Parcel 3: 7012	--	Setup WRITE 1 with source from Gather/Scatter Network, base address of output from register 12
Parcel 4: A100	--	A=functional control of Map Unit internal modules, 1=Gather operation, 00=64-bit elements
Parcel 5: 0000	--	No op to fill packet

3.2.1.159 9E D E SM BUFFER READ/WRITE SETUP

This instruction provides individual setup for each of four ports in the Buffer Unit. A buffer address and vector length can be provided for RB1 and RB2 (Read Buffer 1 and Read Buffer 2) and for WB1 and WB2 (Write Buffer 1 and Write Buffer 2). Subfunctions are specified by the A field of the instruction subformats.

Subformat D1 Register File Reference Mode (16-bit parcel)

Field	Code	Operation
A	0	No op
	1	Set up RB1 Port
	2	Set up RB2 Port
	3	Set up WB1 Port
	4	Set up WB2 Port
B	0	Register File Reference Mode (Subformat D1)
C	0	64-bit Mode
	1	32-bit Mode
D (for write setup, WB1 and WB2 source)		
	0	S1 (Source 1)
	1	S2 (Source 2)
	2	AR1 (Arithmetic Result 1)
	3	AR2 (Arithmetic Result 2)
D (extension form for nonconformal vectors, RB1 and RB2 setup)		
	0	Extend this stream with indefinites
	1	Extend this stream with machine zero
	2	Extend this stream with floating-point ones
	3	Repeat this stream from the beginning
E	00-FF	Register File Designator (for base address and field length)

(continued)

3.2.1.159 (Cont.)

Subformat 05 Immediate Reference Mode (32-bit parcel)

Field	Code	Operation
A	1	Set up RB1 Port
	2	Set up RB2 Port
	3	Set up WB1 Port
	4	Set up WB2 Port
B	1	Immediate Reference Mode (Subformat 05)
C	0	64-bit Mode
	1	32-bit Mode
D (for write setup, WB1 and WB2 source)		
	0	S1 (Source 1)
	1	S2 (Source 2)
	2	AR1 (Arithmetic Result 1)
	3	AR2 (Arithmetic Result 2)
D (extension form for nonconformal vectors, RB1 and RB2 set up)		
	0	Extend this stream with indefinites
	1	Extend this stream with machine zero
	2	Extend this stream with floating-point ones
	3	Repeat this stream from the beginning

E 12-bit field length of vector in words (if C field is zero) and half-words (if C field is a one); the right-most three (or four) bits of the field length are ignored, thus all vector functions from and to the buffer operate on groups of 8 words (or 16 half-words).

F 12-bit buffer base address in swords (eight-word groups). Thus a base address of 000_{16} would address words 0 through 7 of the vector and base address 001_{16} would reference words 8 through 15 of the buffer.

(continued)

3.2.1.159 (Cont.)

Base addresses taken from the Register File are bit addresses, however the low order nine bits are ignored, effectively making the address to the buffer a sword address.

Example: to perform a simple add of two vectors contained in the buffer with the result returned to the buffer would require three buffer setup commands and a Vector Unit command. The buffer instructions would appear as follows:

9E03 Instruction header for Buffer Unit setup, 3 packets to follow.

Parcel 1: 18100000

A field =1, Setup RB1
B field =1, Immediate Reference
C field =0, 64-bit Mode
D field =0, Extend with indefinites
E field =100 (16), number of similar elements to be processed (field length)
F field =000 (16), elements start at buffer address 000 (16) (base address)

Parcel 2: 28100100

A field =2, Setup RB2
B field =1, Immediate Mode
C field =0, 64-bit Mode
D field =0, Extend stream with indefinites
E field =100 (16), number of elements to be processed (field length)
F field =100 (16), elements start at buffer address 100 (16) (base address)

(continued)

3.2.1.159 (Cont.)

parcel 3: 3A100200

A field =3, Setup WB1 Port
B field =1, Immediate Mode
C field =0, 64-bit Mode
D field =2, Select AR1 bus for results from Vector Unit
E field =100 (16), number of elements to be stored in the buffer
F field =200 (16), start at base address 200 (16)

Thus the sequence 9E03000018100000281001003A100200 would deliver two streams of data to the Vector Unit via RB1 (Read Buffer 1) and RB2 (Read Buffer 2) starting at addresses 000 (16) and 100 (16), respectively, and continuing for 100 (16) elements. The results from the Vector Unit would be stored into the buffer beginning at base address 200 (16) for 100 (16) elements. Note that a no op (0000) was inserted after 9E03 to fill out the header packet.

3.2.1.160 9F .E E SM Vector Arithmetic

This 32-bit instruction controls the operations, and selection of input and output data busses, for the Vector Unit. Referring to instruction format E, the fields are defined as follows:

Field	Function	Codes and Meaning		
A	Operation Code	9F	Vector Arithmetic	
B	Suboperation	Code	AR1 Bus	AR2 Bus
	(Suboperations 00-17 are performed with normalized arithmetic)	00	A	C
		01	B	D
		02	A+B	C+D
		03	A+B	C*D
		04	A*B	C*D
		05	(A+B)*D	A+B
		06	(A+B)*(C*D)	C*D
		07	(A+B)*(C+D)	A+B
		08	(A+B)*(C+D)	Expand 32-bit C to 64 bits
		09	A+B+D	A+B
		0A	(A+B)+C*D	C*D
		0B	(A+B)*C+D	(A+B)*C
		0C	(A*B)+(C*D)	C*D
		0D	A*(B+C*D)	B+(C*D)
		0E	(A+B)*D	(A+B)*C
		0F	(A*C)+D	(A*C)+B
		10	(A*B)+D	C*D
		11	(A*B)+D	C+D
		12	DIVIDE 1	
		13	DIVIDE 2	
		14	Sum of products	
		15	Product of sums	
		16	Sum	
		17	Product	
		18	A+B Upper Sum	C+D Upper Sum
		19	A+B Lower Sum	C+D Lower Sum

(continued)

3.2.1.160 (Cont.)

Field	Function	Codes and Meaning		
		Code	AR1 Bus	AR2 Bus
		1A	A+B Upper Sum	C*D Upper Product
		1B	A+B Lower Sum	C*D Lower Product
		1C	A*B Upper Product	C*D Upper Product
		1D	A*B Lower Product	C*D Lower Product
C,D,E,F	Source busses for A,B,C&D streams	0	Source 1 (S1)\	
		1	Source 2 (S2)/from Map Unit	
		2	Read Buffer 1 (RB1)---	
		3	Read Buffer 2 (RB2)--!	
			!	
			v	
			from Buffer Unit	
G	Round/No round	0	No round results	
		1	Round results	
H,J	Complement B,D	0	No complement	
		1	Complement operands	
K	Null field	must be zero		
L,M	Select result busses to buffer	0	Do not select arithmetic result to buffer	
(L=AR1 select, M=AR2 select)		1	Select arithmetic result to buffer	
N	Write Bus 1 select	0	No select	
		1	Select AR1 to Write Bus 1	
		2	Select AR2 to Write Bus 1	
		3	Illegal	

(continued)

3.2.1.160 (Cont.)

Example:

The Vector Unit has four input busses and three output busses associated with it.
The busses are:

S1	Source 1 from the Map Unit
S2	Source 2 from the Map Unit
RB1	Read Buffer 1 from the Buffer Unit
RB2	Read Buffer 2 from the Buffer Unit
AR1	Arithmetic Result 1 --to the Buffer Unit
AR2	Arithmetic Result 2 --to the Buffer Unit
W1	WRITE 1 --to the Map Unit (either output AR1 or AR2 can be selected into the write bus trunk)

The contents of the internal busses AR1 and AR2 are defined by the suboperation field B in the instruction. A simple vector add utilizing data from Main Memory (via the Map Unit) and placing results back into Main Memory (via the Map Unit) requires a subfunction code of 2, with source operands into A and B selected from S1 and S2 respectively and with the AR1 output sent to W1. The resulting instruction would appear as:

9F021101 with the fields broken down as follows:

A=9F	Vector arithmetic
B=02	Select A+B and C+D to AR1 and AR2 respectively
C=0	A source from S1
D=1	B source from S2
E=0	C source from S1
F=1	D source from S2
	(C and D are unused in this example but necessary to activate the error checking)
G,H, J=0	No complement, no rounding
K=0	Must be zero
L,M=0	Do not select AR1 or AR2 to buffer ports
N=1	Select AR1 (which will contain A+B) into WRITE 1

(continued)

3.2.1.160 (Cont.)

Note that identical input selections into the A,C and B,D operands, and identical functions A+B and C+D, cause an automatic checking of the two adder outputs. In addition, the outputs of the multipliers, though idle, are compared for error checking during the processing of the addition operations.

3.2.1.161 A0 ILLEGAL

3.2.1.162 A1 ILLEGAL

3.2.1.163 A2 ILLEGAL

3.2.1.164 A3 ILLEGAL

3.2.1.165 A4 ILLEGAL

3.2.1.166 A5 ILLEGAL

3.2.1.167 A6 ILLEGAL

3.2.1.168 A7 ILLEGAL

3.2.1.169 A8 ILLEGAL

3.2.1.170 A9 ILLEGAL

3.2.1.171 AA ILLEGAL

3.2.1.172	AB	ILLEGAL				
3.2.1.173	AC	ILLEGAL				
3.2.1.174	AD	ILLEGAL				
3.2.1.175	AE	ILLEGAL				
3.2.1.176	AF	ILLEGAL				
3.2.1.177	B0	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) EQ (Z)	
3.2.1.178	B1	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) NE (Z)	
3.2.1.179	B2	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) GE (Z)	
3.2.1.180	B3	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) LT (Z)	
3.2.1.181	B4	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) LE (Z)	
3.2.1.182	B5	C	E	BR	COMPARE INTEGER, BRANCH IF (A) + (X) GT (Z)	

If bit 0 of the G designator is cleared/set, registers A, X, C and Z are 64/32 bits respectively. Registers B and Y are always 64 bits.

G bits 1 and 2 must be set to zero.

(continued)

3.2.1.182 (Cont.)

These instructions are executed in the following 5 steps:

1. Form the sum of the 48-bit (24-bit if G bit 0 = 1) integers from the right-most portion of registers A and X, ignoring overflows. If designators A and/or X equal zero, machine zero will be supplied.

2. Read register Z. If the Z designator is equal to zero compare against 48 zeros (24 zeros if G bit 0 = 1) may be made.

3. Store the following in register C:

   - o The sum from step 1 is stored into the right-most 48 bits (24 bits if G bit 0 = 1) of register C.
   - o The left-most 16 bits (8 bits if G bit 0 = 1) of register A are copied into the left-most portion of register C.

4. Compare the sum formed in step 1 with register Z as follows:

   - o G bit 3 = 0 The integers compared are the 48-bit (24 bits if G bit 0 = 1) result of step 1 and the right-most 48 bits (24 bits if G bit 0 = 1) read from register Z in step 2.
   - o G bit 3 = 1 The integers compared are the 64 bits that are stored into register C in step 3 and 64 bits read from register Z in step 2.

     This compare is defined only for the B0 and B1 instructions (EQ and NE).

     When both G bit 0 and G bit 3 are 1 the instructions are undefined.

(continued)

3.2.1.182 (Cont.)

- o G bit 4 = 0 The integers compared are interpreted as signed two's complement numbers.
- o G bit 4 = 1 The integers compared are interpreted as unsigned numbers.

The following table indicates the ordering of numbers from largest to smallest as controlled by G bit 4.

	0	1
Largest	7F -------- FF	FF -------- FF
	7F -------- FE	FF -------- FE
	.	.
	.	.
	.	.
	00 -------- 01	80 -------- 01
	00 -------- 00	80 -------- 00
	FF -------- FF	7F -------- FF
	.	.
	.	.
v	.	.
Smallest	80 -------- 01	00 -------- 01
	80 -------- 00	00 -------- 00

5. If the specified compare condition is met the instruction performs as follows:

- o G bit 5 = 0 Branch to the address formed by adding the half-word item count from register Y left shifted 5 places to the base address from register B.

(continued)

3.2.1.182 (Cont.)

- o G bit 5 = 1 Branch to the address formed by adding (G bit 6 = 0) or subtracting (G bit 6 = 1) the half-word item counts from the B and Y designators (16 bits), left shifted 5 places, to the program address of this instruction.

If the specified compare condition is not met, the instructions will continue execution at the next sequential instruction.

If any of the following conditions occur, the operation of these instructions is undefined.

- o G bit 0 = 1 and G bit 3 = 1
- o G bit 3 = 1 for B2, B3, B4 and B5
- o G bit 5 = 0 and G bit 6 = 1

The CDC FMP has expanded capabilities for the B0 through B5 instructions implemented by means of G bit 0 through 3 combinations.

B0	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) EQ (Z)
B1	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) NE (Z)
B2	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) GE (Z)
B3	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) LT (Z)
B4	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) LE (Z)
B5	C	E	NT	COMPARE INTEGER, SET CONDITION IF (A) + (X) GT (Z)

If bit 0 of the G designator is cleared/set, registers A, X, Y, C and Z are 64/32 bits respectively. Register B is not used and must be set to zero.

G bit 1 = 0 and G bit 2 = 1

These instructions are executed in 5 steps of which the first four (compare) steps are identical to the first four steps described for B0 through B5 instructions with G bits 1 and 2 equal to zero (compare branch)

(continued)

3.2.1.182 (Cont.)

If the specified compare condition is, met the instruction performs as follows:

Store into register Y and 64-bit quantity (32-bit if G bit 0 = 1) 000---001 and continue execution at the next sequential instruction.

If the specified compare condition is not met, the instruction oerforms as follows:

Store into register Y and 64-bit quantity (32-bit if G bit 0 = 1) 000---000 and continue execution at the next sequential instruction.

If any of the following conditions occur, the operation of these instructions is undefined:

- G bit 0 = 1 and G bit 3 = 1
- G bit 3 = 1 for B2, B3, B4 and B5
- G bit 5 = 1, G bit 6 = 1 or G bit 7 = 1
- The C designator is equal to the Z designator

```
B0   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) EQ (X)
B1   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) NE (X)
B2   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) GE (X)
B3   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) LT (X)
B4   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) LE (X)
B5   C   E   BR   COMPARE F.P., BRANCH IF (A) + (X) GT (X)
```

If bit 0 of the G designator is cleared/set, registers A and X are 64/32 bits respectively. Registers B and Y are always 64 bits. Registers C and Z are not used and must be set to zero.

G bit 1 = 1 and G bit 2 = 0

These instructions compare the two floating-point operands from registers A and X according to the floating-point compare rules in Section 3.1.4.5.

(continued)

3.2.1.182 (Cont.)

If the specified compare condition is met, the instructions perform as follows:

- o G bit 5 = 0 Branch to the address formed by adding the half-word item count from register Y, left shifted 5 places, to the base address from register B.
- o G bit 5 = 1 Branch to the address formed by adding (G bit 6 = 0) or subtracting (G bit 6 = 1) the half-word item counts from the B and Y designators 16 bits), left shifted 5 places, to the program address of this instruction.

If the specified compare condition is not met, the instructions will continue execution at the next sequential instruction.

If any of the following conditions occur, the operation of these instructions is undefined:

- o G bit 3 = 1, G bit 4 = 1 or G bit 7 = 1
- o Designator Z and/or C not equal to zero
- o G bit 5 = 0 and G bit 6 = 1

Data Flag: bit 46.

```
B0  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) EQ (Z)
B1  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) NE (Z)
B2  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) GE (Z)
B3  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) LT (Z)
B4  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) LE (Z)
B5  C  E  NT  COMPARE F.P, SET CONDITION IF (A) + (X) GT (Z)
```

If bit 0 of the G designator is cleared/set, registers A, X, and Y are 64/32 bits respectively. Registers B, C and Z are not used and must be set to zero.

(continued)

3.2.1.182 (Cont.)

G bit 1 = 1 and G bit 2 = 1

These instructions compare the two floating-point operands from registers A and X according to the floating-point compare rules in Section 3.1.4.5.

If the specified compare condition is met the instruction performs as follows:

Store into register Y and 64-bit quantity (32-bit if G bit 0 = 1) 000---000 and continue execution at the next sequential instruction.

If the specified compare condition is not met, the instruction performs as follows:

Store into register Y the 64-bit quantity (32-bit if G bit 0 = 1) 000---001 and continue execution at the next sequential instruction.

If any of the following conditions occur, the operation of these instructions is undefined:

- Any one of G bits 3 through 7 is set
- Designators B, Z and/or C are not equal to zero

Data Flag: bit 46.

R A D L

3.2.1.183 B6 5 NA BR BRANCH TO IMMEDIATE ADDRESS;(R)+I(48 BITS)

The right-most 48 bits of register R contain an item count of half-words. The right-most 48 bits of the instruction word contain an immediate operand which is used as a base address. An unconditional branch is taken to the branch address formed by adding the item count to the base address (the item count is shifted left 5 places before the addition and overflow, if any, is ignored).

A direct branch is taken to the base address from the instruction word if the R designator is zero or if the right-most 43 bits of register R are zeros.

3.2.1.184 B7 ILLEGAL

3.2.1.185 B8 ILLEGAL

3.2.1.186 B9 ILLEGAL

3.2.1.187 BA ILLEGAL

3.2.1.188 BB ILLEGAL

3.2.1.189 BC ILLEGAL

3.2.1.190 BD ILLEGAL

3.2.1.191 BE 5 64 IN ENTER (R) WITH I(48 BITS)

Clear register R and transfer the right-most 48 bits of this instruction to the right-most 48 bits of register R.

R A D L

3.2.1.192 BF 5 64 IN INCREASE (R) BY I(48 BITS)

Replace the right-most 48 bits of register R by the sum of those bits and the right-most 48 bits of this instruction word. Arithmetic overflow is ignored.

3.2.1.193 C0 ILLEGAL

3.2.1.194 C1 ILLEGAL

3.2.1.195 C2 ILLEGAL

3.2.1.196 C3 ILLEGAL

3.2.1.197 C4 ILLEGAL

3.2.1.198 C5 ILLEGAL

3.2.1.199 C6 ILLEGAL

3.2.1.200 C7 ILLEGAL

3.2.1.201 C8 ILLEGAL

3.2.1.202 C9 ILLEGAL

3.2.1.203 CA ILLEGAL

3.2.1.204 CB ILLEGAL

3.2.1.205 CC ILLEGAL

3.2.1.206 CD 5 32 IN HALF-WORD ENTER (R) WITH I(24 BITS)

Clear register R and transfer the right-most 24 bits of this instruction to the right-most 24 bits of register R.

3.2.1.207 CE 5 32 IN HALF-WORD INCREASE (R) BY I(24 BITS)

Replace the right-most 24 bits of register R by the sum of those bits and the right-most 24 bits of this instruction word. Arithmetic overflow is ignored.

3.2.1.208 CF ILLEGAL

3.2.1.209 D0 ILLEGAL

3.2.1.210 D1 ILLEGAL

3.2.1.211 D2 ILLEGAL

3.2.1.212 D3 ILLEGAL

3.2.1.213 D4 ILLEGAL

3.2.1.214 D5 ILLEGAL

3.2.1.215 D6 ILLEGAL

3.2.1.216 D7 ILLEGAL

3.2.1.217 D8 ILLEGAL

R A D L

3.2.1.218 D9 ILLEGAL

3.2.1.219 DA ILLEGAL

3.2.1.220 DB ILLEGAL

3.2.1.221 DC ILLEGAL

3.2.1.222 DD ILLEGAL

3.2.1.223	DE	ILLEGAL
3.2.1.224	DF	ILLEGAL
3.2.1.225	E0	ILLEGAL
3.2.1.226	E1	ILLEGAL
3.2.1.227	E2	ILLEGAL
3.2.1.228	E3	ILLEGAL
3.2.1.229	E4	ILLEGAL
3.2.1.230	E5	ILLEGAL
3.2.1.231	E6	ILLEGAL
3.2.1.232	E7	ILLEGAL
3.2.1.233	E8	ILLEGAL
3.2.1.234	E9	ILLEGAL
3.2.1.235	EA	ILLEGAL
3.2.1.236	EB	ILLEGAL
3.2.1.237	EC	ILLEGAL
3.2.1.238	ED	ILLEGAL
3.2.1.239	EE	ILLEGAL
3.2.1.240	EF	ILLEGAL

RADL

3.2.1.241	F0	ILLEGAL
3.2.1.242	F1	ILLEGAL
3.2.1.243	F2	ILLEGAL
3.2.1.244	F3	ILLEGAL
3.2.1.245	F4	ILLEGAL
3.2.1.246	F5	ILLEGAL
3.2.1.247	F6	ILLEGAL
3.2.1.248	F7	ILLEGAL
3.2.1.249	F8	ILLEGAL
3.2.1.250	F9	ILLEGAL
3.2.1.251	FA	ILLEGAL
3.2.1.252	FB	ILLEGAL
3.2.1.253	FC	ILLEGAL
3.2.1.254	FD	ILLEGAL
3.2.1.255	FE	ILLEGAL
3.2.1.256	FF	ILLEGAL

3.2.2 Instruction Execution Times

Instruction execution times are to be included in the appropriate machine description specifications. See Section 2.0.

4.0 TEST REQUIREMENTS (not applicable)

5.0 PREPARATION FOR DELIVERY (not applicable)

6.0 NOTES

6.1 ASCII/EBCDIC Reference Charts

The following table defines the control characters used in the ASCII Reference Chart.

NUL Null	DLE Data Link Escape (CC)
SOH Start of Heading (CC)	DC1 Device Control 1
STX Start of Text (CC)	DC2 Device Control 2
ETX End of Text (CC)	DC3 Device Control 3
EOT End of Transmission (CC)	DC4 Device Control 4 (Stop)
ENQ Enquiry (CC)	NAK Negative Acknowledge (CC)
ACK Acknowledge (CC)	SYN Synchronous Idle (CC)
BEL Bell (audible or attention signal)	ETB End of Transmission Block (CC)
BS Backspace (FE)	CAN Cancel
HT Horizontal Tabulation (punched card skip (FE)	EM End of Medium
LF Line Feed (FE)	SUB Substitute

NOTE: (CC) Communication Control
(FE) Format Effector
(IS) Information Separator

(continued)

6.1 (Cont.)

VT Vertical Tabulation (FE)	ESC Escape
FF Form Feed (FE)	FS File Separator (IS)
CR Carriage Return (FE)	GS Group Separator (IS)
SO Shift Out	RS Record Separator (IS)
SI Shift In	US Unit Separator (IS)
	DEL Delete [1]

NOTE: (CC) Communication Control
(FE) Format Effector
(IS) Information Separator

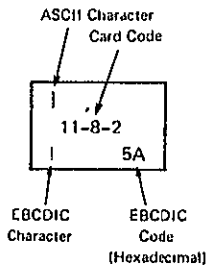
[1] In the strict sense, DEL is not a control character.

(continued)

AMERICAN NATIONAL STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII) WITH PUNCHED CARD CODES AND EBCDIC TRANSLATION

	b_8 b_7 b_6 b_5	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
b_4 b_3 b_2 b_1	COL / ROW	0	1	2	3	4	5	6	7	8	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)
0 0 0 0	0	NUL 12-0-9-8-1 NUL 00	DLE 12-11-9-8-1 DLE 10	SP no punch SP 40	0 0 0 F0	@ 8-4 @ 7C	P 11-7 P D7	` 8-1 ` 79	p 12 11-7 p 97	12-11 0-9 8-1 DS 20	12-11 0-9 8-1 30	12-0-9-1 41	12-11 9 8 58	12-11-0-9 6 76	12 11 8-7 9F	12 11 0 8 B8	12 11 9 8 4 DC
0 0 0 1	1	SOH 12-9-1 SOH 01	DC1 11-9-1 DC1 11	! 12-8-7 ! 4F	1 1 1 F1	A 12-1 A C1	Q 11-8 Q D8	a 12-0-1 a 81	q 12-11-8 q 98	0-9-1 SOS 21	9-1 31	12-0-9-2 42	11 8-1 59	12-11 0-9-7 77	11 0 8 1 A0	12 11 0-9 B9	12 11 9 8-5 DD
0 0 1 0	2	STX 12-9-2 STX 02	DC2 11 9-2 DC2 12	8 7 7F	2 2 2 F2	B 12-2 B C2	R 11-9 R D9	b 12-0-2 b 82	r 12-11-9 r 99	0-9-2 FS 22	11-9 8-2 CC 1A	12 0-9 3 43	11-0-9-2 62	12 11 0 9 8 78	11 0 8 2 AA	12 11 0 8-7 BA	12 11 9 8 6 DE
0 0 1 1	3	ETX 12-9-3 ETX 03	DC3 11-9-3 TM 13	= 8-3 = 7B	3 3 3 F3	C 12-3 C C3	S 0-2 S E2	c 12-0-3 c 83	s 11-0-2 s A2	0 9-3 23	9-3 33	12-0-9-4 44	11-0 9-3 63	12-0-8-1 80	11-0-8 3 AB	12-11 0 8 3 BB	12 11 9 8 7 DF
0 1 0 0	4	EOT 9-7 EOT 37	DC4 9-8-4 DC4 3C	$ 11-8-3 $ 5B	4 4 4 F4	D 12-4 D C4	T 0-3 T E3	d 12-0-4 d 84	t 11-0-3 t A3	0-9-4 BYP 24	9-4 PN 34	12-0-9-5 45	11-0-9-4 64	12-0 8 2 8A	11-0-8 4 AC	12 11 0-8-4 BC	11-0-9 8-2 EA
0 1 0 1	5	ENQ 0-9-8-5 ENQ 2D	NAK 9-8-5 NAK 3D	% 0-8-4 % 6C	5 5 5 F5	E 12-5 E C5	U 0-4 U E4	e 12-0-5 e 85	u 11-0-4 u A4	11-9-5 NL 15	9-5 BS 35	12-0-9-6 46	11-0 9-5 65	12-0-8-3 8B	11-0-8-5 AD	12-11-0 8-5 BD	11-0-9-8-3 EB
0 1 1 0	6	ACK 0-9-8-6 ACK 2E	SYN 9 2 SYN 32	& 12 & 50	6 6 6 F6	F 12-6 F C6	V 0-5 V E5	f 12-0-6 f 86	v 11-0-5 v A5	12-9-6 LC 06	/ 9-6 UC 36	12 0 9 7 47	11-0-9 6 66	12-0-8-4 8C	11 0 8 6 AE	12-11-0 8-6 BE	11 0-9 8 4 EC
0 1 1 1	7	BEL 0-9-8-7 BEL 2F	ETB 0-9-6 ETB 26	' 8-5 ' 7D	7 7 7 F7	G 12-7 G C7	W 0-6 W E6	g 12-0-7 g 87	w 11-0-6 w A6	11-9-7 EL 17	12-9-8 ETB 08	12-0-9-8 48	11-0 9-7 67	12-0-8-5 8D	11 0 8 7 AF	12 11-0 8-7 BF	11 0-9-8-5 ED
1 0 0 0	8	BS 11-9-6 BS 16	CAN 11-9-8 CAN 18	(12-8-5 (4D	8 8 8 F8	H 12-8 H C8	X 0-7 X E7	h 12-0-8 h 88	x 11-0-7 x A7	0-9-8 28	9-8 38	12-8-1 49	11-0-9-8 68	12-0-8-6 8E	12-11-0-8-1 B0	12 0-9-8-2 CA	11 0-9 8-6 EE
1 0 0 1	9	HT 12-9-5 HT 05	EM 11-9-8-1 EM 19	) 11-8-5) 5D	9 9 9 F9	I 12-9 I C9	Y 0-8 Y E8	i 12-0-9 i 89	y 11-0-8 y A8	0-9-8-1 29	9-8-1 39	12-11-9-1 51	0 8-1 69	12-0-8-7 8F	12 11-0-1 B1	12-0 9-8-3 CB	11-0-9-8-7 EF
1 0 1 0	10 (A)	LF 0-9-5 LF 25	SUB 9-8-7 SUB 3F	* 11-8-4 * 5C	: 8-2 7A	J 11-1 J D1	Z 0-9 Z E9	j 12-11-1 j 91	z 11-0-9 z A9	0-9-8-2 SM 2A	, 9-8-2 3A	12-11 9-2 52	12 11-0 70	12-11-8-1 90	12-11-0-2 B2	12 0-9-8-4 ¢ CC	12-11-0-9 8-2 (0 VM) FA
1 0 1 1	11 (B)	VT 12-9-8-3 VT 0B	ESC 0-9-7 ESC 27	+ 12-8-6 + 4E	; 11-8-6 ; 5E	K 11-2 K D2	[12 8 2 [4A	k 12-11-2 k 92	{ 12 0 { C0	0-9-8-3 CU2 2B	9 8-3 CU3 3B	12-11-9-3 53	12-11-0-9-1 71	12-11-8-2 9A	12-11 0-3 B3	12-0-9-8-5 CD	12-11-0-9-8-3 FB
1 1 0 0	12 (C)	FF 12-9-8-4 FF 0C	FS 11-9-8-4 IFS 1C	0-8-3 , 6B	< 12-8-4 < 4C	\ 0-8-2 \ E0	L 11-3 L D3	l 12-11-3 l 93	| 12-11 | 6A	0-9-8-4 2C	12-9-4 PF 04	12-11-9-4 54	12 11 0-9-2 72	12 11-8-3 9B	12-11-0-4 B4	12 0-9-8-6 ↓ CE	12 11-0-9-8-4 FC
1 1 0 1	13 (D)	CR 12-9-8-5 CR 0D	GS 11-9-8-5 IGS 1D	- 11 - 60	8-6 - 7E	M 11-4 MD4	] 11 8 2] 5A	m 12-11-4 m 94	} 11 0 } D0	12-9-8-1 RLF 09	11-9-4 RES 14	12-11 9-5 55	12-11-0-9-3 73	12-11-8-4 9C	12-11-0-5 B5	12-0-9 8-7 CF	12-11 0-9-8 5 FD
1 1 1 0	14 (E)	SO 12-9-8-6 SO 0E	RS 11-9-8-6 IRS 1E	. 12-8-3 . 4B	> 0-8-6 > 6E	N 11-5 N D5	^ 11 8 7 ^ 5F	n 12 11-5 n 95	~ 11-0-1 ~ A1	12-9-8-2 SMM 0A	9-8-6 3E	12-11-9-6 56	12-11-0-9-4 74	12-11-8 5 9D	12-11 0-6 B6	12-11-9-8-2 DA	12 11-0-9 8-6 FE
1 1 1 1	15 (F)	SI 12-9-8-7 SI 0F	US 11-9-8-7 IUS 1F	/ 0-1 / 61	? 0-8-7 ? 6F	O 11-6 O D6	_ 0-8-5 _ 6D	o 12-11-6 o 96	DEL 12-9-7 DEL 07	11-9-8-3 CUI 1B	11-0-9-1 E1	12-11-9-7 57	12 11-0-9-5 75	12-11-8-6 9E	12-11-0-7 BJ	12-11-9-8-3 DB	EO 12 11-0 9-8-7 FF

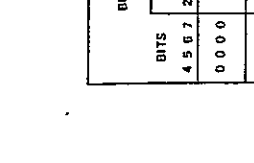
LEGEND



EXTENDED BINARY CODED DECIMAL INTERCHANGE CODE (EBCDIC) WITH PUNCHED CARD CODES AND ASCII TRANSLATION

BITS 4 5 6 7	1ST HEX 2ND	BITS 0 1 2 3 = 0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0	NUL 12-0-9-8-1 NUL 00	DLE 12-11-9-8-1 DLE 10	DS 11-0-9-8-1 80	12-11-0-9-8-1 90	SP no punch SP 20	& 12 & 26	- 11 - 20	12-11-0	12-0-8-1 C3	12-11-8-1 CA	11-0-8-1 D1	12-11-0-8-1 D8	12-0 {11-0 7B 7D	0-8-2 \ 5C	0 0 30
0001	1	SOH 12-9-1 SOH 01	DC1 11-0-1 DC1 11	SOS 0 9 1 81	0-1 91	12-0-9-1 A0	12-11-0-1 A9	/ 0-1 2F	12-11-0-9-1 B9	a 12-0-1 61	j 12-11-1 6A	~ 11-0-1 7E	/ 12-11-0-1 D9	A 12-1 A 41	J 11-1 J 4A	11-0-9-1 9F	1 1 31
0010	2	STX 12-9-2 STX 02	DC2 11-9-2 DC2 12	FS 0-9-2 82	SYN 9-2 SYN 16	12-0-9-2 A1	12-11-0-9-2 AA	11-0-9-2 B2	12-11-0-9-2 BC	b 12-0 2 62 b	k 12-11-2 6B	s 11-0-2 73	12-11-0-2 DA	B 12-2 B 42	K 11-2 K 4B	S 0-8-2 S 53	2 2 32
0011	3	ETX 12-9-3 ETX 03	TM 11-9-3 DC3 13	0-9-3 83	9-3 93	12-0-9-3 A2	12-11-0-3 AB	11-0-9-3 B3	12-11-0-9-3 BD	c 12-0-3 63 c	l 12-11-3 6C	t 11-0-3 7A	12-11-0-3 DB	C 12-3 C 43	L 11-3 L 4C	T 0-3 T 54	3 3 33
0100	4	PF 12-9-4 9D	RES 11-9-4	BYP 0-9-4 84	PN 9-4 94	12-0-9-4 A3	12-11-0-4 AC	11-0-9-4 B4	12-11-0-9-4 BE	d 12-0-4 64 d	m 12-11-4 6D	u 11-0-4 75	12-11-0-4 DC	D 12-4 D 44	M 11-4 M 4D	U 0-4 U 55	4 4 34
0101	5	HT 12-9-5 HT 09	NL 11-9-5 85	LF 0-9-5 LF 0A	RS 9-5 95	12-0-9-5 A4	12-11-0-5 AD	11-0-9-5 B5	12-11-0-9-5 BF	e 12-0-5 65 e	n 12-11-5 6E n	v 11-0-5 76	12-11-0-5 DD	E 12-5 E 45	N 11-5 N 4E	V 0-5 V 56	5 5 35
0110	6	LC 12-0-6 86	BS 11-9-6 08	ETB 0-9-6 ETB 17	UC 9-6 96	12-0-9-6 A5	12-11-0-6 AE	11-0-9-6 B6	12-11-0-9-6 C0	f 12-0-6 66 f	o 12-11-6 6F o	w 11-0-6 77	12-11-0-6 DE	F 12-6 F 46	O 11-6 O 4F	W 0-6 W 57	6 6 36
0111	7	DEL 12-9-7 DEL 7F	IL 11-9-7 87	ESC 0-9-7 ESC 1B	EOT 9-7 EOT 04	12-0-9-7 A6	12-11-0-7 AF	11-0-9-7 B7	12-11-0-9-7 C1	g 12-0-7 67 g	p 12-11-7 70 p	x 11-0-7 78 x	12-11-0-7 DF	G 12-7 G 47	P 11-7 P 50	X 0-7 X 58	7 7 37
1000	8	GE 12-9-8 97	CAN 11-9-8 CAN 18	0-9-8 88	9-8 98	12-0-9-8 A7	12-11-0-8 B0	11-0-9-8 B8	12-11-0-9-8 C2	h 12-0-8 68 h	q 12-11-8 71 q	y 11-0-8 79 y	12-11-0-8 E0	H 12-8 H 48	Q 11-8 Q 51	Y 0-8 Y 59	8 8 38
1001	9	RLF 12-9-8-1 8D	EM 11-9-8-1 EM 19	0-9-8-1 89	9-8-1 99	12-0-9-8-1 A8	11-8-1 B1	0-8-1 B9	` 8-1 60	i 12-0-9 69 i	r 12-11-9 72 r	z 11-0-9 7A	12-11-0-9 E1	I 12-9 I 49	R 11-9 R 52	Z 0-9 Z 5A	9 9 39
1010	A 10	SMM 12-9-8-2 8E	CC 11-9-8-2 92	SM 0-9-8-2 8A	9-8-2 9A	¢ 12-8-2 [5B	! 11-8-2] 5D		8-2 3A	12-0-8-2 C4	12-11-8-2 C8	11-0-8-2 D2	12-11-0-8-2 E2	12-0-9-8-2 EB	12-11-9-8-2 EC	11-0-9-8-2 F4	11(LVM) 12-11-0-9-8-2 FA
1011	B 11	VT 12-9-8-3 VT 0B	CU1 11-9-8-3 8F	CU2 0-9-8-3 8B	CU3 9-8-3 9B	$ 12-8-3 $ 2E	$ 11-8-3 $ 24	` 0-8-3 , 2C	= 8-3 = 23	12-0-8-3 C5	12-11-8-3 CC	11-0-8-3 D3	12-11-0-8-3 C3	12-0-9-8-3 E9	12-11-9-8-3 EF	11-0-9-8-3 F5	12-11-0-9-8-3 FB
1100	C 12	FF 12-9-8-4 FF 0C	IFS 11-9-8-4 1C	DC4 9-8-4 DC4 8C	9-8-4 14	< 12-8-4 < 3C	. 11-8-4 . 2A	% 0-8-4 % 25	@ 8-4 @ 40	12-0-8-4 C6	12-11-8-4 CD	11-0-8-4 D4	12-11-0-8-4 E4	J 12-0-9-8-4 EA	11-0-9-8-4 F0	12-11-0-9-8-4 FC	
1101	D 13	CR 12-9-8-5 CR 0D	IGS 11-9-8-5 1D	ENQ 0-9-8-5 ENQ 05	NAK 9-8-5 NAK 15	(12-8-5 (28	) 11-8-5) 29	~ 0-8-5 ~ 5F	, 8-5 , 27	12-0-8-5 C7	12-11-8-5 CE	11-0-8-5 D5	12-11-0-8-5 E5	12-0-9-8-5 EB	12-11-9-8-5 F1	11-0-9-8-5 F7	12-11-0-9-8-5 FD
1110	E 14	SO 12-9-8-6 SO 0E	IRS 11-9-8-6 1E	ACK 0-9-8-6 ACK 06	9-8-6 9E	+ 12-8-6 + 2B	; 11-8-6 ; 3B	> 0-8-6 > 3E	. 8-6 . 3D	12-0-8-6 C8	12-11-8-6 CF	11-0-8-6 D6	12-11-0-8-6 E6	Ų 12-0-9-8-6 EC	12-11-9-8-6 F2	11-0-9-8-6 F8	12-11-0-9-8-6 FE
1111	F 15	SI 12-9-8-7 SI 0F	IUS 11-9-8-7 1F	BEL 0-9-8-7 BEL 07	SUB 9-8-7 SUB 1A	| 12-8-7 ! 21	¬ 11-8-7 ^ 5E	? 0-8-7 ? 3F	' 8-7 ' 22	12-0-8-7 C9	12-11-8-7 D0	11-0-8-7 D7	12-11-0-8-7 E7	12-0-9-8-7 ED	12-11-9-8-7 F3	11-0 9-8-7 F9	12-11-0-9 8-7 FF EO

LEGEND



AMERICAN NATIONAL STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII) WITH PUNCHED CARD CODES AND EBCDIC TRANSLATION

b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1	ROW \ COL. →	0	1	2	3	4	5	6	7	8	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)
0 0 0 0	0	NUL	DLE	SP	0	@	P	\	p								
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q								
0 0 1 0	2	STX	DC2	"	2	B	R	b	r								
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s								
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t								
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u								
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v								
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w								
1 0 0 0	8	BS	CAN	(	8	H	X	h	x								
1 0 0 1	9	HT	EM	)	9	I	Y	i	y								
1 0 1 0	10 (A)	LF	SUB	*	:	J	Z	j	z								
1 0 1 1	11 (B)	VT	ESC	+	;	K	[	k	{								
1 1 0 0	12 (C)	FF	FS	,	<	L	\	l	¦								
1 1 0 1	13 (D)	CR	GS	-	=	M	]	m	}								
1 1 1 0	14 (E)	SO	RS	.	>	N	^	n	~								
1 1 1 1	15 (F)	SI	US	/	?	O	_	o	DEL								EO

R A D L

EXTENDED BINARY CODED DECIMAL INTERCHANGE CODE (EBCDIC) WITH PUNCHED CARD CODES AND ASCII TRANSLATION

b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1	ROW \ COL. →	0	1	2	3	4	5	6	7	8	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)
0 0 0 0	0	NUL	DLE			SP	&	–						{	}	\	0
0 0 0 1	1	SOH	DC1					/		a	j	~		A	J		1
0 0 1 0	2	STX	DC2		SYN					b	k	s		B	K	S	2
0 0 1 1	3	ETX	DC3							c	l	t		C	L	T	3
0 1 0 0	4									d	m	u		D	M	U	4
0 1 0 1	5	HT		LF						e	n	v		E	N	V	5
0 1 1 0	6		BS	ETB						f	o	w		F	O	W	6
0 1 1 1	7	DEL		ESC	EOT					g	p	x		G	P	X	7
1 0 0 0	8		CAN							h	q	y		H	Q	Y	8
1 0 0 1	9		EM						\	i	r	z		I	R	Z	9
1 0 1 0	10 (A)					[	]	¦	:								
1 0 1 1	11 (B)	VT				.	$	,	#								
1 1 0 0	12 (C)	FF	FS		DC4	<	*	%	@								
1 1 0 1	13 (D)	CR	GS	ENQ	NAK	(	)	_	'								
1 1 1 0	14 (E)	SO	RS	ACK		+	;	>	=						.		
1 1 1 1	15 (F)	SI	US	BEL	SUB	!	^	?	"								EO

APPENDIX A

A1.0 SCOPE

The intent of this Appendix is to provide additional information regarding some of the characteristics of the CDC FMP. Further information can be found in other appropriate specifications. (See Section 2.0, Applicable Documents).

A2.0 SELF-MODIFYING PROGRAMS

The use of self-modifying programs is not allowed. The following rules which would have to be followed illustrate why this must be true.

The following rules apply to all programs:

1. The twenty-four 64-bit words before (having addresses lower than current instruction word) and the thirty-two 64-bit words after (having addresses higher than current instruction word) the current instruction word shall not be modified by the current instruction.

2. The twenty-four instructions before (in terms of order of execution) and the thirty-two instructions after (in terms of order of execution) the current instruction word shall not be modified by the current instruction.

3. The store into Main Memory for the 13, 5F, and 7F instructions may not take place before the execution of the next instruction in sequence. Therefore, if these instructions are used to modify code, it is difficult to guarantee that the store has taken place before the execution of that code. There are three procedures to guarantee that the store has taken place prior to execution of the intended modified code.

   a. The execution of any instruction which references Main Memory with the exception of the 12, 13, 32, 5E, 5F, 7E and 7F instructions. These instructions must be executed between the store instruction which modifies the code and the use of that modified code.

(continued)

A2.0 (Cont.)

b. The execution of the conditional branch feature of the 32 instruction between the store instruction which modifies the code and the use of that modified code.

c. Execution of a load instruction (12, 5E, or 7E) followed by a transmit (78) instruction where the source register for the 78 instruction conflicts with the destination register for the load instruction. These instructions must be executed between the store instruction which modifies the code and the use of that modified code.

The instructions referenced in a., b. and c. above must be executed from addresses at least four swords before or at least three swords after the modified code.

A3.0 INSTRUCTION STACK

Each machine has a different size instruction stack thus program optimization must be approached with different parameters. Further information is contained in the appropriate execution timing specification.

Number of Words in Instruction Stack

CDC STAR-1B	1	64-bit word
CDC STAR-100	32	64-bit words
CDC STAR-100A	128	32-bit words
CDC FMP	128	32-bit words

A4.0 N/A

A5.0 VECTOR FORMATS

In the CDC FMP, a vector is defined as a contiguous set of bits, bytes or floating-point operands. The contiguous set of bits or bytes is called a string, while the contiguous set of floating-point elements is called an array.

(continued)

A5.0 (Cont.)

Operands are used in the following vector formats:

Array - a counted, variable-length, contiguous, floating-point operand field. Vector operations can be performed on defined fields consisting entirely of 32-bit operands or entirely of 64-bit operands.

Index List - a counted data array of integer values in floating-point format.

A6.0 INVISIBLE PACKAGE

A6.1 Contents of the Invisible Package

The CDC FMP performs as specified with an addition. Bit 12 of word 8 contains the stall bit. The stall bit is a "1" if no data was processed during the last job time-slice that resulted in the preparation of the invisible package.

A6.2 Program Address Register

The only requirement on the program address stored into the first location of the invisible package when program interruption has occurred is that the computer be able to restart the job from the same point at which the interruption occurred.
The following table is included for information only.

(continued)

A6.2 (Cont.)

Interrupt Condition	Program Address Stored In Invisible Package
Exit force instruction at address A in Job Mode.	A + 20 (16)
Illegal instruction with function code less than 80 (16) at address A in Job Mode.	A
Illegal instruction with function code greater than or equal to 80 (16) at address A in Job Mode.	A

A7.0 DATA FLAGS

A7.1 Soft Interrupt Bit

Monitor software can set bit 35 of a job's data flag branch register while the register is stored in the job's invisible package. If, after exchanging back to job mode, bit 35 and its corresponding mask bit (bit 19) are set, a normal data flag branch occurs following completion of the current instruction.

A7.2 Free Data Flags - Bits 56 and 57

The following are the definitions for free data flags 56, 57, and 58.

Bit 56 - A CPU gate associated with the maintenance station monitoring counters (See Section 3.6.4.1.1 of Eng. Spec. 10354637).

Bit 57 - A CPU gate associated with the maintenance station monitoring counters (see Section 3.6.4.1.1 of Eng. Spec. 10354637).

Bit 58 - Not used on CDC FMP.

A7.3 Data Flag Branch

The automatic data flag branch can occur up to 35 instructions after the instruction which caused it. The point at which the branch occurs can vary between executions of the same program as a result of the asynchronous I/O activity affecting the load/store operations.

The following points pertain to the use of the data flag register:

1. The contents of the DFR as stored into the register file by a 38 instruction will reflect all previous activity on it. Also, activity prior to the 38 instruction will not affect the new contents of the DFR.

2. ADF8's caused by a 38 instruction or any instruction previous to it may occur after the next one or two instructions, but no later.

(continued)

A7.3 (Cont.)

3. Sampling or altering a data flag bit with a 33 instruction may occur out of sequence with a previous pipeline instruction up to 35 instructions earlier.

4. If a 33 instruction alters a bit which causes an ADFB, the branch may occur up to two instructions later, regardless of the fact that all pipeline instructions previous to it may have finished. Again, if the ADFB is also contingent on the completion of a pipeline instruction, the automatic data flag branch may occur up to 35 instructions after the instruction which caused it.

When registers 1, 2 or 4 in the FMP register file are altered by an instruction, and this instruction is followed by an automatic flag branch or illegal monitor instruction branch, the store operation may happen out of sequence with the branch operation. Thus, for example, if a 7E instruction loads register 4 with a certain value, and this instrucion is followed by an illegal monitor mode instruction, the automatic branch will be to the address specified by either the old or new contents of register 4, depending on the timing of the 7E and the instruction stream.

A8.0 ADDRESS DISCONTINUITIES

When addressing non-existent areas of memory the FMP will generate an operand abort.

A9.0 EXTERNAL INTERRUPT BIT ASSIGNMENT

The following chart describes the external interrupt bit assignments.

Bit	CDC FMP
0	I/O Channel 0
1	I/O Channel 1
2	I/O Channel 2
3	I/O Channel 3
4	I/O Channel 4
5	I/O Channel 5
6	I/O Channel 6
7	I/O Channel 7
8	I/O Channel 8
9	I/O Channel 9
10	I/O Channel 10
11	I/O Channel 11
12	I/O Channel 12
13	I/O Channel 13
14	I/O Channel 14
15	I/O Channel 15
16	Monitor Interval
17	Non-Existent
18	^
19	!
	!
20	!
21	!
22	v
23	Non-Existent
24	Non-Existent
.	^
.	!
.	!
.	!
.	!
.	!
.	!
.	v
39	Non-Existent

A10.0 04 4 64 NT BREAKPOINT - MAINTENANCE

The breakpoint instruction transfers R to the breakpoint register. The breakpoint register is used as a maintenance and program debugging aid.

	Usage Bits	Breakpoint Address		
0	8 9 15	16	58 59	63

Bits 0-8 and 59-63 are not used.

The breakpoint address is compared with various addresses such as the current instruction address, READ 1 and READ 2 operand addresses, etc. If the breakpoint address matches one of these addresses and the proper usage bit is set, bit 47 of the data flag branch register is set indicating a breakpoint. Any combination of usage bit is permissible, therefore, the breakpoint address can be checked against any or all of the addresses listed below. The breakpoint register is part of the invisible package of a job.

Breakpoint Usage Bits

Bits 9-15 are breakpoint usage bits where if:

a. Bit 9 is set, breakpoint on half-word contents of the program address register (P) just after the execution of the instruction at that location.

b. Bit 10 is set, breakpoint on the READ 1 operand address for vector, or the read operand on random addressing instructions.

c. Bit 11 is set, breakpoint on the READ 2 operand address for a stream instruction.

d. Bit 12 is set, breakpoint on the WRITE 1 address for a stream instruction or the write operand on a random addressing instruction.

e. Bit 13 is set, breakpoint on the READ 3 control vector or operand address (mask) for a stream instruction.

(continued)

A10.0 (Cont.)

f. Bit 14 is set, breakpoint on the READ 1 order vector address.

g. Bit 15 is set, breakpoint on the READ 2 order vector address.

Breakpoint Compares

1. When in job mode or monitor mode, addresses are compared with breakpoint. Since the monitor program does not have an invisible package, the breakpoint register must be set up each time the monitor program is entered. The breakpoint register is automatically cleared to zero during the exchange to the monitor.

2. Program address compares are made on half-word boundaries, and all other compares are made on sword boundaries.

A11.0 06 7 NA MN FAULT TEST - MAINTENANCE

This instruction is used to set up modes which modify certain logic functions in the CPU in order that the fault sensing circuitry may be checked out.

The instruction is only enabled when bit 13 of word 8 in the job's invisible package is set (Refer to Eng. Spec. 10354637); if this bit is a zero, the 06 instruction will act as a no op instruction.

The instruction is always enabled during monitor mode.

The modes are set up by executing this instruction with a "1" in the appropriate R designator bit and are cleared by executing the instruction with a "0" in the same bit location.

SECDED FAULTS

The test is initiated by executing an 06 instruction with any combination of ones in bits 9 through 15 of the instruction (R designator field) to complement the respective checkword bits of all half-words stored in Main Memory via the READ 3 bus. By appropriate selection of data bits and complementation of checkword bits when writing in memory, one should be able to generate SECDED faults on all Read buses. This should allow complete checking of the Read SECDED hardware and also the fault recording hardware for type and address of the fault.

The forced complementing of the checkword bits is discontinued by executing an 06 instruction with bits 9 through 15 of the instruction (R designator field) set to zero.

(continued)

A11.0 (Cont.)

The S and T designators are undefined.

No interrupts or I/O memory requests can be allowed during the execution of these tests.

A12.0 FLOATING-POINT SUBTRACT

The Instruction Descriptions Specification (paragraph 3.1.4.6.3) defines the floating-point subtract operation as "performed by complementing the coefficient of the subtrahend and performing a floating-point addition operation". It is further added "that the complement of an 8000 0000 0000 coefficient is 4000 0000 0000 with one added to the value of the exponent associated with the coefficient".

The hardware used for floating add or subtract operations has an extra (or extended) coefficient sign bit. This means that the complementation of an 8000 coefficient is handled without the specified right shift of one and increase of the exponent by one. This will cause a result (although not mathematically incorrect) which may differ from the specified result when the following conditions are met:

1. The operand of the pair having the large exponent (OR either of the two operands if their exponents are equal) must have a coefficient of 8000 ---

2. This operation must require this same operand to be complemented due to
   a. being the subtrahend in a subtract operation OR
   b. sign control in either a subtract or an add operation ----

3. The "other" operand must have a negative coefficient.

(continued)

A12.0 (Cont.)

Example I A - B

A 60 F F F 0 0 0
B 64 8 0 0 0 0 0

	CDC FMP						Instruction Specification					
	Extra Sign Bit											
	↓											
Complement B B	-64	(1)	8	0	0	0	0	0	64	8	0	0	0	0	0
$\overline{B}$	→64	(0)	8	0	0	0	0	0	65	4	0	0	0	0	0
Align operand with smaller exponent	-60	(1)	F	F	F	0	0	0	-60	F	F	F	0	0	0
	→64	(1)	F	F	F	F	0	0	→65	F	F	F	F	8	0
Add A plus complement of B A	64	(1)	F	F	F	F	0	0	65	F	F	F	F	8	0
$+\overline{B}$	64	(0)	8	0	0	0	0	0	65	4	0	0	0	0	0
	6	(0)	7	F	F	F	0	0	65	3	F	F	F	8	0
	64		7	F	F	F	0	0	65	3	F	F	F	8	0

Example II A - B

A 50 F F F 0 0 0
B 6F 8 0 0 0 0 0

	CDC FMP						Instruction Specification					
	Extra Sign Bit											
	↓											
Complement B B	-6F	(1)	8	0	0	0	0	0	6F	8	0	0	0	0	0
$\overline{B}$	→6F	(0)	8	0	0	0	0	0	70	4	0	0	0	0	0
Align operand with smaller exponent	-50	(1)	F	F	F	0	0	0	50	F	F	F	0	0	0
	→6F	(1)	F	F	F	F	F	F	70	F	F	F	F	F	F
Add A plus complement of B A	6F	(1)	F	F	F	F	F	F	70	F	F	F	F	F	F
$+\overline{B}$	6F	(0)	8	0	0	0	0	0	70	4	0	0	0	0	0
	6F	(0)	7	F	F	F	F	F	70	3	F	F	F	F	F

(continued)

A12.0 (Cont.)

If this operation is a subtract upper, the specified result is indefinite (with the appropriate data flags) while the CDC FMP result did not overflow. If this operation were a subtract normalized, note the following:

	CDC FMP	Instruction Specification
Result of Subtract Upper	6F (0) 7 F F F F F	70 3 F F F F F
Normalize the Upper Result (3.1.4.7) shifting zeros in from the right	6F 7 F F F F F ^	6F 7 F F F F E ^

APPENDIX B

FMP FUNCTIONAL COMPUTER SPECIFICATION

(R) CDC FLOW MODEL PROCESSOR

Functional Computer

Specification

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

1.0 SCOPE

The CDC Flow Model Processor (FMP) Functional Specification is intended to provide information of the following types to either the user or maintenance personnel.

- Information that may be obtained about computer/program operation via the Maintenance Control Unit (MCU).
- Changes in mode or operation internal to the FMP that may be made via the MCU or program that are not specified in the FMP Instruction Specification.
- Information concerning computer operation that is of value in debugging software/hardware or in program optimization.

This specification is not intended to provide information as to how a unit performs its specified tasks such as would normally be found in a Theory of Operation.

2.0 APPLICABLE DOCUMENTS

10354636 CDC Flow Model Processor Instruction Specification

3.0 REQUIREMENTS

3.1 General Functional Description

The Flow Model Processor (FMP) is an extremely high speed computational system designed specifically for the solution of flow simulations related to the design and construction of aerodynamic bodies. It is based, in part, on the Control Data STAR-100 architecture, with both Main Memory and Scalar Processor design taken from the STAR-100 family. The resulting basic structure is augmented by a massive

(continued)

3.1 (Cont.)

Backing Storage capability (up to 256 million words of CCD memory), a Swap Unit (to perform exchanges between the Backing Store and the Main Memory), a Map Unit (for gathering vector data from Main Memory and storing results), and a Vector Unit (for the computational portions of the problem solution). Figure 3.1-1 shows the overall block diagram of the FMP.

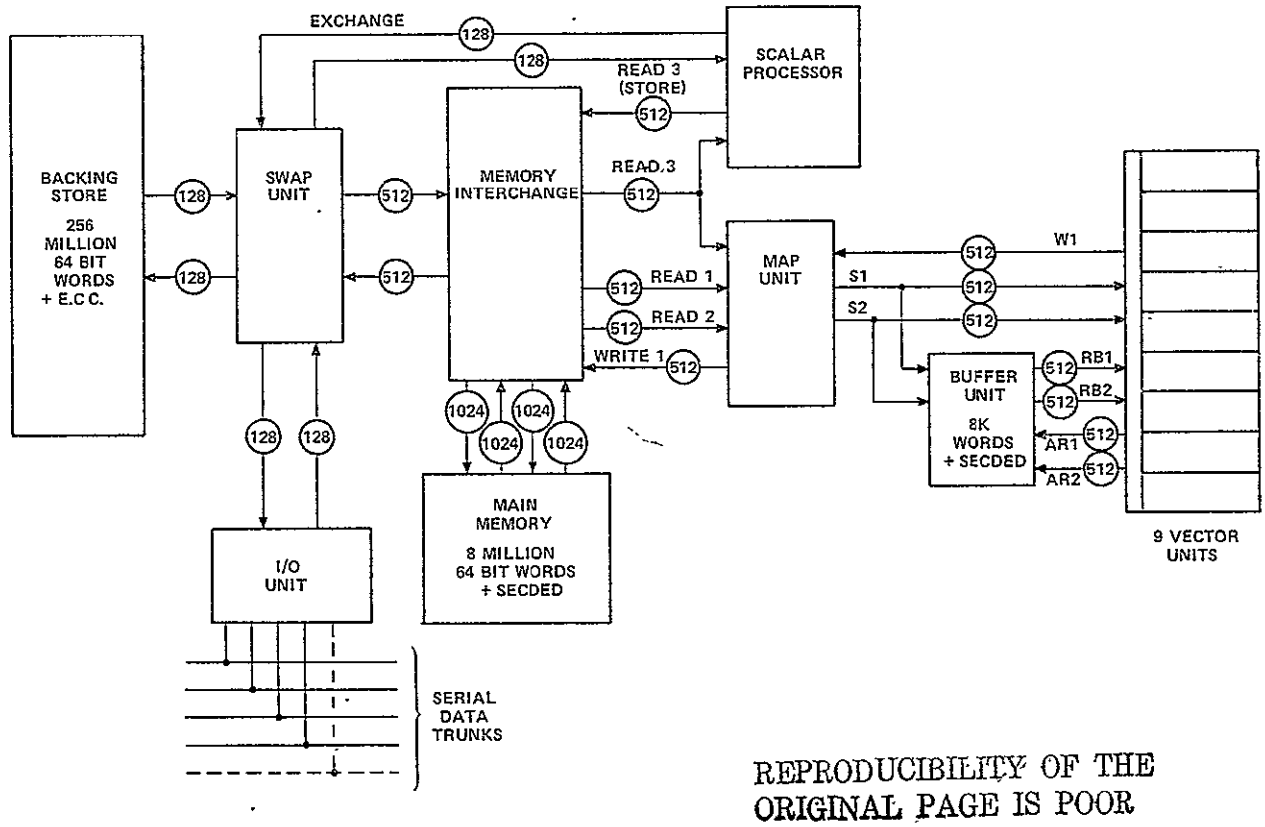


Figure 3.1-1 Basic CDC FMP Configuration

(continued)

3.1 (Cont.)

Data and programs are entered into the FMP via the Input/Output ports attached to the Backing Store. Once the various fragments of a job are aggregated in the Backing Store, and the FMP is idle, the job is "rolled" into the Main Memory via the Swap Unit. Certain portions of the computations, all of the bookkeeping and all of the FMP's overall control are accomplished in the Scalar Processor . It is this processor that interprets the instruction stream, acts on those instructions which it can, and distributes the remaining instructions to the appropriate attached units (Map, Swap, Buffer, Vector).

The FMP is designed to operate at a minor clock cycle rate of ten nanoseconds, with all data transfers, and all pipeline segments capable of clocking a new data quantity (32, 64, 128, 512, or 2048 bits wide) every minor cycle. The maximum rate of arithmetic results production in the 8 sector pipelines then becomes 3 (operations peak rate)*2 (32-bit results per pipeline)*8(pipelines)=48 per minor cycle of 10 nanoseconds=4.8 billion floating-point operations per second.

The Scalar, Map, Swap, and Vector Units are capable of operating simultaneously so that a majority of bookkeeping and data mapping (reorganization functions) can be overlapped with the computation. This enables the effective rate of problem solution to approach 60% of the peak rate, or 2.8 billion operations per second, which exceeds the original objectives established for Navier-Stokes solutions for flow field simulations.

Unlike the STAR-100, the FMP is designed for monoprogramming of computational jobs, thus there is no virtual memory mechanism. All user jobs are given the use of the entire eight million words of Main Memory minus the first 65K words which are reserved for the FMP monitor. This monitor area cannot be accessed by the job mode programs. A series of several monitor mode instructions permits the management and allocation of the Backing Store, as well as control communications with the I/O processors attached to the FMP. These I/O processors, called PDCs (Programmable Device Controllers), are capable of intelligent control of

(continued)

3.1 (Cont.)

the I/O trunks (up to four attached to each PDC) and intelligent communications with the monitor, as well as providing 200-megabit data transfer rates between the Backing Store and the trunks, and from 50 to 100-megabit transfer rate on the actual coax trunks themselves.

The instruction set for scalar operations is a compatible subset of the STAR-100 family which supports most STAR software, with the addition of a few operations made necessary by the unique I/O and Backing Store configuration provided on the FMP. The Map Unit provides execution capability of the STAR-100 "Iverson" operators of vector MASK,MERGE,COMPRESS and SCATTER/GATHER while the Vector Unit, in cooperation with the Map Unit, performs the "Iverson" SELECT, SEARCH, and SEARCH INDEXED LIST operations. The Map Unit is capable of performing memory to memory operations while the Vector Unit is performing buffer to buffer operations independently. In addition, several combinations of Memory, Buffer, and Vector Unit operations may be invoked.

The Vector Unit performs the add, subtract, multiply, divide operations commonly found on most processors, in addition to a series of linked and macro operations providing combinations of additions and multiplications every minor cycle. The set of linked operations chosen were based on the characteristics of flow-model simulations that have been analyzed by Control Data Corporation. In addition to the simple combinations of add/subtract and multiply, the functions SUM,PRODUCT,SUM OF PRODUCTS, and PRODUCT OF SUMS are included for matrix computations.

To ensure the reliability and maintainability of the FMP, a number of error checking and recovery facilities are built in, as well as a group of maintenance functions which can be invoked by a designated computer attached to one or more of the I/O trunks. Single Error Correction, Double Error Detection (SECDED) is carried through all data trunks up to the functional unit actually using the data. Checking for errors is done at several points in the data path (for example at the Memory, at the Map Unit, and at the Vector Unit) so that faults can be quickly isolated, while the error correction is applied at the point where the data is used, for example the input stream of the Vector Unit.

(continued)

(to be supplied later)

Figure 3.1-2 CDC FMP Floor Layout

3.2 Scalar Processor

The Scalar Processor is physically contained in a cabinet attached to the Vector Processor cabinet. Main Memory is attached to the Scalar Processor cabinet in order to reduce transfer delays and gain performance.

The Scalar Processor has synchronous internal logic with a clock period of 10 nanoseconds and is implemented using LSI circuits. A block diagram of the functional components of the Scalar Processor is shown in Figure 3.2-1.

The CDC FMP instruction control is contained in the Scalar Processor. The Instruction Issue Unit consists of two parallel parts, one for the monitor program and one for the job program; it receives and decodes all instructions from Main Memory. A semiconductor instruction stack provides buffering for eight swords for the job and one sword for monitor (512 bits per sword) each of which can contain up to 128 32-bit instructions or 64 64-bit instructions or a mixture. The job instruction stack can contain up to 6 discontiguous swords with two swords lookahead. The Read Next Sword (RNS) portion of the RNS/Branch Unit provides the control for loading the instruction stack. The Branch portion performs branch condition testing and executes the branch instructions.

The Instruction Issue Unit is pipelined and is capable of issuing instructions at the rate of one instruction every 10 nanoseconds. The Instruction Issue Unit decodes all instructions and directs decoded stream instructions to the appropriate processor for execution. Thus, with independent vector and scalar instruction controls operating on a single instruction stream, the Scalar Processor can execute scalar instructions in parallel with most stream instructions.

The instruction stack contains eight superwords (swords) containing 512 bits each. If an instruction is referenced which is not presently in the stack, the Issue Unit is halted and a memory request is made for the word containing the required instruction. The sword thence brought from memory

(continued)

3.2 (Cont.)

must replace one of the swords already in the stack. The sword that is "thrown away" or overlaid by the incoming sword is the least recently used (LRU) sword. Thus if words numbered consecutively 0 through 7 have been executed without any intervening branches, word 8 (required by the next consecutive instruction) would be brought from memory and overlaid in the stack in the position originally held by sword number 0 which, in this case, is the LRU sword.

(continued)

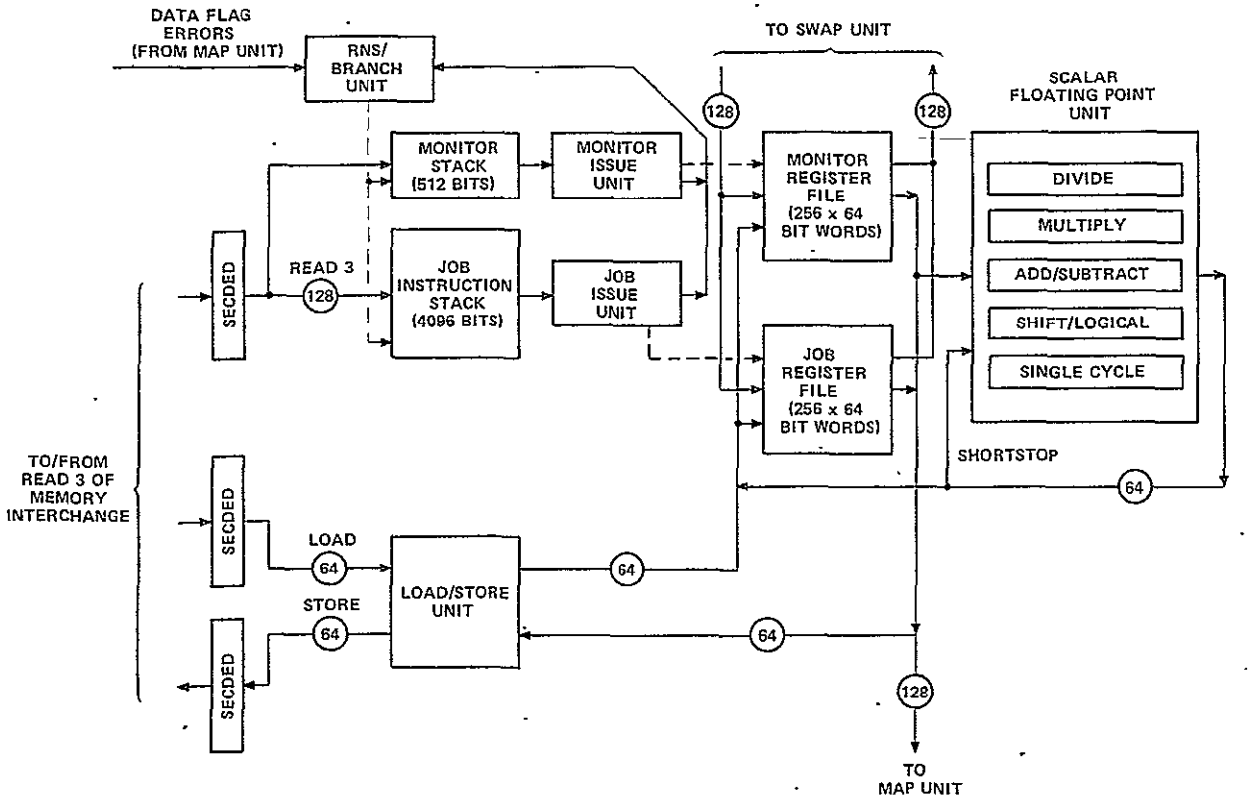


Figure 3.2-1 Scalar Processor Block Diagram

3.2 (Cont.)

The Load/Store Unit provides special handling of the Load and Store instructions. The unit acts as a pipeline and is capable of accepting a new request rate of one load every minor cycle or one store every two minor cycles, provided a memory busy or register file write-bus busy does not occur. A circular buffer containing six registers provides buffering for up to six load requests, or three store requests, or a mixture of loads and stores.

The Load/Store Unit is capable of loading a randomly accessed word of data from Main Memory into the Register File in 150 nanoseconds after reading the base address and item count of the data. This time assumes a memory busy or register file write-bus busy does not occur. A memory busy would add up to 40 nanoseconds to the load time.

The Scalar Floating-Point Unit contains completely independent functional elements to attain high scalar performance. The following are the times in nanoseconds to produce a 32-bit or 64-bit result in each functional element. These times corresoond to the shortstop times. Shortstop is the process by which a result from any arithmetic element may be returned directly to either input of any arithmetic element. This occurs in parallel with the storing of the result in the Register File. Shortstop eliminates the time necessary to store the result in the Register File and then retrieve it for use in the next arithmetic operation.

Add/Subtract Pipe	50 ns
Multiply Pipe	50
Shift/Logical Pipe	40
Single Cycle Pipe	10
Divide/SQRT/Convert Element	240

The pipe elements are segmented and capable of accepting new operands every 10 nanoseconds. The Divide/SQRT/Convert element must complete each operation before a new one can begin. All elements are capable of being shortstopped. The Scalar Processor contains a semiconductor Register File which provides 256 64-bit registers for use in instruction and operand addressing, indexing, field lengths, and as

(continued)

3.2 (Cont.)

source and destination registers for scalar instruction operands and results. The Register File is capable of two reads and one write every 10 nanoseconds.

3.2.1 Scalar Processor Error Checking

The basic design of the FMP Scalar Processor is based on the design of the STAR-100A and STAR-100B Scalar Processors. In these designs (already being implemented) there exists a moderate amount of error checking on busses:

a. SECDED - All data busses in and out of the Scalar Processor carry seven bits of single error correction, double error detection code bits for each 32 bits of data. The data busses are the Load/Store data bus (64 bits wide), the Instruction Read data bus (128 bits wide), the Register File exchange path (128 bits wide each way), and the Register File data bus to the Map Unit (128 bits wide).

b. Parity - All microcode memories in the Issue and Floating-Point Units contain parity bit checking. The microcode carries a parity bit from the time it is assembled on a front-end processor, until it is read during execution in a given unit. A parity fault causes an immediate stoppage of the CPU, and an error flag to be sent to the Maintenance Control Unit (MCU). The instruction stack contains parity information in like manner.

c. Illogical function - Communication between the various functional elements of the Scalar Processor is performed by sequences of microcode generated function codes, which are decoded at the receiving end by microcode. Sufficient entropy has been included in the function code scheme to permit some detection of internal control signal failures.

(continued)

3.2.1 (Cont.)

The Scalar Processor for the FMP is being examined closely to see if parity (1 bit for each 8 bits of data) can be included in all internal data trunks and functional elements. Checking of the arithmetic elements (with the exception of the multiply element) can be accomplished by this method, thus ensuring a high degree of integrity for this unit. The penalty for this measure however can be a seriously reduced performance for some scalar operations (particularly where recursion is invoked). An examination of the tradeoffs of cost, performance, and reliability will have to await more detailed design and analysis of the Scalar Processor.

3.2.2 SECDED (Single Error Correction Double Error Detection)

The SECDED error information is logged by the Maintenance Control Unit (MCU). The information logged is syndrome word, single error, double error, Read bus code, and CPU word address bits 37-58.

SECDED ERROR INFORMATION

1. SYNDROME BITS - These seven bits generated by the error correcting code . The 39 unique syndrome words for single bit errors are listed on Table 3.2-1. Of these 39 (odd bit) syndrome words, only the 32 data bit codes will toggle a bit when error correction is enabled. Other odd bit codes latched in SECDED that differ from the 39 unique syndrome words will be flagged by the MCU as a multiple odd bit error. Double error syndrome words have an even number of bits.

2. SINGLE ERROR - Bit 5 of channel AT88 (see section 3.6.1) will set if there is a single error not preceded by a double error.

3. DOUBLE ERROR - This MCU display register will set unconditionally on a double error.

4. SECDED FAULT BUS CODE - These MCU display registers define the read bus on which the SECDED error occurred.

(continued)

3.2.2 (Cont.)

			Read Bus
CODE	0	=	I/O
CODE	1	=	R1
CODE	2	=	R2
CODE	3	=	R3
CODE	4	=	Scalar
CODE	5	=	RNI
CODE	6	=	Swap

The error logging priority for simultaneous SECDED errors on multiple buses is:

1. RNI
2. SCALAR
3. R2
4. R1
5. SWAP
6. I/O
7. R3

5. HALF-WORD ADDRESS (Bits 57,58) - These address bits decode the four 32-bit groups within a quarter sword. The error logging priority for simultaneous SECDED errors more than one half-word is in order: HW0, HW1, HW2, and HW3.

6. CPU WORD ADDRESS (Bits 37-56) - These address bits indicate the following:

Bit	
37-39	Select 1 of 8 Memory Chips
40-49	Select 1 of 1K Memory Cells
50	1024K Select
51	512K Select
52-54	Bank Select
55-56	Quarter Sword Select

7. LATCHED ADDRESS BITS (37-58) - In SECDED these address bits are always the physical CPU Word Address Bits.

(continued)

3.2.2 (Cont.)

TABLE 3.2-1 UNIQUE SYNDROME WORDS FOR SINGLE BIT FAILURES

Bit	Data	Syndrome Word
0	80000000	70
1	40000000	68
2	20000000	58
3	10000000	64
4	08000000	54
5	04000000	7C
6	02000000	7A
7	01000000	76
8	00800000	1C
9	00400000	1A
10	00200000	16
11	00100000	19
12	00080000	15
13	00040000	1F
14	00020000	5E
15	00010000	5D
16	00008000	07
17	00004000	46
18	00002000	45
19	00001000	26
20	00000800	25
21	00000400	67
22	00000200	57
23	00000100	37
24	00000080	61
25	00000040	51
26	00000020	31
27	00000010	49
28	00000008	29
29	00000004	79
30	00000002	75
31	00000001	6D
32	Check Bit 0	40
33	Check Bit 1	20
34	Check Bit 2	10
35	Check Bit 3	08
36	Check Bit 4	04
37	Check Bit 5	02
38	Check Bit 6	01

(continued)

3.2.2 (Cont.)

The syndrome word is latched if the bit shown in the data pattern in Table 3.2-1 is in error. For example, if and only if, bit 0 failed on any data pattern, then the syndrome word would be 70.

The SECDED error latching hardware has two basic modes of operation - Mode 1 and Mode 2.

Selection between the two modes is accomplished through the MCU/CPU Maintenance Line called SELECT SECDED ERROR LOG MODE TWO.

For both modes in the event of simultaneous SECDED errors, the information to be latched is dependent on the relative priority of the data buses or half-words which contain the errors. All information will be correct for the error selected. It is possible in both modes to encounter a single and double error simultaneously and latch the single error. The double error flag will set unconditionally. Therefore, if the double error flag is set, the syndrome bits must be checked to determine if single or double error was latched. In the event the single error flag is set, and no double error, the error will be a single error.

Mode 1

The first error to occur after a master clear or error clear will have its error information latched. The information will be correct in all cases, regardless of subsequent errors. If a double error follows a single without an error clear, the double error information will be lost.

Mode 2

Operation in Mode 2 is the same as in Mode 1 except for the following enhancement: An attempt will be made to latch the error information for the first double error encountered whether or not a single error has previously been latched.

(continued)

3.2.2 (Cont.)

As in Mode 1, the double error flag will set unconditionally when a double error is encountered. However, other aspects of Mode 2 operation are less certain. The conditions which may result are listed below:

Case 1

In the event of simultaneous errors, Mode 2 is the same as Mode 1. If the double error flag is set, the syndrome bits must be checked to determine if a single or double error was latched.

Case 2

If the SECDED checker encounters a single or several single errors, and is absent of the double error flag, then the error information will be that of the first single error. All information is correct as in Mode 1.

Case 3

If the SECDED checker encounters a double followed by other double or single errors then the error information will be that of the first double error. All information is correct as in Mode 1. However, the MCU cannot be distinguished from Case 1 with the doubled error latched, so the syndrome bits must be checked.

Case 4

If the SECDED checker encounters a single error and N minor cycles later (N<8) a double error is encountered: Address bits 37 thru 54 for either the single or double error may be latched; bits 55 and 56 are indeterminate: and the remaining error information would be that of the double error.

Case 5

If the SECDED checker encounters a single error and N minor cycles later (N>8) a double error is encountered, the double error information will be correct. However, the MCU cannot distinguish this case from Case 4.

(continued)

3.2.2 (Cont.)

Case 6

If the SECDED checker encounters a double error and one or more minor cycles later a single or double error is encountered, this is simply Case 3. The first double error information will be latched.

Mode 2A Double Error Log

This mode is electronically identical to Mode 2. The difference is strictly operational. Specifically, after a master clear or error clear, the MCU deliberately creates a single error using the maintenance function to toggle a check bit. This error is not cleared, and effectively blocks detection of all subsequent single errors. Consequently, when the MCU detects the double error flag, it knows that this is Case 5 and the error log information is correct for that double error.

BLOCK WRITE ENABLES

The MCU has the capability to enable block write enable if a SECDED error occurs. There are two options which can be selected depending on SECDED error mode.

1. With Mode 1, the write enables will be blocked when SECDED receives its first single or double error.

2. With Mode 2, the write enable will be blocked when SECDED receives its first double error.

COMPLEMENT I/O CHECKWORD BITS

This maintenance feature enables the MCU to toggle the Write I/O checkword bits before write into memory. Toggling the 128 combinations on each half-word of the six Read Data Buses allows checkout of the SECDED checker.

(continued)

3.2.2 (Cont.)

GENERAL USAGE

Mode 1 is a good SECDED latch design for a memory with low error rate. All error log information is correct. However, it will not latch the double error if it follows a single error within the cycle time of the MCU.

Mode 2 is a better SECDED latch design for a memory with a high error rate. All single errors latched are correct, and all double errors following a single error by greater than eight minor cycles (80 ns) are correct. A double error occurring before a single error is also latched correctly.

Mode 2A is a double error logging system for use if single errors are to be ignored. This mode will miss the double error only if there is a simultaneous single error with higher latching priority. If this condition would occur, a diagnostic requesting only one bus will get around the bus priority. If the diagnostic fails and still latches a single error, then the double error is in a lower priority half word.

3.2.3 Associative Unit

N/A

3.2.4 Instruction Issue/Decode

All instructions are read from memory by the Scalar Processor and decoded for subsequent issue. This is accomplished in the Issue Unit which is composed of two parts, one for monitor and one for job. After decoding an instruction, the Issue Unit issues it to the unit responsible for its execution: the Vector Unit, the Swap Unit, or the Scalar Processor itself. Responsibilities for all instructions are shown in table 3.2-2.

These units are essentially independent of one another and can execute instructions in parallel. The remainder of section 3.2 provides additional information on Scalar Processor operation. Section 3.3 describes the Vector Unit and section 3.7 covers the Swap Unit.

(continued)

TABLE 3.2-2 INSTRUCTION RESPONSIBILITY

Second Digit of Instruction Code	First Digit of Instruction Code															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	S	S	S	S	S	S	S	S	I	I	I	S	I	I	I	I
1	S	S	S	S	S	S	S	S	I	I	I	S	I	I	I	I
2	S	S	I	S	S	S	S	S	I	I	I	S	I	I	I	I
3	I	S	I	S	I	S	S	S	I	I	I	S	I	I	I	I
4	S	I	I	S	S	S	S	S	I	I	I	S	I	I	I	I
5	I	I	I	S	S	S	S	S	I	I	I	S	I	I	I	I
6	S	I	I	S	S	X	S	S	I	I	I	S	I	I	I	I
7	I	I	I	S	I	I	S	S	I	I	I	I	I	I	I	I
8	S	I	I	S	S	S	S	S	I	I	I	I	I	I	I	I
9	S	I	I	S	S	S	S	S	I	I	I	I	I	I	I	I
A	S	I	I	S	I	S	I	S	I	I	I	I	I	I	I	I
B	I	I	S	S	S	S	S	S	I	I	I	I	I	I	I	I
C	I	I	S	S	S	S	S	S	I	I	I	I	I	I	I	I
D	I	I	S	S	S	S	S	X	I	V	I	I	S	I	I	I
E	S	I	S	S	S	S	S	S	I	V	I	S	S	I	I	I
F	I	I	S	S	S	S	S	S	I	V	I	S	I	I	I	I

S - Executed within the Scalar Processor (Note that Data Flag information will be passed to the Data Flag Register in the Vector Processor for appropriate instructions).

V - The Scalar Processor initiates the Vector Processor to execute portions (or all) of the instructions.

I - Illegal instruction.

X - Executed in the Swap Unit.

3.2.5 Register File

The Register File of the FMP contains 256 64-bit words. This Register File is capable of accomplishing two read operations and one write operation every 10 nanosecond minor cycle. In addition, the Register File can be exchanged at the rate of two registers in and two out every minor cycle. A complete swap of the Register File is accomplished in 256 10-nanosecond minor cycles plus set-up time.

The FMP has 16 Result Address Registers (RAR) used to conflict check each scalar instruction ready for issue against register file addresses that are to be written by an already issued scalar instruction. If a conflict exists, the action taken depends on whether the needed result can be shortstopped or not (see sections 3.2 and 3.2.8 for additional information on shortstop). If shortstop is possible, the instruction is issued at the appropriate time and instruction issue continues. If shortstop is not possible (e.g., the result of a previous load is needed), issue stops.

The RARs are set sequentially from the result register designators of issued scalar instructions. They are cleared when the result is written into the Register File.

3.2.6 Branch/Instruction Stack

The Branch instruction execution times may be found in section 3.9 of this Specification.

The instruction stack implemented in the FMP accommodates up to 8 swords (512 bits per sword), 6 of which may be discontiguous. To sustain the instruction rate a two-sword "lookahead" will be done by reading the two swords following the one being executed. Issue will not be blocked if the swords following are not in the stack.

An address is maintained for each of the eight swords so that out-of-stack branches may be taken without voiding the entire stack. For instance, it would be possible to call a subroutine of up to 3 swords (48 instructions/32 bits each) several times from a three sword instruction stream and never branch out-of stack after the first branch which loads the subroutine into the stack.

(continued)

3.2.6 (Cont.)

Self-Modifying Programs

The material describing the restrictions concerning self-modifying programs will be added at a later date. This will be similar to that found in paragraph A2.0 of Eng. Spec. 10354636.

3.2.7 Load/Store Unit

The Load/Store Unit executes the 12, 13, 32, 5E, 5F, 7E and 7F instructions. There are six address registers in the Load/Store Unit which enable requests to be stacked and executed in the proper order. The 12, 5E and 7E instructions each require one register and can be executed (in the absence of memory conflicts) at the rate of one load per minor cycle. The 5F and 7F instructions each require two address registers and can be executed at one store per two minor (10 ns) cycles. The 13 and 32 instructions each require two address registers which are then busy for 17 minor cycles.

The Load/Store Unit is thus capable of streaming Load/Store instructions (other than the 13 and 32) at one minor cycle per load and two minor cycles per store assuming no Memory or Register File conflicts. For example, a stream of N loads will execute in N + 14 minor cycles from the issue of the first load until the operand from the last load available in the Register File. A stream of N stores will execute in $2N + \lfloor \frac{N}{13} \rfloor$ minor cycles from issue of the first store until issue of the last store.

3.2.8 Scalar Floating Point

The FMP has an arithmetic unit dedicated to scalar (non-vector) operations. This Scalar Floating-Point Unit will be divided into five separate functional elements: one each for add/subtract, multiply and logical, a single cycle element for the add/subtract address and transmit type instructions, and one combining divide, square root and convert.

(continued)

3.2.8 (Cont.)

All elements of the arithmetic unit are separately and independently controlled to allow concurrent operation. However, only one operand pair is issued to the arithmetic unit each minor cycle so this becomes the limiting factor determining the result rate from concurrent operations.

The first four are effectively segmented pipeline elements which accept a new pair of operands every minor cycle. They each produce a 64 or 32-bit result every minor cycle. The divide - sq.rt. - convert element is not segmented and thus accepts operands only at completion of the previous operation, every 28 minor cycles per 64-bit operand. Using 32-bit operands would approximately double the result rate of the divide - sq. rt. and convert.

Interface Between Scalar Floating Point and Scalar Control Unit

Input Trunks

There are three input trunks to the Scalar Floating-Point Unit. The characteristics of these trunks are outlined in the following description. All input operands are treated as 64 or 32-bit floating-point quantities, except as noted. If an indefinite or machine zero floating-point operand is received, its coefficient will be set to all zeros.

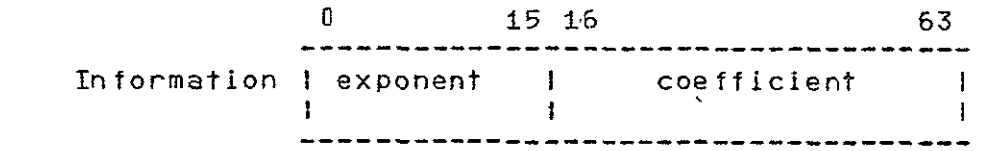
(continued)

3.2.8 (Cont.)

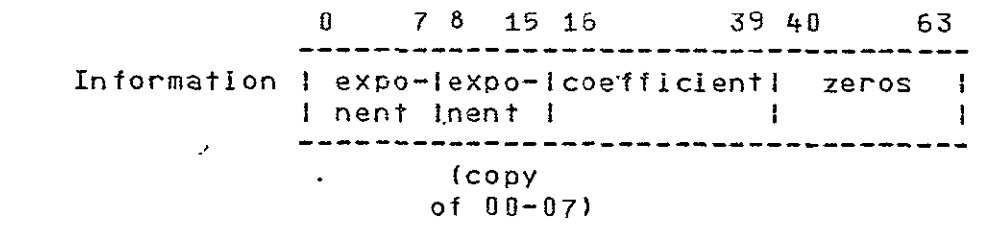
A Input Trunk

This trunk is 64 bits wide. It receives 64 data bits from register location R in the following format:

64-Bit Mode



32-Bit Mode



All bits transferred on this trunk should be held on the trunk for a period of 10 nsec. measured at the input to the Scalar Floating-Point Unit.

B Input Trunk

The B trunk receives data from register location S and is identical to the A trunk.

(continued)

3.2.8 (Cont.)

Control Trunk

The control trunk carries the signals which control the Scalar Floating-Point Unit. It is made up of the following signals:

Control Address

The control address bits are the bits that select the proper set of internal control signals for the floating-point instruction being executed. There is a unique code for each instruction as listed in Table 3.2-3. Using the input data to the Floating-Point Unit as a reference, these control bits must arrive at the floating-point logic 1.5 cycles ahead of the data and be valid for 10 nsec.

Mode Controls

The mode controls are Mode 64 In, Mode 64 Out, G-bits and Divide. The Mode 64 and G-bit lines must lead the input data by 1.0 minor cycles and the Divide signal must lead by 1.5 minor cycles. These should remain up for 10 nsec.

Issue Controls

These controls are S-Shortstop, R-Shortstop, S-Clockgate, R-Clockgate, S-Shortstop Enable, R-Shortstop Enable and Go. These controls all must be valid 1.0 cycles ahead of the data. The Shortstop Enable signals enable the setting or clearing of the Shortstop control flip-flops. The Shortstop signals set or clear signals cause data to be clocked into the floating-point input registers when these signals are a one. The Go signal tells the Floating-Point Unit to begin processing the operands that are in the input registers.

(continued)

3.2.8 (Cont.)

TABLE 3.2-3 INSTRUCTION CODES

INSTR	M64 IN	M64 OUT	CONTROL ADDRESS	G-BITS	DIV.	CYCLE TIME	BUSY TIME	A TRUNK	B TRUNK	OUTPUT CONTROL
10	1	1	01		1	20	17	0	R	DT,DB,DFLG39
11	1	1	02		1	53	50	0	R	DT,DB
20	1	1	10		0		0	R	S	
21	1	1	11		0		0	R	S	
2A	1	1	18		0	3	0	I	R	
2B	1	1	19		0	3	0	I	R	
2C	1	1	1A		0	3	0	R	S	
2D	1	1	1B		0	3	0	R	S	
2E	1	1	1C		0	3	0	R	S	
2F	1	1	1D	G2,G3	0	1	0	0	T	
30	1	1	1E		0	3	0	R	S	
31	1	1	1F		0	1	0	R	+1	
34	1	1	20		0	3	0	R	S	
35	1	1	21		0	1	0	R	-1	
36	1	1	22		0	1	0	CIAR	+20	
38	1	1	23		0	1	0	R	T	
3C	0	0	24		0	5	0	R	S	
3D	1	1	25		0	5	0	R	S	
3E	1	1	26		0	1	0	R	I	
3F	1	1	27		0	1	0	R	I	
40	0	0	28		0	5	0	R	S	DFLG42,43,46
41	0	0	29		0	5	0	R	S	DFLG42,43,46
42	0	0	2A		0	5	0	R	S	DFLG42,43,46
44	0	0	2B		0	5	0	R	S	DFLG42,43,46
45	0	0	2C		0	5	0	R	S	DFLG42,43,46
46	0	0	2D		0	5	0	R	S	DFLG42,43,46
48	0	0	2E		0	5	0	R	S	DFLG42,43,46
49	0	0	2F		0	5	0	R	S	DFLG42,43,46
4B	0	0	30		0	5	0	R	S	DFLG42,43,46
4C	0	0	31		1	30	25	R	S	DFLG41,42,43,46
4D	0	0	32		0	1	0	I	0	
4E	0	0	33		0	1	0	R	I	
4F	0	0	34		1	30	25	R	S	DFLG41,42,43,46
50	0	0	35		0	5	0	0	R	DFLG46

(continued)

3.2.8 (Cont.)

TABLE 3.2-3 INSTRUCTION CODES (Cont.)

INSTR	M64 IN	M64 OUT	CONTROL ADDRESS	G-BITS	DIV.	CYCLE TIME	BUSY TIME	A TRUNK	B TRUNK	OUTPUT CONTROL
51	0	0	36		0	5	0	0	R	DFLG46
52	0	0	37		0	5	0	0	R	DFLG46
53	0	0	38		1	30	26	0	R	DFLG43,45,46
54	0	0	39		0	5	0	S	R	DFLG42,43,46
55	0	0	3A		0	5	0	S	R	DFLG42,46
56	1	1			0	4	0	R	S	
58	0	0	3B		0	1	0	R	0	
59	0	0	3C		0	5	0	0	R	DFLG42,43,46
5A	0	0	3D		0	3	0	0	R	
5B	0	0	3E		0	3	0	R	S	
5C	0	0	3F		0	5	0	0	R	DFLG43,46
5D	0	1	40		0	5	0	0	R	DFLG43,46
60	1	1	41		0	5	0	R	S	DFLG42,43,46
61	1	1	42		0	5	0	R	S	DFLG42,43,46
62	1	1	43		0	5	0	R	S	DFLG42,43,46
63	1	1	44		0	1	0	R	S	
64	1	1	45		0	5	0	R	S	DFLG42,43,46
65	1	1	46		0	5	0	R	S	DFLG42,43,46
66	1	1	47		0	5	0	R	S	DFLG42,43,46
67	1	1	48		0	1	0	R	S	
68	1	1	49		0	5	0	R	S	DFLG42,43,46
69	1	1	4A		0	5	0	R	S	DFLG42,43,46
68	1	1	4B		0	5	0	R	S	DFLG42,43,46
6C	1	1	4C		1	54	49	R	S	DFLG41,42,43,56
6D(1)	1	1	4D		0	4	0	R	S	
6D(2)	1	1	4E		0	3	0	T	0	
6E	1	1	4F		0	3	0	R	S	
6F	1	1	50		1	54	49	R	S	DFLG41,42,43,46
70	1	1	51		0	5	0	0	R	DFLG64
71	1	1	52		0	5	0	0	R	DFLG64
72	1	1	53		0	5	0	0	R	DFLG64
73	1	1	54		1	54	50	0	R	DFLG43,45,46
74	1	1	55		0	5	0	S	R	
75	1	1	56		0	5	0	S	R	
76	1	1	57		0	5	0	0	R	
77	1	0	58		0	5	0	0	R	
78	1	0	59		0	1	0	R	0	
79	1	1	5A		0	5	0	0	R	
7A	1	1	5B		0	3	0	R	0	
78	1	1	5C		0	3	0	R	S	
7C	1	1	5D		0	3	0	R	0	

Note: The 6D instruction requires three references to the Register File; this takes two minor cycles. The "(1)" is the first and the "(2)" is the second.

(continued)

3.2.8 (Cont.)

TABLE 3.2-3 INSTRUCTION CODES (Cont.)

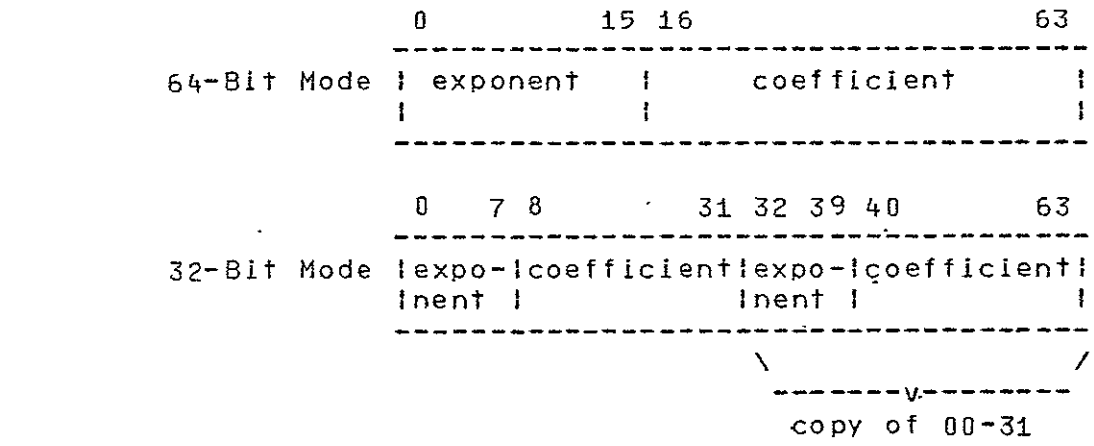
INSTR	M64 IN	M64 OUT	CONTROL ADDRESS	G-BITS	DIV.	CYCLE TIME	BUSY TIME	A TRUNK	B TRUNK	OUTPUT CONTROL
B0,G1=0	1	1	60	G1,2,3,4	0	3	0	A	X	
B0,G1=1	1	1	70	G1,2,3,4	0	5	0	A	X	
B1,G1=0	1	1	61	G1,2,3,4	0	3	0	A	X	
B1,G1=1	1	1	71	G1,2,3,4	0	5	0	A	X	
B2,G1=0	1	1	62	G1,2,3,4	0	3	0	A	X	
B2,G1=1	1	1	72	G1,2,3,4	0	5	0	A	X	
B3,G1=0	1	1	63	G1,2,3,4	0	3	0	A	X	
B3,G1=1	1	1	73	G1,2,3,4	0	5	0	A	X	
B4,G1=0	1	1	64	G1,2,3,4	0	3	0	A	X	
B4,G1=1	1	1	74	G1,2,3,4	0	5	0	A	X	
B5,G1=0	1	1	65	G1,2,3,4	0	3	0	A	X	
B5,G1=1	1	1	75	G1,2,3,4	0	5	0	A	X	
8E	1	1	76		0	1	0	0	I	
8F	1	1	77		0	1	0	I	R	
CD	0	0	78		0	1	0	0	I	
CE	0	0	79		0	1	0	I	R	

(continued)

3.2.8 (Cont.)

Output Trunk

This trunk is 64 bits wide. It transmits output data to the Map and Swap Units. The data formats for 32 and 64-bit mode are as shown below. Data will remain on this trunk for 10 nsec.



Output Control Trunk

The output control trunk transmits control or fault bits associated with results generated by the Scalar Floating-Point Unit. These signals come up with data and are held up for 10 nsec. The following signals are transmitted on the output control trunk:

Signal	Meaning of a "1" on Signal Line
Branch Cond. Met	The operands meet the compare condition. This line is zero when a compare is not being done.
Exit Cond. Met	The operands do not meet the compare condition. This line is zero when a compare is not being done.
Divide Timing Pulse	Divide operands will follow this timing pulse by 14 cycles.
Divide Busy	The divide element cannot accept new operands during the time this signal is a "1".

(continued)

3.2.8 (Cont.)

Signal	Meaning of a "1" on Signal Line
Data Flags 39, 41, 42, 43, 45, 46	See specification 10354636 for these definitions.

Instruction Conflicts

Due to the various instruction cycle times, conflicts may arise at the output of the Floating Point Interface and within the unit. Floating Point operations must not be initiated on cycles which will cause conflicts. The following procedure can be used to determine these conflict cycles:

C_A = the cycle at which operation A is initiated.

L_A = the number of cycles operation A spends in floating point.

C_B = the cycle time at which operation B is initiated.

L_B = the number of cycles operation B spends in floating point.

If operation B is initiated after operation A then $C_B \neq C_A + L_A - L_B$ to avoid a conflict.

In addition it must be remembered that no divide instruction may be initiated if the busy time has not expired from a previous divide.

(continued)

3.2.8 (Cont.)

Scalar Floating Point Maintenance Aid

This feature currently under study would allow the display of pertinent registers in floating point during multi-cycle instructions.

The Scalar Floating Point Unit is controlled by two separate microcodes. Each of these mirocodes provides control information to the integrated circuit logic to implement the instruction being performed. By altering this control information, i.e., reloading the microcode memories with specially modified microcodes, the contents of interval registers could be transmitted, unaltered, to the output of the unit. Maintenance software would use this information for display or fault isolation.

Disolay:

Only the registers critical to that instruction would be displayed grouped in timing ranks. It may be possible with a multi-oipe machine to display comparisons.

Additional information on this will be provided at a later date.

3.2.9 Absolute Bounds Address

The absolute bounds address mechanism provides the facility to notify the MCU of a memory reference (read or write) inside a specified block of memory. The block of memory is specified by an upper bounds sword address and a lower bounds sword address. Note that the addresses are absolute physical sword addresses transmitted from the MCU. The bounds addresses are defined as not included in the block of memory.

The classes of reference are:

1) Vector Read and/or Write Requests

2) CPU and/or Swap Requests

Bounds checking is effectively disabled, if either (or both) class 1 or 2 has neither possibility selected.

(continued)

3.2.9 (Cont.)

The Checker can selectively test various classes of requests for in-bounds conditions. Any combination of classes may be selected.

If the FMP has been stopped by a bounds hit, the hit must be cleared by the clear fault signal from the MCU before the FMP can be restarted. The FMP can be restarted to execute the next instruction in sequence.

The occurrence of a bounds hit (i.e., a selected memory reference inside bounds) is signaled to the MCU. To identify a second bounds hit, the MCU must clear the first bounds hit signal via the clear fault signal.

When a bounds hit is made, the sword address of the causing request is saved in the bounds hit register until a Master Clear or Fault Clear occurs.

The bounds limits and the bounds hit address refer to physical addresses, which are independent of all Memory Degradation modes. (The bounds test is applied to the address after any Degradation mode manipulation has been applied).

3.2.10 Trace Register

Register file address zero is used as the trace register. The trace register contains the address from which the most recent branch was taken. Register zero can be referenced by executing a 7D instruction. See the instruction specification for the mode of the 7D instruction which will move register zero to Main Memory. The maintenance station can read register zero by storing the Register File and reading absolute zero from memory. After a job to monitor exchange, the job's address zero in memory contains the address of the last branch taken prior to the exchange operation. After a monitor to job exchange, monitor's address zero (absolute zero) contains the address of the last branch taken prior to the exchange operation.

3.3 Vector Processor

The Vector Processor consists of three distinct subsystems, the Map Unit, the Buffer Unit and the Vector Floating-Point Ensemble (VFPE). The Map Unit is a single homogeneous logical element which controls all memory accesses by vector operations, and performs certain limited functions (such as Transmit Vector and Scatter/Gather) itself. The Floating-Point Ensemble is designated an ensemble because it consists of a set of nine identical arithmetic units, all of which operate in lock-step synchronization. One of the units is designated by control signals from the Maintenance Control Unit (MCU) as the auxiliary or spare unit. Normally, this unit will be performing the same functions as the remaining eight units, utilizing data inputs common to one of the operational units, but with its output data ignored. The self-checking circuits internal to that unit then can be exercised continuously even though the unit is off-line.

The Buffer Unit is physically part of the VFPE but is treated as a logical entity. It has nine identical sections, each of which is directly associated with one of the nine Vector Units in the VFPE.

The Vector Processor runs under its own local control. That is, the Instruction Issue Unit in the Scalar Processor passes sufficient information to the Vector Processor so that it can proceed independently. No active control is required from the Issue Unit. When the Vector Processor is given a process to perform, it checks for resources required and, if available, sets up and performs the required operations. If the resources are not available, the setup information is held in a one-word queue until the resources become available. When the Issue Unit finds the queue full, it suspends issuing to the Vector Processor until the queue is emptied.

(continued)

3.3 (Cont.)

The Vector Processor queue is, in fact, three queues - one each for the Vector, Map, and Buffer Units. The Buffer and Map Unit queues are further broken down according to the separate resources of each unit. Thus, if an individual resource is available, it immediately tries to perform the desired function.

For example, if the Map Unit is requested to get two vector streams from memory to be added in the VFPE and to store the results in the Buffer Unit, the Issue Unit sends information to set up the two read buses, R1 and R2, as part of the instruction issue. If R2 is in use at the time of the issue, the setup information for R2 is held in its queue and the R1 setup is performed (provided R1 was not in use). R1 then makes memory requests but, because the current operation which has R2 in use does not require R1 data, no data moves through R1.

As another example, consider a vector stream from memory, via R1 in the Map Unit, being added in the Vector Units to a data stream from the Buffer Unit with the result going back to memory. Concurrent with this the Issue Unit can send setup information to R2 and S2 in the Map Unit and WB1 in the Buffer Unit to cause a load of the buffer from memory. Because all of these resources - R2, S2, and WB1 - are not in use the setups are performed and the buffer starts loading. No conflicts occur because of the vector add being executed in parallel.

Holding its own setup information locally, a resource has two additional requirements in order to perform a function: valid data at its input and a place that will accept the processed output. This then is the control system for the Vector Processor - when valid data is presented at the input to a function resource, if the resource has been set up to perform an operation it sends an "accept" to the sending resource, and some number of cycles later produces valid output data. If the receiving resource is able to take the data it does so. If, however, proper setup of the receiving resource has not as yet been fully accomplished, acceptance of the data is not forced. If a given resource does not receive an "accept" to the resource supplying its input. Valid data is indicated by a "valid" signal on a single line (called the valid line) that accompanies the data. The "accept" signal is also a single line (called the accept line).

(continued)

3.3 (Cont.)

Thus, a complete operation consists of setting up a complete "valid" chain - from things that can source "valids" (Main Memory and Buffer Unit) to things that can sink "valids" (also Main Memory and Buffer Unit). If a complete "valid" chain is established for a given operation, that operation will proceed to completion. In most cases completion is determined by the write length of the output vector going to zero. When this occurs a signal is sent backward along the "valid" chain (in the "accept" direction) stopping the generation of "valid" and "accept" signals. However, operations being performed, memory addresses being referenced, and the "valid" connections are maintained. Thus, only the changes from one operation to another need be sent to start the next operation.

3.3.1 Vector Floating-Point Ensemble (VFPE)

Figure 3.3-1 provides a simplified block diagram of a single Vector Unit in the ensemble. Each unit is completely independent of another, with no interconnections between them for data or control. All incoming and outgoing control passes between each unit and the Map Unit or the Scalar Processor. Each Vector Unit contains two full multiplier and adder elements and two half-adder elements, each of which is capable of operating on pairs of 64-bit input operands or quartettes of 32-bit operands every clock cycle. Each arithmetic element (add, multiply) are segmented pipelines, three segments per element. Each segment requires one clock cycle of pipeline time. Thus two operands proceeding through all three segments for a combination add and multiply ((A+B)*C) would require nine minor cycles to pass from the select network to the result busses, Arithmetic Result 1 (AR_1) and Arithmetic Result 2 (AR2). A simple, normalized ADD operation utilizes the front-end add elements (FADD1 or FADD2), bypasses the multiply elements and completes the addition and post-normalization in the back-end add elements (BADD1 or BADD2). The total segments for a simple, normalized ADD is six or for a simple MULTIPLY operation it is also six segments of pipeline time. This pipeline length contributes to vector startup time as described in section 3.9.2

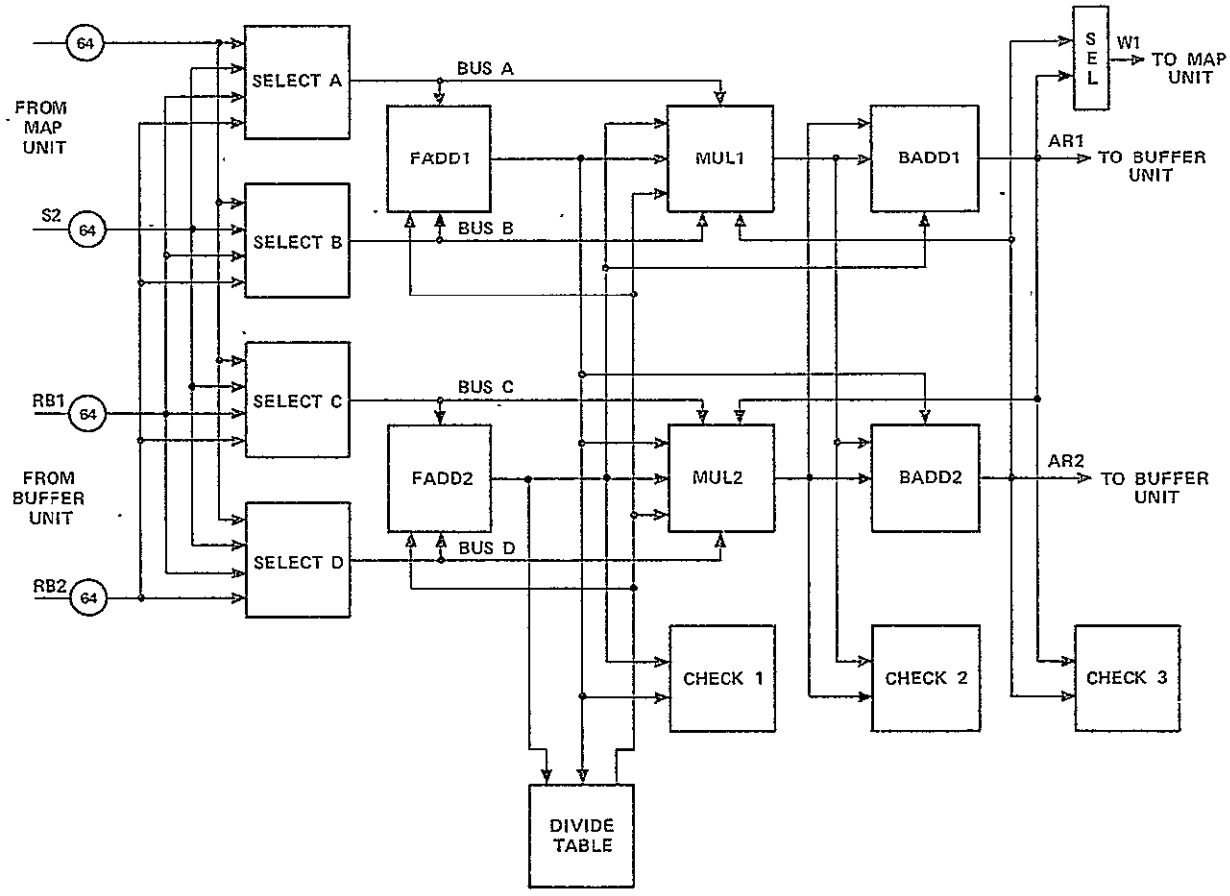


Figure 3.3-1 One Vector Unit

3.3.1.1 Read Bus Select Elements

There are four input data busses for each Vector Unit, RB1 (Read Bus 1 from the Map Unit), RB2 (Read Bus 2 from the Map Unit), S1 (Source 1 from the Buffer Unit), and S2 (Source 2 from the Buffer Unit). Each input bus is capable of supplying operands to any or all four of the functional streams (Bus A, Bus B, Bus C Bus D) which feed the various arithmetic elements. As can be seen from figure 3.3-1 then, any combination of input busses can be fed to any of the arithmetic elements, permitting such combinations to occur as (A*A)+(B*B) by supplying the A stream from the Buffer Unit (for example) via S1 and selecting it through SELECT A and SELECT B to the Bus A and Bus B sides of the multiply element. Likewise the B operands could be supplied from the Buffer Unit via S2 and selected through SELECT C and SELECT D to the C and D sides of the second multiply element (MUL2). The results of MUL1 and MUL2 would then be combined in the final back-end adder BADD 1, to form the sum of the two products.

The read bus select elements SELECT A, B, C and D are individually controlled by the C, D, E and F fields of the 9F (Vector Arithmetic) instruction which is interpreted by the Issue Unit and transmitted to the Vector Unit.

3.3.1.2 Write Bus Select Elements

On any given clock cycle a Vector Unit can transmit one 64-bit or two 32-bit result operands to the Map Unit for storage in memory via the WRITE 1 Bus. On any given clock cycle the BADD 1 and BADD 2 elements (back-end add elements) can produce one 64-bit or two 32-bit results, each of which are placed on their respective Arithmetic Result busses (AR1, and AR2). The results appearing on these two busses are defined by the suboperation codes for the 9F (Vector Arithmetic) instruction. The N field of the 9F instruction (section 3.2.1.160 of the Instruction Specification) directly controls which result bus, AR1, or AR2, or no bus, is connected to the Write Bus, W1.

3.3.1.3 Front-End Add Elements (FADD1 & FADD2)

Two identical arithmetic elements form the front-end functional processors of each Vector Unit. These elements are composed of a brenormalize network which aligns operands of unlike exponents, plus a full two's complement adder producing one 64-bit or two 32-bit results every minor cycle. There is no post-normalization shift network present in these elements. The output results from such an element is the equivalent of the FMP ADD or SUBTRACT UPPER or LOWER, with no normalize shifts being done on the result data.

The primary function of these adders in primitive operations (diadic arithmetic such as A*B) is to perform the pre-normalization of input operands (particularly for the divisor in divide operations) and to provide for complementing of one or more operands for functions such as (-A*B).

Each FADD element has its own independent microcode control so that diagnostics can be loaded via the microcode trunk to perform failure isolation to the lowest replaceable component level (LSI chip).

In addition to the pre-normalization of the divisor in divide operations, the FADD elements perform the necessary complementation of negative source operands prior to performing the table look-up that initiates the reciprocal approximations.

3.3.1.4 Multiply Elements (MUL1 & MUL2)

Each Vector Unit contains two identical multiply elements each with its own independent control logic. The multiply element inputs two 64-bit or four 32-bit operands and produces one 64-bit or two 32-bit results every clock cycle. This multiply operation is performed in three segments, each of which requires a minor cycle. In the first segment, four-bit groups of the multiplier are used to encode 8-bit groups of the multiplicand into a series of partial sums and carries For the remaining two segment times, these partial sums and carries are merged through a series of partial adders yielding a 96-bit wide, final product of

(continued)

3.3.1.4 (Cont.)

partial sums and carries which are finally added together in the back-end adder (BADD1 or BADD2). This addition operation produces a 96-bit wide coefficient result which can either be normalized, truncated, rounded, or left in upper or lower form (for double-precision arithemtic).

Inputs to the multiplier are controlled by the subfunction operations specified in the 9F (Vector Arithmetic) Instruction, and can come from the read bus select networks, the front-end adders, the divide table element (for divide operations), or from one of the arithmetic result busses emerging from the back-end adders depending on which operations, such as PRODUCT, are desired. If one of the two MUL elements is not specified in the suboperation, then identical inputs are selected for both elements and checking is enabled.

3.3.1.5 Back-end Adder Elements (BADD1 and BADD2)

Each Vector Unit contains two identical back-end adder units, each with its own independent control logic. The back-end adder consists of a rank of deskew logic for synchronizing the various partial sums and carries from the multiply elements , and a full three-input adder capable of combining the multiply output results with the output of either FADD1 or FADD2 of the other multiplier element. This function provides facilities such as (A*B)+C or (A*B)+(C*D).

Each back-end adder performs a 96-bit (in 64-bit operand mode) or two 48-bit (in 32-bit operand mode) coefficient addition every minor cycle. The first segment contains the latches and first addition of a pair of operands. The second segment contains the second addition of the resulting input pair of operands plus the final group carry/generate tree, and the final segment contains the rounding/truncation logic and post-normalization network. Post-normalization is controlled by the type of operation specified in the 9F instruction.

Inputs to back-end adders are controlled by the subfunction code in the 9F (Vector Arithmetic)

(continued)

3.3.1.5 (Cont.)

instruction. The outputs are placed on the arithemtic result busses AR1 and AR2 by the adders BADD1 and BADD2 respectively. In addition, each of the result busses is connected directly to the input selection network of the Buffer Unit.

3.3.1.6 Divide Table Element

The divide operation utilizes most of the arithmetic elements in the Vector Unit To achieve a divide rate of one result per minor cycle (for 24-bit coefficient accuracy), the reciprocal divide approximation is utilized. In this mode, the divisor is pre-normalized and its absolute value yielded by a front-end adder. This resulting divisor is then sampled by taking 12 bits of the coefficient from the left-most (or most significant) end, not including the sign (which will always be zero since the absolute value of the divisor is used), and not including the most significant bit (which will always be one since a normalized divisor is used), yielding bits 18-29 of the 64-bit coefficient. These twelve bits are used to address a read-only memory (ROM), or look-up table, called the divide table element. A 39-bit word (plus one parity bit) is read from the ROM at that address. The word is partitioned into two fields, S (14 bits) and T (25 bits). The field is used as input to the other front-end adder (for complementation if the divisor was originally negative), and the S field is used as inout to a multiplier to form the product of S times the remaining bits of the coefficient (the 34 bits not used in the table look-up). The multiplied result is subtracted from T in the back-end adder and that result is then fed into the other multiplier along with the dividend to form a 64-bit result of which 24 bits of the coefficient are accurate. The pair of results out of the back-end adders can be stored in the buffers or memory, thence retrieved during a second pass (DIVIDE 2) to perform the necessary corrections to produce a full 64-bit result.

(continued)

3.3.1.6 (Cont.)

Figure 3.3-2 shows the interconnection for the first pass divide operation (DIVIDE 1) which yields a correct 24-bit coefficient result. Not shown is a network which transmits the lower 34 bits of the divisor with the upper 14 bits cleared to zero. It is this quantity which is multiplied times the slope (S) value to form the first product in the reciprocal approximation.

Figure 3.3-3 shows the interconnection scheme for the second pass divide operation (DIVIDE 2) which is used to produce 64-bit floating-point quotient results. The input operands required for this second pass are: the first pass quotient (which is by itself adequate for 32-bit arithmetic), the original divisor, and the intermediate product which is normally stored in the Buffer Unit.

The divide table element is referenced once each minor cycle during the DIVIDE 1 operation. This means that when in 32-bit mode, the divide rate is the same as for 64-bit mode during the first pass, one result per minor cycle. Usually however, the need for 48-bit accuracy in the coefficient portion of the 64-bit result will require the DIVIDE 2 pass which then creates a true 64-bit divide rate of one result every two minor cycles per Vector Unit.

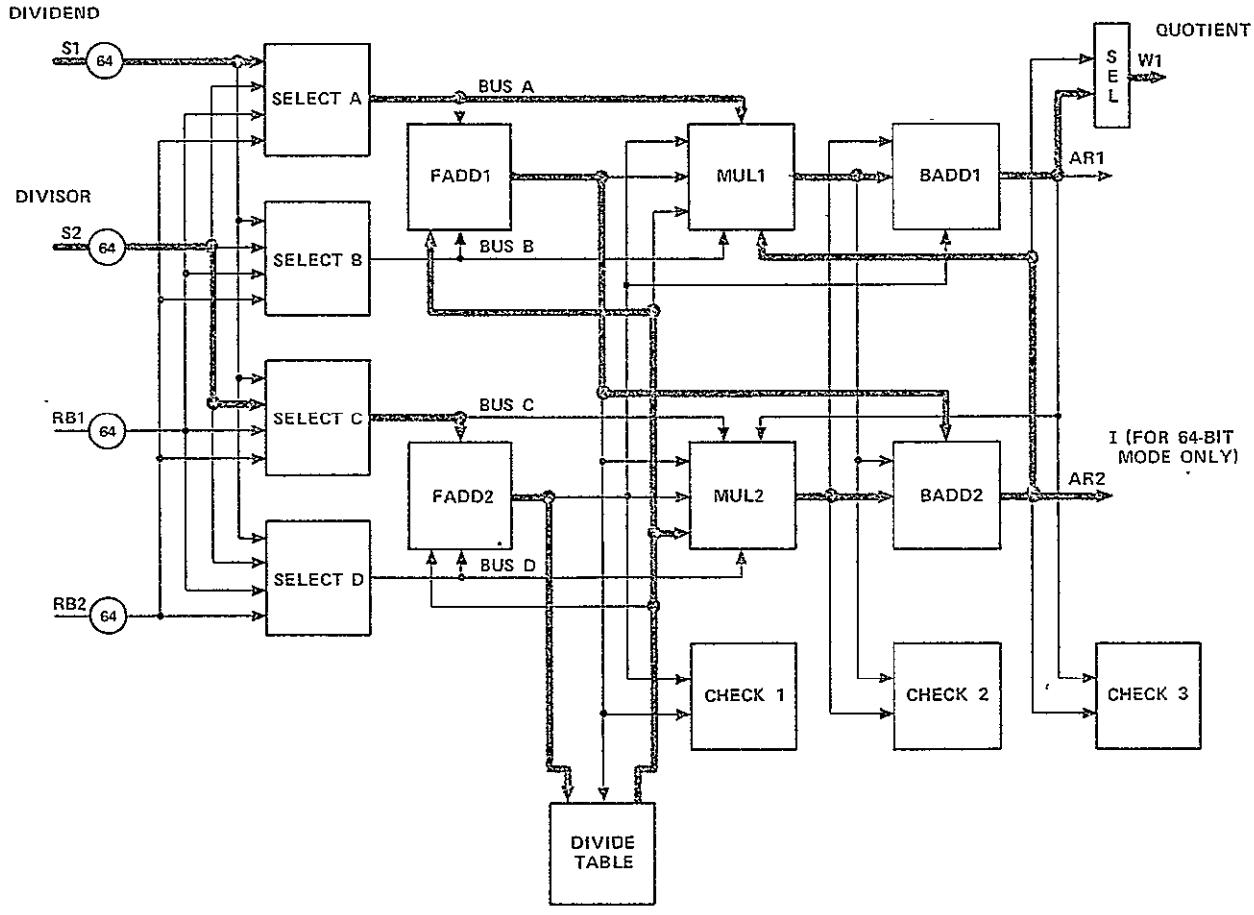


Figure 3.3-2 First Pass for 32-Bit Divide

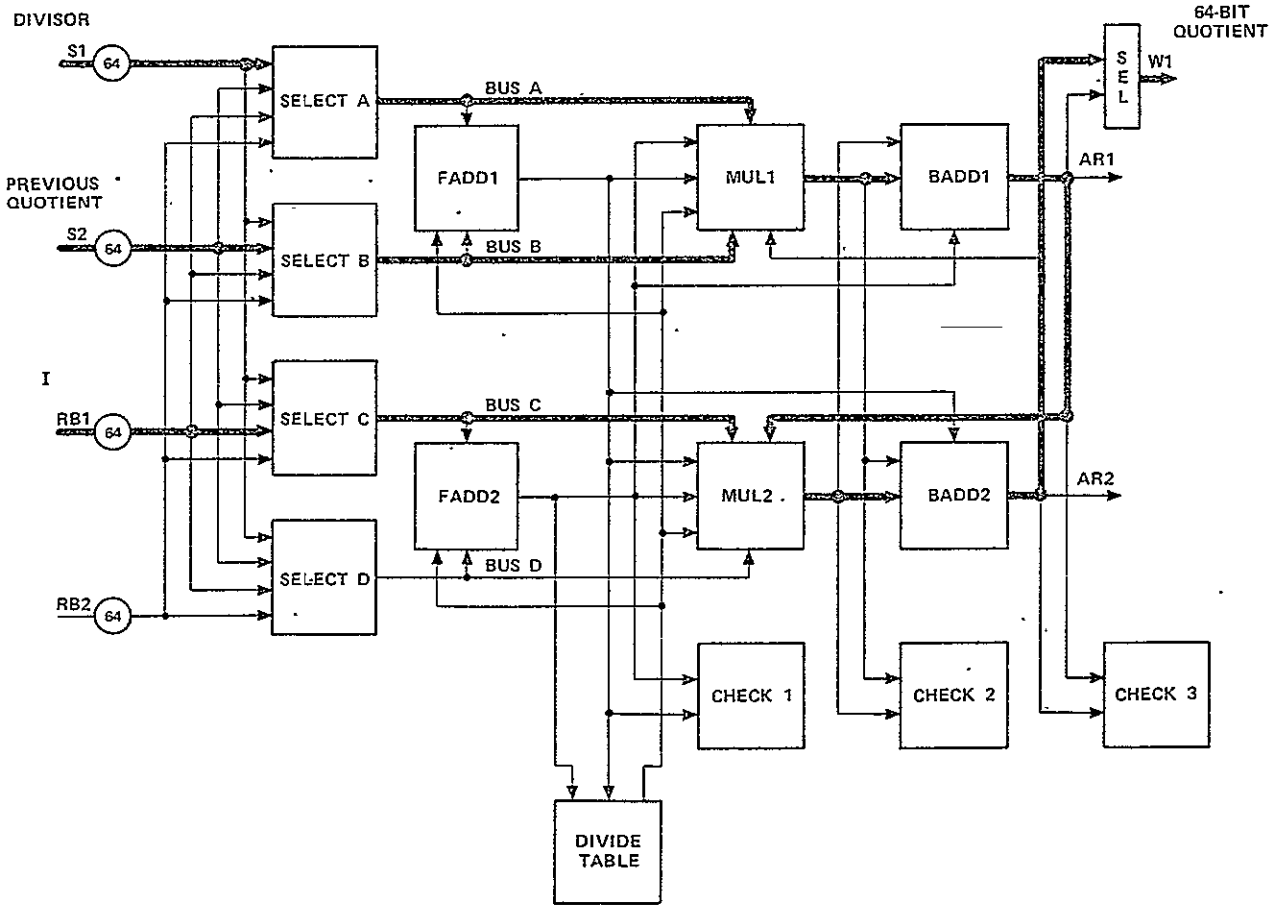


Figure 3.3-3 Second Pass for 64-Bit Divide

3.3.1.7 Error Checking

3.3.1.7.1 SECDED

Each of the trunks entering and leaving a Vector Unit and connecting to the Map Unit contain SECDED (single error correction, double error detection networks). Read Bus 1 (RB_1), Read Bus 2 (RB2) inputs contain SECDED detection and correction circuits, while the Write Bus 1 trunk contains a SECDED code generation network.

SECDED is carried on a 32-bit basis, seven bits for each 32 bits of data. Thus all input and output trunks possess 78 actual bits of transmitted data.

3.3.1.7.2 Parity

The divide table element consists of a loadable RAM that behaves as a read only memory during normal Vector Unit operation. Each 39 bits of divide table data have a single parity bit associated with them. Upon each table read, the parity is checked. If a error occurs, the Vector Unit is immediately halted and the Maintenance Control Unit (MCU) is alerted by an error flag. In addition, the Scalar, Map, and Swap Units are sent stop signals.

Upon command of the MCU the Vector Unit can transmit the failing memory location in the divide table, the operand location in the input vectors for the failing case, and the P counters of all the control microcodes for the Vector Unit, to assist in maintenance actions.

Each of the microcode memories contains a parity bit for each word addressed. In the event that a parity error occurs, the microcode sequence is frozen and the P counter transmitted to the MCU on command. A flag indicating which microcode is failing is sent to the MCU. The Map, Scalar, and Swap Units are also sent stop signals.

3.3.1.7.3 Result Checking

Each Vector Unit is supplied with three coincidence checking networks, capable of comparing the results produced by the identical pairs of arithmetic elements. CHECK 1 compares the outputs of FADD1 and FADD2, CHECK 2 compares the outouts of MUL1 and MUL2, and CHECK 3 compares the results of the final adders BADD1 and BADD2. Checking is enabled under the following circumstances.

1. When the same input trunks are selected into the pair of operand ports A&C and B&D, and the identical functions are selected for the pair of elements FADD1, FADD2 or MUL1, MUL2 or BADD1, BADD2;

2. When a given element is idled during an operation. For example, the suboperation code 02 would invoke the operation A+B and C+D thus idling the multiply elements; In this case a pair of operands emerging from the front-end adders would be enabled into both MUL1 and MUL2 automatically by the Vector Unit. The multiplied output, athough meaningless to the programmer, would be checked by the checking network.

3. When one of a pair of elements is idled by a particular suboperation code. For example the suboperation code 05 would cause the operation (A+B)*D, thus MUL2 would be idle. In this case the Vector Unit would automatically enable the same pair of inputs to both multiply elements. The checker would then be enabled.

It can be seen that the programmer can explicitly control checking in some cases by setting the appropriate fields in a 9F add instruction to select identical operands to identical elements.

In the event that an enabled checker discovers a mismatch in the output data, the Vector Unit is halted, a stop signal is sent to the Map, Swap,and Scalar Units, and the MCU is alerted.

3.3.1.8 32/64-Bit Arithmetic

Each Vector Unit is capable of processing two 32-bit or one 64-bit results each minor cycle in each of its arithmetic element segments, except for the divide table which produces one 32-bit result per cycle.

Each arithmetic element except for the divide table can also process a combination of one 64-bit and one 32-bit operand each minor cycle as input to an operation. For example, the FADD1 element could be accepting a 64-bit input operand on its A trunk and A 32-bit operand on its B trunk. In this mixed mode FADD1 would produce a 64-bit result.

Each of the input trunks, from either the Map Unit or Buffer Unit, provide a flag indicating what mode that particular trunk is operating in, either 64 or 32-bit. The Vector Unit then automatically configures its arithmetic elements to accept that form of data on that trunk.

The output trunks to the Map Unit and Buffer Unit also provide a flag to the Vector Unit indicating what mode they expect their operands to be in. Thus the Vector Unit is responsible for the necessary truncation or expansion of data to match that format required by the receiving units.

(continued)

3.3.1.8 (Cont.)

Floating-point numbers in the CDC FMP are two lengths, 32 bits and 64 bits. The 32-bit format has an 8-bit exponent and a 24-bit coefficient (Figure 3.3-4). The 64-bit format has a 16-bit exponent and a 48-bit coefficient. The left-most bit of each exponent and coefficient is the sign bit. A detailed description of floating arithmetic is presented in the instruction specification.

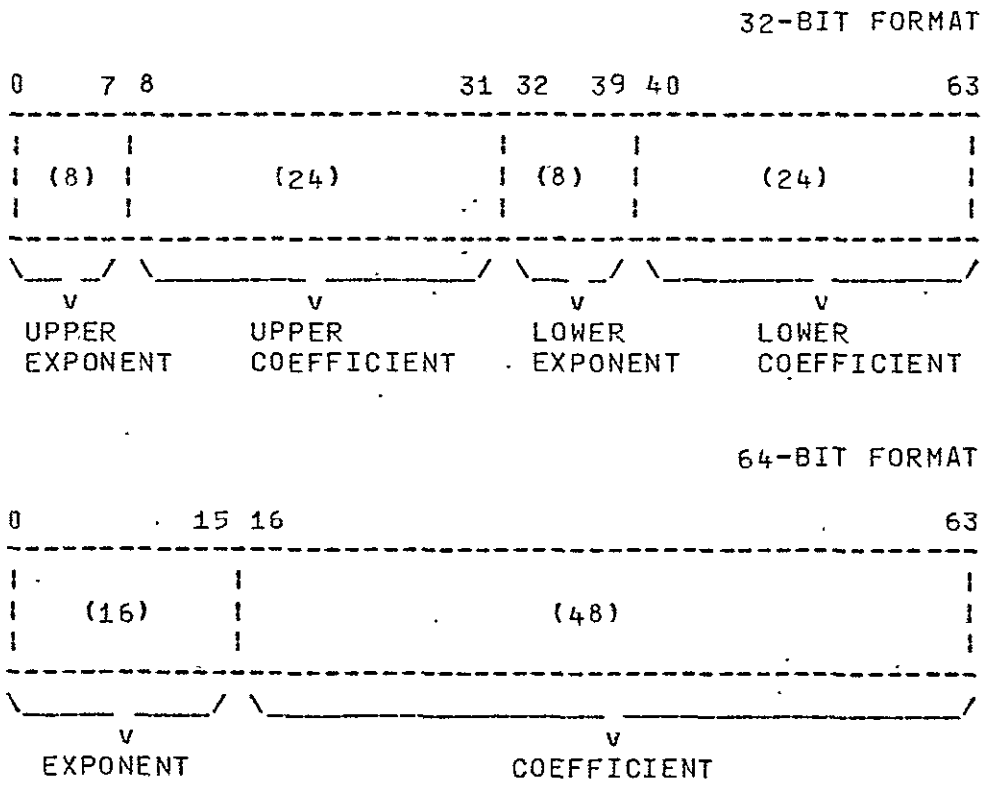


Figure 3.3-4 Operand Formats

3.3.1.9 Asynchronous Control

As all other units, the Vector Unit controls the movement of operands through its various elements by request/accept signals. Therefore, as soon as data ready signals appear at the ports selected by a particular 9F operations, the Vector Unit will begin to move tha data through its networks. The results will be placed on the selected output busses, and no more data will be placed there until an accept is received from the selected trunk destination. That is the purpose of the fields in the 9F which designate which output ports to expect accepts on during an arithmetic operation.

Likewise, the Vector Unit returns an accept for every operand it takes from an input port, thus allowing the unit supplying operands to move a fresh operand into place on the trunk. In the case of mixed mode operations where the rate of supply can exceed the rate that the Vector Unit can process, the accept flag consists of two bits indicating whether the lower or upper 32-bit operand has been accepted on the particular 64-bit trunk.

3.3.1.10 Control Signals

(To be defined later)

3.3.1.11 Microcode Terms

(To be defined later)

3.3.1.12 Interface Timing

(To be defined later)

3.3.1.13 Exchange Operation and Interruots

The purpose of the exchange is to change the prime role of the CPU. In job mode, job tasks are performed; in monitor mode, the system decisions are made.

Some instructions in progress may be interrupted prior to their completion. The invisible flags stored in the invisible package are used to restart the interrupted instruction exactly where its output left off.

Job mode data processing can be monitored, during monitor mode, by examining the Stall Bit in Word 8 of the job's invisible package. The Stall Bit is a "1" if no data was processed during the job time-slice that resulted in the preparation of the invisible package.

Invisible Package

The invisible package is always stored starting at an even numbered sword address. Therefore, the right-most 10 bits of the starting address of the invisible package must be zeros. This is as indicated in the Exit Force instruction write up in the Instruction Specification.

The monitor must set up an invisible package for each job. There is NO invisible package for the monitor program itself.

If a job is to be re-entered, the monitor should not alter the job's invisible package.

Figure 3.3-5 shows the format of the invisible package.

(continued)

3.3.1.13 (Cont.)

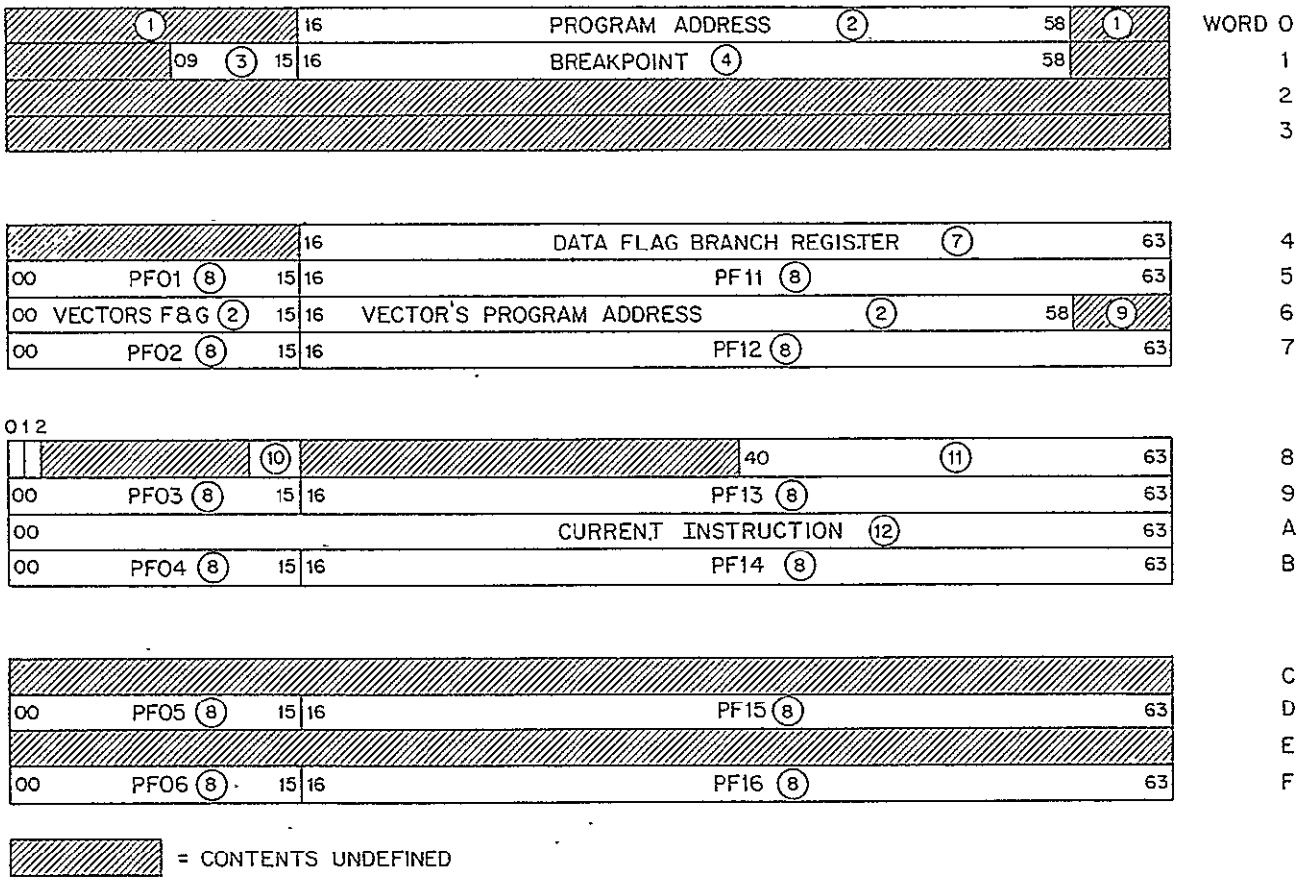


Figure 3.3-5 Invisible Package Format

(continued)

3.3.1.13 (Cont.)

The following notes apply to Figure 3.3-5.

1 Bits 0-15 and 59-63 are not used and must be set to zeros.
2 Quantity is loaded or read/stored or written by the Scalar Processor only.
3 Usage bits for breakpoint register.
4 Quantity is loaded/stored by Vector Processor only.
5 N/A
6 Bit 16 Flag 0
Bit 17 Flag 1
Bit 18 Flag 2
Bit 19 Flag 3
Bit 20 Interrupt Flag
Bit 21 NOT USED
Bit 22 Load/Store1
Bit 23 Load/Store2
Bit 24 Subfunction bit 0
Bit 25 Subfunction bit 1
Bit 26 Subfunction bit 2
Bit 27 Subfunction bit 3
7 Quantity is loaded/stored by the Vector Processor only.
8 Words 5,7,9,B,D and F are loaded by both the Scalar and Vector Processor. These words are stored by the Vector Processor if the vector restart bit (word 8 bit 0)=1 and by the Scalar Processor if the bit = 0.
9 Bits 59-63 are not used and must be set to zeros.

(continued)

3.3.1.13 (Cont.)

10 Bit 0 vector restart bit.
The Vector Processor's instruction register receives bits 0-15, word 6 and bits 16-63 word A. A vector will restart without reloading the vector instruction from memory only if bits 16-63, word A are not needed to restart (Bit 0, Word 8=1).

Bit 1 Register file's scalar enable (Bits 0 and 1 are loaded by the Scalar Processor and stored by the Vector Processor).
Bit 2-11 are not used.
Bit 12 Stall bit. This bit is a "1" if no data is processed.
Bit 13 Fault test instruction enable. For further information see specification 11845800.
Bit 14 Monitoring counters enable. For further information see Section 3.7 in this specification.
Bit 15 ASCII =0, EBCDIC =1 (Bits 12-15 are loaded/stored by the Vector Processor only).

11 Job Interval Timer. Quantity is loaded/stored by the Vector Processor only.
12 Quantity is stored by the Scalar Processor and loaded by neither.

(continued)

3.3.1.13 (Cont.)

Exchange from the Monitor to a Job

This is always accomplished with an Exit Force instruction. The monitor program must set up the invisible package for the job prior to exchanging to that job via the Exit Force instruction. The Exit Force operation is as follows:

1. The CPU's invisible registers and flags are loaded from the invisible package located starting at the bit address in the monitor's register T specified by the Exit Force instruction. This starting address is saved in a register to provide for storing the current invisible package when returning to the monitor program.

2. The Register File for monitor is stored into absolute memory locations 0 through $3FC0_{16}$. The Register File for the job is loaded from the job's memory locations $100000-103FC0_{16}$. Any job mode references to this area of a job's memory causes the executing instruction to be treated as an illegal instruction.

3. The CPU mode is changed from monitor mode to job mode.

4. The contents of P (program address register) are then read up and an appropriate start sequence is executed.

(continued)

3.3.1.13 (Cont.)

Exchange from Job to the Monitor

The Exit Force instruction and the channel interrupt are the two normal ways of getting from a job in the job mode to the monitor program in Monitor Mode. Attempting to execute a monitor-type instruction in job mode or an attempt to execute an undefined op-code comprise the third way into the monitor. Except for the starting point in the monitor program, the operations performed in getting to the monitor are identical for the three.

The operation is as follows:

1. The current invisible registers and flags are stored into the invisible package starting at the same address used to load the invisible package when the job was entered.

2. The Register File for the job is stored in memory locations 100000-103FC0 and memory locations 0 through $3FC0_{16}$ are read and put into the Register File.

3. The CPU mode is changed from job mode to monitor mode. Any external interrupts which occur after this point are honored only if the CPU executes an Idle instruction. If the CPU does not execute an Idle instruction, the interrupts are saved until the CPU mode reverts to job mode, or until the monitor program clears those interrupts with a 0E (Translate External Interrupt) instruction.

4. The monitor program is executed starting at the absolute address contained in the right-most 48 bits of the monitor's register 3,5,6 or 7.

   Refer to Table 3.3-1 for methods of getting from job to monitor mode.

(continued)

3.3.1.13 (Cont.)

If an attempt is made by the monitor program to perform an undefined op-code, an automatic branch is made to the absolute address contained in the monitor's register 4. This hardware trap is to aid in the debugging of the monitor software and to trap some hardware failures. This trap is not to be utilized by the monitor software as a "normal" branch.

TABLE 3.3-1. JOB TO MONITOR METHODS

Method of Getting to the Monitor	Monitor Register, the Contents of which is Used to Set P
1. Undefined instruction, Monitor type instruction in Job Mode, or a reference to the Register File as memory (bit address 0000-3FFF $_{16}$.)	Register 3
2. Undefined OP Code in Monitor or reference to the Register File as memory (bit address 0000-3FFF $_{16}$).	Register 4
3. Exit Force.	Register 5
4. Channel Interrupt.	Register 6

(continued)

3.3.1.13 (Cont.)

The bits in the external interrupt register are assigned as shown in the following table:

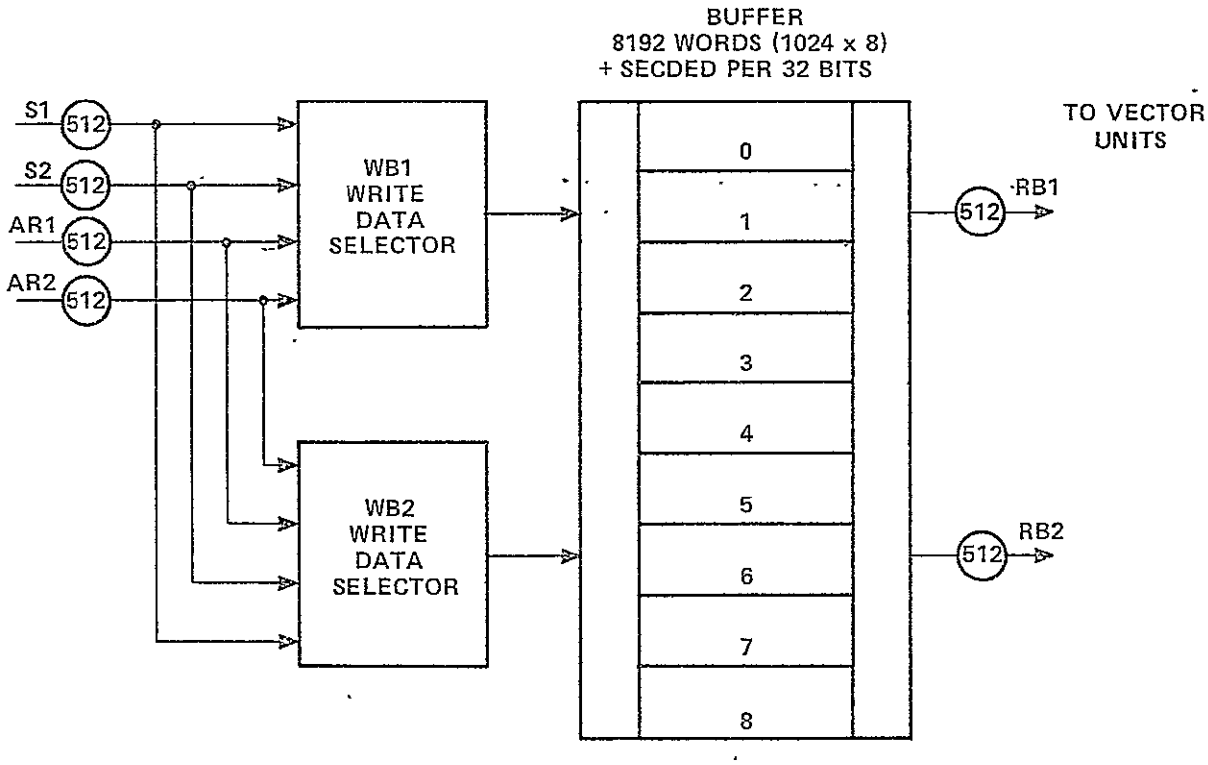
TABLE 3.3-2. EXTERNAL INTERRUPT REGISTER BIT ASSIGNMENTS

External Interrupt Line	Assignment	
0	I/O channel	0
1		1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10		10
11		11
12		12
13		13
14		14
15	I/O channel	15
16	Monitor Interval Timer	

3.3.2 Buffer Unit

Each Vector Unit contains a buffer capable of holding from 1024 to 8192 64-bit operands, depending upon machine configurations. The total configuration of eight buffers (plus one spare) constitutes the Buffer Unit. See Figure 3.3-6. The basic configuration of 1024 operands provides a capacity of 8192 total operands that can be held within the Vector Processor. In the maximum configuration this can be as high as 65,536 operands.

Although the buffers are physically contained within each Vector Unit to limit access times, they are treated as logically separate entities on a par with the Map and Swap Units. Thus there is a separate control for addresses for reading and writing and the format of the operands (32 or 64-bit).



NOTES:

1. S1, S2 FROM MAP UNIT
2. AR1, AR2 FROM VECTOR UNIT
3. ALL DATA PATHS CARRY SECDED BUT ONLY THE DATA BITS CARRIED IN A PATH ARE NUMBERED

Figure 3.3-6 Buffer Unit

3.3.2.1 Read and Write

Each buffer is capable of reading two independent operands every minor cycle, as well as writing two independent result operands per minor cycle. The selection of which operands to write into the buffer is performed by the write data selectors 1 & 2 (see figure 3.3-6). Note from the figure that any bus can feed any of the two select networks at the same time. Thus it is possible to input a data stream from S1 (the Map Unit) to both Write Buffer 1 (WB1), and Write Buffer 2 (WB2) simultaneously.

The write addresses or WB1 and WB2 are independent and are set up by separate Buffer Unit suboperations. Thus in the illustration, S1 could be written into two separate, independent areas of the Buffer Unit, simultaneously.

Read operations can proceed from independent addresses in the same minor cycle since the buffer is composed of high-speed ECL RAMS allowing random addressing at the rate of two per minor cycle. As operands are read from the buffer and placed on the designated trunk: a "data valid" signal is placed on the corresponding trunk control lines. In addition, the format of the data (32 or 64-bit) is also flagged on the respective trunks to the Vector Unit.

The four ports providing input operands are connected to the Map Unit (S1 and S2) and the output result bus of the Vector Unit (AR1 and AR2).

3.3.2.2 Control

The Buffer Unit processes its own control logic, despite the fact that it is intimately connected within the host Vector Unit. The control scheme is based on a loadable microcode which handles the interpretation of the particular suboperation code, the setting up of addresses, and the control of incrementing of address counters and testing of termination thresholds for all vectors whose source or destination is within the Buffer Unit.

3.3.2.3 Error Checking

All data paths within the Buffer Unit carry single error correction, double error detection codes with each 32-bit operand. In the event that a read operation causes the discovery of a single-bit error, the data will be corrected and the unit will not halt. However, the error address at which the data was read will be "latched up" for sampling by the MCU, along with the SECDED syndrome bits which will be sent an error flag.

In the event of a double-bit error, the Vector Unit will be halted, a stop signal sent to he Swap, Map, and Scalar Units and an error flag transmitted to the MCU.

3.3.2.4 Control Signals

(To be defined later)

3.3.2.5 Microcode Terms

(To be defined later)

3.3.2.6 Interface Timing

(To be defined later)

3.3.3 Map Unit

Figure 3.3-7 gives a general block diagram of the Map Unit. This unit is divided into 12 functional elements, each of which contains its own control microcode, and thus is able to operate somewhat independently of the other elements. This feature is primarily intended to facilitate fault isolation and maintenance. The Map Unit controls all accesses to Main Memory for reading and writing by the Vector Unit or the Map Unit itself. The Map Unit also contains certain vector functions which it can perform itself (SCATTER, GATHER, COMPRESS, MASK, MERGE).

RADL

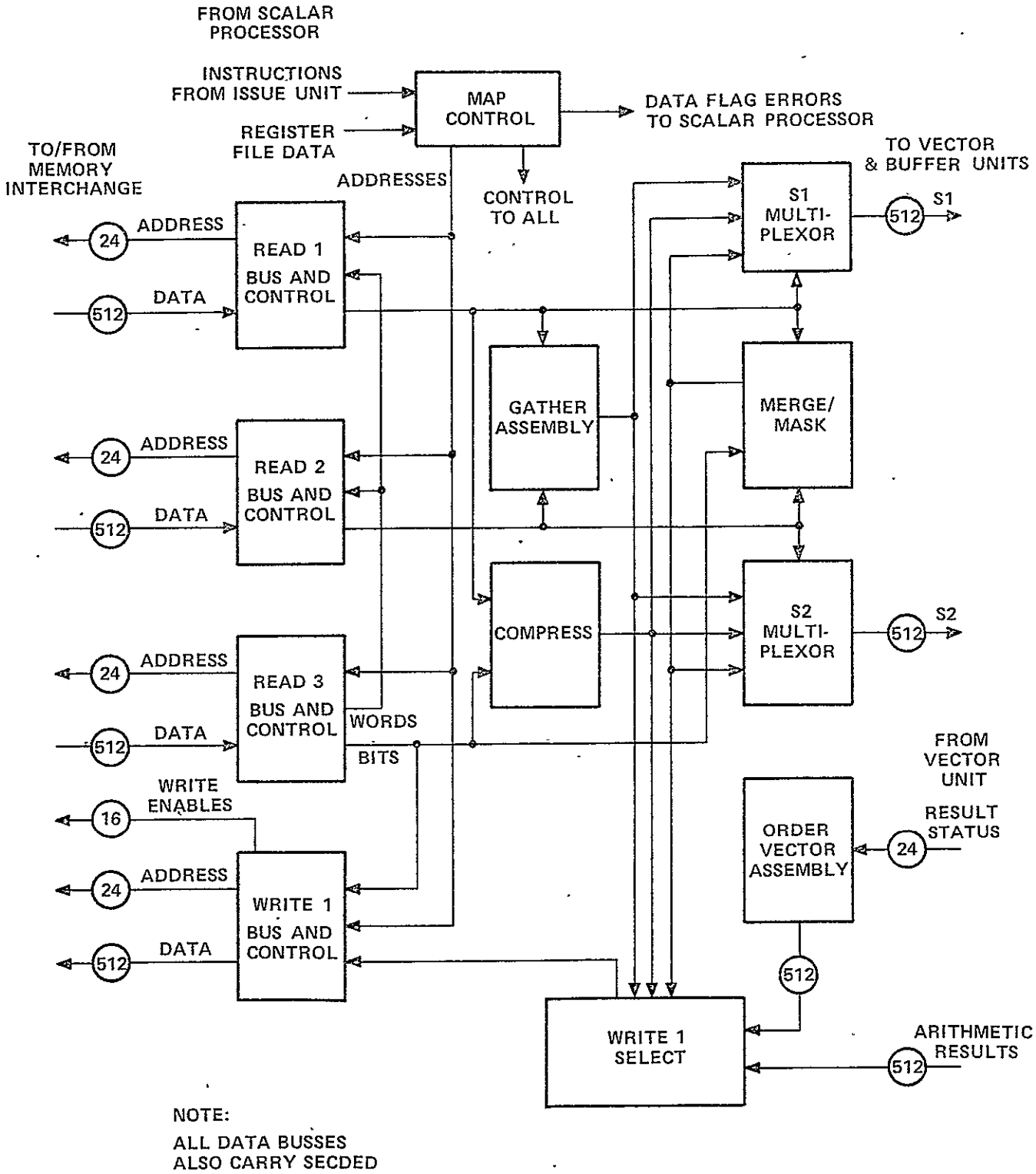


Figure 3.3-7 Map Unit

3.3.3.1 READ 1 and READ 2

The Map Unit contains two identical bus read elements termed READ 1 bus and control and READ 2 bus and control networks. Each element provides the addressing, address incrementing and data bus buffering required for the interconnection to Main Memory busses.

3.3.3.1.1 Error Checking

All data busses within the Map Unit provide 7 bits of SECDED code for every 32 bits of data. To assist in fault isolation, each read bus element contains a SECDED error checking network, for its respective input port. In the event that a single-bit error is discovered, the contents of the respective address counter, an error flag, and the syndrome bits are held in MCU interface registers for sampling by the MCU.

In the event that a double-bit error occurs, all of the above actions take place, but in addition, the entire Map Unit is halted, a stop signal is sent to the Vector, Swap, and Scalar Units, and a fatal error signal is transmitted to the MCU.

3.3.3.1.2 Data Movement

Each data bus can move 512 bits (plus SECDED) of data every minor cycle. All requests to memory yield a full 512-bit data word, while the outputs of the read bus elements can emit 512, 128, 64 or 32-bit data items.

3.3.3.1.3 Address Control

Each bus element manages its own addressing control of memory access. Initial addresses are sent to the bus elements by the map control element. In addition, address increment values (+1, -1 or some prescribed variable N) are sent to each buffer element by the map control. An additional set of control lines from the map control indicate what addressing modes, and what data widths are required of the bus control element. A special address increment port is supplied by the READ 3 bus increment, GATHER operation.

(continued)

3.3.3.1.3 (Cont.)

The modes of operation for the address logic are:

1. Full streaming--In this mode data is moved at the maximum rate supplied by the memory system. Normally 512 bits of data are transmitted to the input port each minor cycle; 512 bits are passed directly to the S1 and S2 output ports which supply the Vector and Buffer Units with data. This mode is used for all vector arithmetic operations when all operands are the same size. For example, if the operation performed is a memory-to-memory vector addition with input and result operands all 64 bits wide, the Map Unit will provide data at streaming rates. However, if one of the operand streams is not the same size (say one 32-bit input and one 64-bit input operand), then the bus supplying the 32-bit operands will move half as much data per minor cycle in order to synchronize with the 64-bit data movement.

2. Half streaming--In this mode memory requests are not made each minor cycle, but are made every other minor cycle. This case arises for the mixed 32/64-bit mode previously discussed.

3. Word or half-word streaming--Depending on the operand size, the input read streams can be moved at regular intervals in word or half-word increments. This mode is used for the COMPRESS and MASK operations which guarantee that new elements will be used evey minor cycle.

4. Burp mode--This mode moves word or half-word elements, as needed, by the functional element in the Map Unit. Its peak rate is one data element every minor cycle, while it is possible for many minor cycles to pass before the data element is required. The major use for this mode is in the vector MERGE operation where data elements are moved from the read stream depending on the presence of one-bits in the corresponding order vector.

(continued)

3.3.3.1.3 (Cont.)

5. Read reverse mode--The memory system and addressing and data disassembly networks are capable of streaming data in any of the previous modes in reverse order if the address increment is -1.

6. Random access mode--All previous modes deal with sequentially accessed data, moving such data at rates prescribed by the operation in process. In the case of the vector GATHER operation, data is accessed non-sequentially from the Main Memory. Two submodes are provided in this case--fixed increment and variable index.

   In the fixed increment mode, the map control element provides a positive or negative integer value which is used as an increment for each new memory address. Thus, instead of the normal increment of +1, any integeral value can be used in this mode. In such cases for example, the memory addresses produced are M, M+N, M+2N and so forth, where M is the initial address and N is the fixed increment.

   In the variable index mode, the value of the memory address is computed from the initial address M and the contents of a list of integers I. For each integer (positive or negative) in the list, a memory address is computed and a corresponding request sent to memory at the rate of one per minor cycle, per bus control element. The memory request will yield either one 64-bit or one 32-bit operand depending on the format desired by the operation.

The two bus control elements (READ 1 and READ 2) are capable of transmitting simultaneous memory requests, if the addresses are odd and even, respectively. Thus a peak rate of two random access requests is possible with the two elements operating in tandem.

3.3.3.2 READ 3 Bus and Control

A third input bus is provided in the Map Unit for handling special streams of data used in the Map Unit.

3.3.3.2.1 Error Checking

SECDED error codes are carried on the READ 3 data bus, but are not passed on to other Map Unit elements as are the READ 1 and READ 2 data bus error codes. This is due to the fact that READ 3 data is disassembled into bit streams or index streams and the 32-bit SECDED parcel is no longer intact. Thus the READ 3 bus element provides error checking of the inout data only.

Error handling and correction and reporting to the MCU are identical to that supplied by the READ 1 and READ 2 bus elements. See paragraph 3.3.3.1.1.

3.3.3.2.2 Data Movement

The movement of data within and out of the READ 3 bus element is more complex than the READ 1 and READ 2 counterparts. This is due to the nature of operands provided by the READ 3 element.

1. Control vector--The writing of data into Main Memory on the WRITE 1 data bus can be controlled down to the 32-bit level. That is, in any given minor cycle, 512 bits of data can be transmitted to the Memory Interchange, but any combination of 32-bit operands within that 512 bits can be suppressed (not written into memory). This action is controlled by the group of bits called control vector bits. When operating a vector to memory operation at full streaming rate, 16 32-bit quantities are transmitted to memory. If the control vector operation is invoked (see instruction specification for 9D instruction), a group of 16 bits is provided by the READ 3 bus control element. Thus READ 3 must, in full streaming mode, be capable of disassembling the 512-bit input data sword (super-word) into groups of 16-bit parcels, one parcel per minor cycle.

(continued)

3.3.3.2.2 (Cont.)

2. Order vector--The vector operations MASK, MERGE, and COMPRESS derive their control of data movement from a group of bits called order vector bits (taken from Iverson's APL). The presence of a bit in the order vector may mean the transmission of that corresponding data element in the READ 2 stream (in the case of vector MASK operations). In this mode, bits are transmitted in groups of sixteen to the appropriate functional element (MASK, MERGE or COMPRESS networks) at the rate required by that element. In normal operation four bits are moved (as are four operands) every minor cycle, thus requiring the READ 3 bus to provide 16 bits every four minor cycles.

3. Indexed list--When performing the vector operations GATHER or SCATTER using a list of indexes, the READ 3 bus supplies these indexes at the rate required by the particular bus element. Indexes always fill a 64-bit operand, although the maximum index requires only 16 bits. Thus indexes are moved at the rate of two words (64 bits wide) every clock cycle requested by the READ 1 and READ 2 controls.

   Since memory requests in this mode are essentially random, the rate of data movement is not predictable, due to memory conflicts between READ 1 or READ 2 or memory busy's due to previous requests. Thus the memory address and request control operates in a form of burp mode.

3.3.3.2.3 Address Control

The management of addressing and memory requests is much simpler in the READ 3 control element, since all data is sequentially accessed. Note that the maximum rate at which READ 3 is required to deliver operands (in the indexed list mode) is two 64-bit words each minor cycle. This means that when the Vector Unit is running in full streaming, READ 3 is making memory requests every 16 minor cycles, and when not in full streaming (SCATTER/GATHER), READ 3 is requesting memory evey 4 minor cycles. This lower rate for READ 3 makes it possible to share the residual memory bandwidth with the Swap Unit and instruction stream requests issued by the Swap and Scalar Units.

3.3.3.3 WRITE 1 Bus and Control

The WRITE 1 Bus (W1) provides the output port for the Vector and Map Units back to Main Memory.

3.3.3.3.1 Error Checking

The WRITE 1 bus provides 7 bits of SECDED code for every 32 bits of data transmitted. SECDED codes are generated by each of the functional components of the Vector and Map Units, so that WRITE 1 control only checks for errors in the operands being transmitted through it. This feature is provided for fault isolation and maintenance procedures. Error checking, correction and MCU communications are the same as for READ 1, READ 2 and READ 3.

3.3.3.3.2 Data Movement

The WRITE 1 bus is capable of transmitting 512 bits plus SECDED each minor cycle when in streaming mode.

In addition to the 512 bits, a group of sixteen bits called write enables are transmitted to enable the storage, or suppression, of any 32-bit quantity transmited to the memory system. These write enables permit the storage of vectors beginning at memory address other than 512-bit boundaries, and the ending of vectors on other than 512-bit boundaries.

The write enables are also used to transmit control vector bits for selective storage suporession when invoked by that particular suboperation for the Map Unit.

3.3.3.3.3 Address Control

WRITE 1 is capable of writing sequential data at streaming rates, or writing data at fixed increments (as READ 1 and READ 2 can read data at fixed increments), or writing data based on a list of indexes provided by the READ 3 trunk (SCATTER operation).

3.3.3.4 GATHER Assembly Network

When performing the GATHER operation, the individual operands (either two 64 or two 32-bit operands) which are delivered each minor cycle by the READ 1 and READ 2 busses must be assembled into a contiguous vector. This network provides that function. In addition to handling the maximum rate of operand input, the network must also be able to handle burp rates as memory and bus conflicts interrupt the smooth flow of data.

All data gathered includes its corresponding SECDED codes (which is based on 32-bit parcels), but no error checking or correcting logic is included.

3.3.3.5 S1 and S2 Multiplexors

The Map Unit provides two data ports supplying the Vector and Buffer Units. These are the S1 (Source 1) and S2 (Source 2) ports appearing in Figure 3.3-7. These multiplexors are provided to permit the selection of one of the Map Unit functional elements (GATHER, COMPRESS, MASK/MERGE) or the contents of READ 1 bus or READ 2 bus as inputs to the source stream going to the Vector or Buffer Units. Both busses can move data at the maximum streaming rate, or at slower rates depending on the specified function. For every operand segment transmitted, the S1 and S2 busses must receive an accept signal on their respective control lines before moving a new data quantity onto the bus.

3.3.3.6 WRITE 1 Select

This network provides a simple selection multiplexor for the stream to be written back to memory. No error checking logic is included although the busses carry the SECDED codes.

3.3.3.7 MERGE/MASK Network

The vector functions MERGE and MASK are provided by this element. Inputs are data streams from READ 1 and READ 2 at 128 bits apiece. A disassembly register is provided for each stream to break down the data into 64 or 32-bit operands.

(continued)

3.3.3.7 (Cont.)

Data is moved at the rate governed by the input order vector (sixteen bits at a time) and the specified function:

1. MERGE--The MERGE operation has two modes, replace and shuffle. In the replace mode, the vector input on READ 1 is moved through the element at the rate of four operands per minor cycle. The order vector is moved at the rate of four bits per minor cycle. When a one-bit is found in the order vector a data element from READ 2 is replaced for the corresponding data element in the READ 1 vector, and the next READ 2 data element is moved up to await insertion. Thus READ 2 is moved at the rate of one-bits appearing in the order vector.

   In the shuffle mode, READ 1 is advanced for every one-bit in the order vector and READ 2 is advanced for every zero-bit in the order vector. When a stream is advanced, one operand from that stream is moved into the output stream. In both cases the output stream is moved at the rate of four operands per minor cycle.

2. MASK--In the MASK operation all streams, input and output, are moved at the rate of four operands per minor cycle. In this case, if an operand is not used from the input streams it is thrown away.

SECDED error codes are carried through this element, but no checking or correcting is done there.

3.3.3.8 COMPRESS Network

The COMPRESS network operates in a similar fashion to the MASK/MERGE network, however, it utilizes only one input stream, READ 1, and produces output data at the rate at which one-bits appear in the order vector. The READ 1 stream is moved at the rate of four operands per minor cycle, the order vector is moved at the rate of four bits per minor cycle. In the event that the order vector was completely filled with one-bits, the output stream would move at the rate of four operands per minor cycle. On the other hand, if the order vector was completely vacuous, no output would be produced, but the entire READ 1 vector would be input and thrown away.

3.3.3.9 Order Vector Assembly

One set of vector functions permit the creation of the order or control vector based on arithmetic comparisons performed in the Vector Units. In this mode each Vector Unit transmits 3 control bits indicating the state of comparison of a set of operands in a given clock cycle. This comparison can only be performed by one of the two back-end (final) adders (BADD1 or BADD2) in the Vector Units. The condition codes transmitted are: A=B, A>B, A<B.

The condition codes are then selected by the Map Unit for any combination (A<=B) of conditions and used to form a bit vector indicating the truth or falsity of the selected condition. The resulting order/control vector is then stored into memory via the WRITE 1 bus.

Since the order/control vector is generated at this point, the assembly network also generates the SECDED codes needed for all operands. Order/control vectors are formed at the rate of 8 bits per minor cycle (8 pipeline units at one condition each per minor cycle).

3.3.3.10 Map Control

The map control element provides the interface between the Instruction Issue Unit and the Map Unit. When a map instruction (op code 9D) is detected by the Issue Unit, it is transmitted (along with all suboperation parcels) to the map control element which then makes any necessary register file references, forms the starting addresses, increments, and control signals and transmits them to the appropriate Map Unit elements. The Map Unit is responsible for releasing the Issue Unit to go on issuing further instructions.

Many conditions arise during vector and map operations which can effect the contents of the data flag register. The map control element deskews all such data flag information and makes the appropriate changes in the data flag register.

3.3.3.11 Pipeline Selection

Figure 3.3-8 gives a block diagram overview of the interconnection of the Map Unit and the Vector Units. In this diagram, the Buffer Unit is not shown, but instead is considered imbedded within the respective Vector Units.

As can be seen from the figure, there are actually nine physically distinct Vector Units comprising the Floating-Point Ensemble. Nine input select and eight outut select networks are housed within the Map Unit to provide connections to the Vector Units and the input and output data busses. Only one of the input trunks is shown here, labeled S(0) through S(7), corresponding to the source trunk S1 of emerging from the Map Unit. Also shown is a special data trunk labeled maintenance data, which can be selected into any or all of the nine physical Vector Units. The selection of maintenance data in and maintenance data out is under the control of the Maintenance Control Unit (MCU).

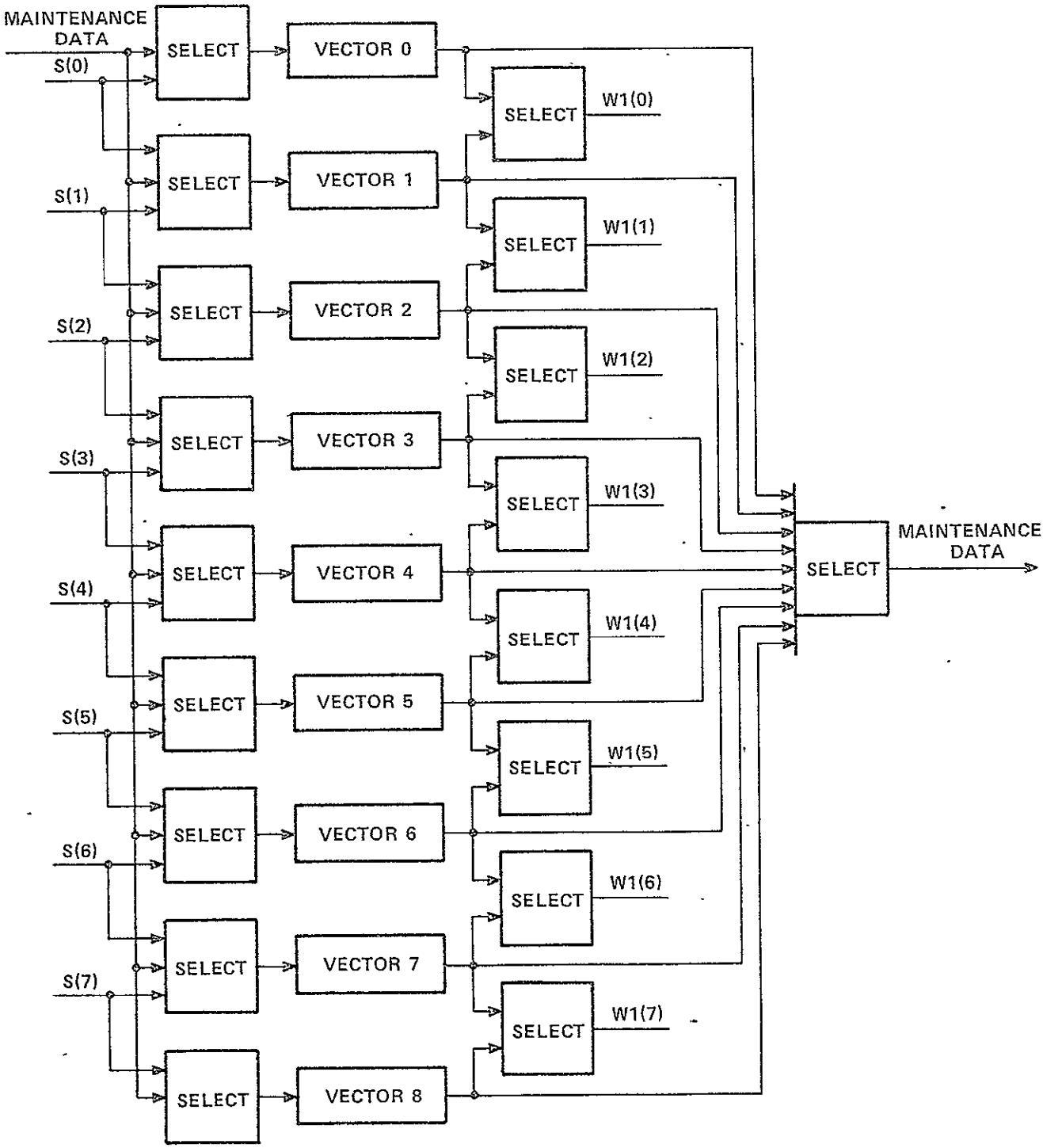


Figure 3.3-8 Map/Vector Interconnection

3.3.3.11.1 Normal Operation of Selection Networks

Upon deadstart of the FMP, the Maintenance Control Unit (MCU) sets up the input data selection networks and output data selection networks for eight pipelines. Normally the pipelines would be configured with Vector 0 through Vector 7 on-line to the input and output data trunks. In addition, the data trunk of the adjacent Vector Unit (in this case S(7)) would be enabled to the extra Vector Unit (in this case Vector 8). The output of Vector 8 would not be selected into WRITE 1 (W1), but could be sampled by the Maintenance Control Unit. The same selection would be made for the S2 (Source 2) bus from the Map Unit. Thus during execution of vector arithmetic instructions, Vector 8 (in this example) would be performing identical operations on data identical to that submitted to Vector 7. Thus the internal arithmetic elements and checking circuitry of the excess unit are continuously exercised.

In the event that the excess unit discovers an error in its own operation (checker failure, parity error or SECDED double error), The Vector Unit will be halted but no stop flag will be sent to any other Units. The Maintenance Control Unit (MCU) will be alerted, however. Under control of the MCU, special data trunks can be connected to the input and output of the excess unit and fault isolation diagnostics executed with selected data being forced into the S1 and S2 ports of the failing unit. This technique permits the on-line maintenance of a failing Vector Unit.

3.3.3.11.2 Error Recovery and Maintenance

In the event that an error is detected in one of the on-line Vector Units, the entire FMP is halted and the job in progress is aborted. Before another job is started the MCU will switch the data bus selects so that the excess unit is introduced into the system, and the failing unit removed. For example, if Vector 4 were to fail and thus be switched off-line, the input selects would be changed so that S(4) would now go to Vector 5, S(5) to Vector 6, and so on with S(7) now gated to the previously offline Vector 8. At the same time the output selects would be changed in similar manner, as well as maintenance communications enabled with Vector 4 through the data busses.

(continued)

3.3.3.11.2 (Cont.)

This scheme permits the use of any Vector Unit as the excess unit, dependng on the MCU controls set up, thus all pipelines can be continuously exercised in an on-line manner thoughout the operating day. In such instances, the Maintenance Control Unit could rotate the assignments between jobs.

3.4 Main Memory

Main Memory is a single-level, random-access memory using bipolar, 4K-bit integrated circuits. The memory words are 78 bits which provide for a 64-bit data word and 7 bits of single error correction double error detection (SECDED) for each 32-bit half-word. The semiconductor memory access time is 40 nanoseconds, where access time is defined as the time from the address reaching memory until data is clocked out of the memory. This memory is directly addressable in either monitor mode or job mode.

The basic Main Memory size is two million words with expansions to four or eight million words available as field upgrade options.

Each two million words of Main Memory contains 16 memory stacks each having 256K 39-bit half-words (32 data bits plus 7 SECDED bits). Each 256K stack is arranged in eight phased banks. In streaming mode, a reference will be made simultaneously to the same address in each of the 16 memory stacks to obtain a super-word (SWORD) of 512 data bits. Memory busy conflict rules take into account the 16 physically independent stacks and the eight-bank phasing within each stack to treat the bank address in each of the 16 stacks as a separate entity. Thus, it could be said that each two million words of Main Memory contains 128 phased half-word banks.

The eight-bank phasing plus the physical distribution of the memory stacks allows memory references to be made at a maximum rate of one every 10 nanosecond minor cycle for each two million words of memory. Thus, the Main Memory has very high data transfer bandwidths:

two million words	= 512 bits/minor cycle
four million words	= 1024 bits/minor cycle
eight million words	= 2048 bits/minor cycle

3.4.1 Memory Stack

The memory stack is packed in a freon-cooled .5 cubic ft. area with 8 banks, each 32K x 40 bits. The FMP utilizes thirty-two bits for data and seven bits for SECDED. There are three board types used in the stack: input control, storage, and output. Figure 3.4-1 shows the module organization which lends itself to massive use of distributed loading and emitter-ANDing and also results in "zero-skew" construction which equalizes signal paths through all memory chips to maintain identical timing throughout the stack.

(continued)

3.4.1 (Cont.)

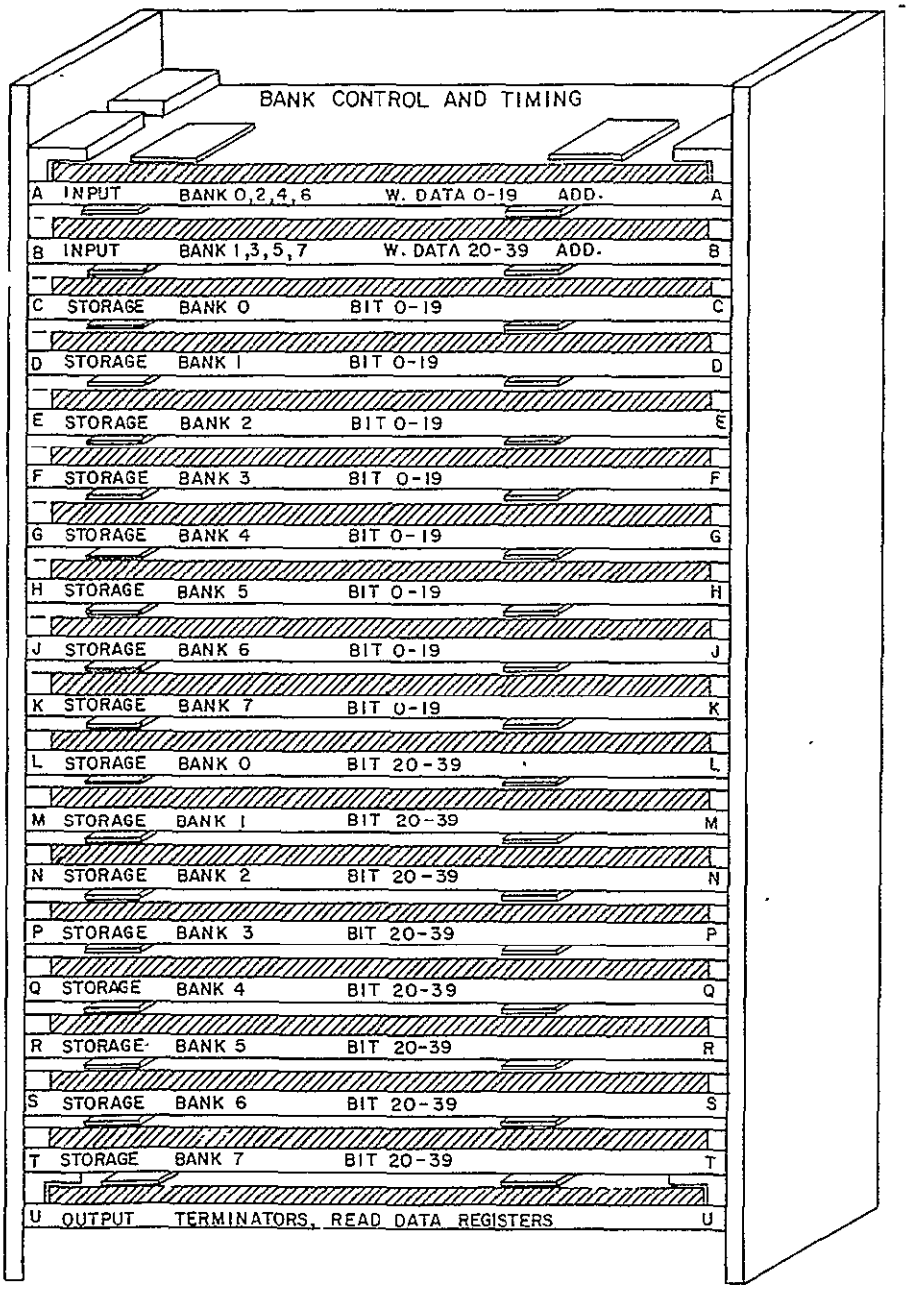


Figure 3.4-1 Memory Stack

(continued)

3.4.1 (Cont.)

108 coax lines connect each memory stack to the Memory Interchange. All signals on the lines except the read data are sent from the interchange to the stack. Below is a list of these lines:

Clock (2) - One for each input board to synchronize the memory stack to the interchange.

Absolute Address (15) - Twelve address bits for the selection of the 4K memory chips and three address bits for the selection of the eight ranks of memory chips.

Bank Address (6) - Three for each input board which are decoded for the selection of the eight banks within a stack.

Stack Request (2) - One for each input board which are decoded for selection of a unique memory stack.

Write Control (2) - One for each input board to inform the stack of a write memory cycle.

Write Data (39) - 39 data bits to memory, 32 for data, 7 for SECDED. Bits 0-19 on the "A" input board, and bits 20-38 on the "B" input board.

Sync (1) - This signal provides a point of time reference for maintenance purposes.

Master Clear (2) - One for each input board.

Read Data (39) - 39 Read data bits from the read data registers on the output board back to the interchange.

3.4.2 Memory Configuration

Memory stacks are located as shown in Figures 3.4-2 and 3.4-3. There are eight stacks per memory section and these sections are positioned around the Memory Interchange, (Figure 3.4-2). Figure 3.4-3 shows where the various half-word segments in a sword of data reside in memory and where stacks reside in a section. Section positioning in this diagram is shown for data clarity and is not the true physical positioning.

(to be supplied later)

Figure 3.4-2 Section Designations FMP Memory

(to be supplied later)

Figure 3.4-3

3.4.3 Memory Interchange

The Main Memory has 512 banks, each bank 39 bits wide (32 + 7 SECDED) by 32K words deep. See figure 3.4-4. The memory has two separate access control networks, each network connected to one-half (256 banks) of memory. The "EVEN" control network addresses and passes data to and from the even numbered 1024-bit groups and the "ODD" control network the odd numbered groups. This is done to enable two separate accesses to memory simultaneously.

The other side of the Memory Interchange are the connections to the Map Unit and the other units that require access to memory - the Scalar Processor and the Map Unit. The memory access is controlled through four memory ports. Three of the ports are dedicated connections to the Map Unit (R1, R2, W1) and the remaining port is time shared between the remaining read/write buses (R3 and swap).

For vector operations R1, R2 and R3 make requests for 2048 bits of data thus requiring both even and odd control networks. For R1 and R2, 2048 bits of data are held and sent 512 bits at a time to the Map Unit. W1 accepts 512 bits and assembles them into 2048 bit groups. This assembly/disassembly means that a port requires access to memory one cycle in every four. Thus the map ports can use up to 3/4 of the memory bandwidth.

The remaining port into memory is divided 3 ways: the Swap Unit, R3 to the Map Unit, and R3 to the Scalar Processor. The Swap Unit will use 128 data bits per cycle and will always address memory sequentially. On a swap request the port will get or store 1024 bits from/to memory. Thus the Swap Unit can make a memory request every eight cycles and data will move to/from the Swap Unit as a 512-bit move every four cycles. The Map Unit uses R3 in either of two modes - to get indexes for the GATHER/SCATTER instructions or to get bit strings (order vectors).

In the first case the Map Unit can use up to 128 bits on a cycle, thus requiring a memory request every eight cycles. In the second case the Map Unit uses up to eight bits on a cycle thus requiring a memory request only every 128 cycles since the port will always request 1024 bits of data. The R3 connection

(continued)

3.4.3 (Cont.)

to/from the Scalar Processor is used for two purposes. The first is to fetch instructions into the Issue Unit and the second is for load and store requests to/from the scalar Register File. Both of these functions are highly asynchronous. The Issue Unit has a buffer to hold up to 4096 bits of instruction so as to enable program loops of reasonable size without making repeated memory requests.

Memory can be accessed in several bit-group sizes - 32 bits (half-word), 64 bits (1 word) 1024 bits (16 words), and 2048 bits (32 words). Each port contains logic to tell memory the access width and the address of the first half-word to access. All accesses must be on proper boundaries for the size requested. Thus, for example, a word access must request an even half-word address. Even though an entire 1024 bits of data may be transferred by a network control, only the data requested will make the memory busy. Thus a request for a full word of data will make two banks of memory busy thereby reducing possible memory conflicts.

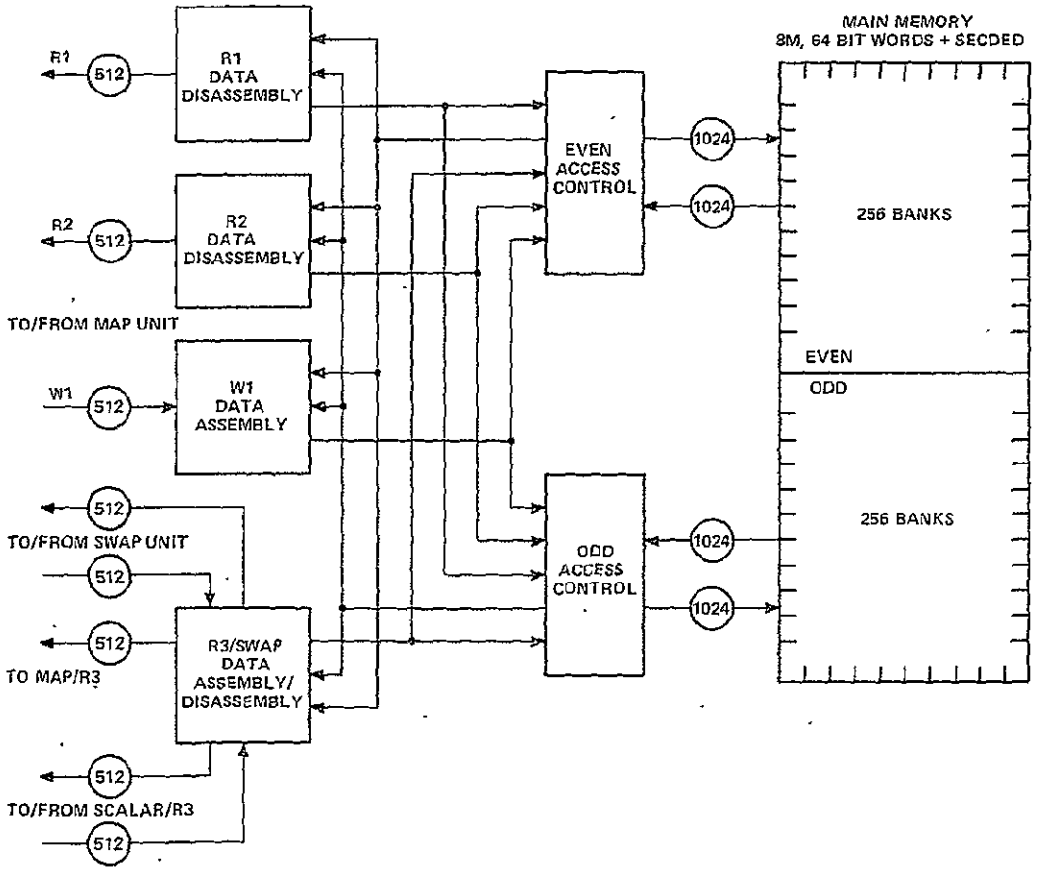


Figure 3.4-4 FMP Memory Interchange and Main Memory

TABLE 3.4-1 MEMORY PORT TRANSFER MODES

MEMORY INTERFACE BUFFER	MEMORY PORT	TRANSFER MODE
R1	Map	Half-word, Word, Sword
R2	Map	Half-word, Word, Sword
W1	Map	Half-word, Word, Sword
R3/Swap	Scalar, Map Swap	Half-word, Word Sword

3.4.4 Memory Degradation

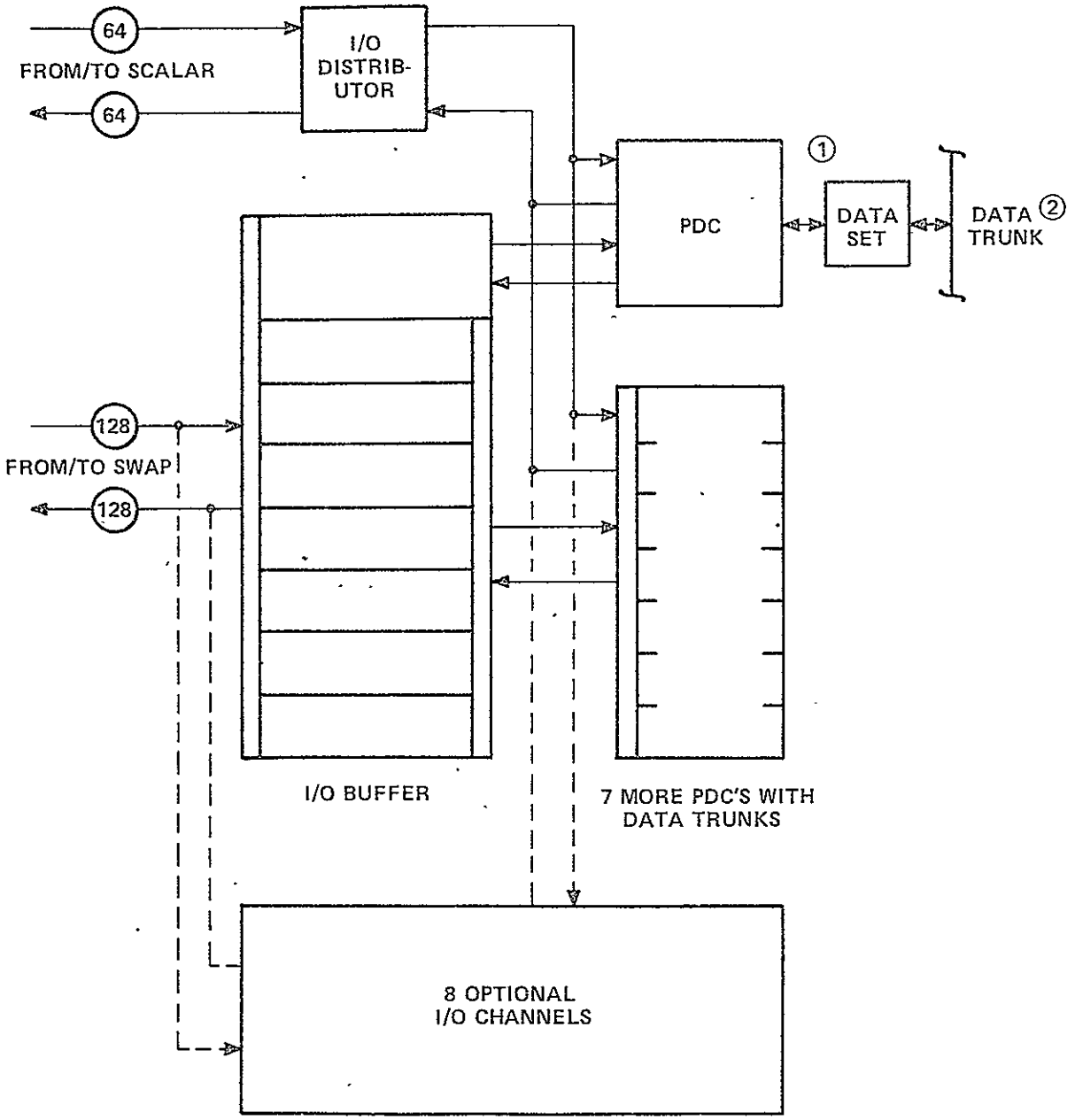
If more than the minimum two-Mword memory is present, degradation may be selected so that the amount of usable memory is less than the total memory on the system. The amount of usable memory is controlled by three degradation bits from the MCU along with a strobe bit.

Memory Sections* Used	DEGRADATION BITS				Usable Memory
A,H	1	1	0	0	
B,G	1	1	1	1	2 Mword
C,F	1	1	0	2	
D,E	1	1	1	3	
A,H;B,G	0	1	X	4	4 Mword
C,F;D,E	0	1	X	5	
A,H;B,G C,F;D,E	0	0	X	6	8 Mword

* See Figures 3.4-2 and 3.4-3

3.5 I/O Channels

The FMP is equipped with a basic set of eight I/O channels, and includes space for an optional eight I/O channels. All input and output is through the Backing Store, via the Swap Unit. Figure 3.5-1 gives a general block diagram of the I/O unit. The input/output characteristics are optimized around large block transfers of data, since small block data handling appears to have a higher overhead associated with memory accesses.



① EACH PDC CAN HAVE UP TO 4 TRUNKS ATTACHED.

② A TRUNK WITH THE ASSOCIATED DATA SET AND PDC CONSTITUTES A CHANNEL.

Figure 3.5-1 I/O Unit

3.5.1 Data Movement

Data can be transmitted from the I/O Unit at the rate of 128 bits every clock cycle. Thus with a 10 nanosecond clock cycle the total I/O bandwidth is 12.8 billion bits per second. However, the I/O Unit must share the Swap Unit bandwidth with Memory to Backing Store swaps, and thus the design goal is for an achieveable, sustained I/O bandwidth of 3.2 billion bits per second.

The I/O Unit is engineered in two parts, CPU and peripheral. The CPU portion includes the I/O distributer for control between the Scalar Processor and PDCs, the I/O buffers and associated channel control not shown. The individual CPU-end capability for tranfers is one 32-bit (plus SECDED) data item transfered per minor cycle per channel. The current peripheral-end bandwidth is 5.0 megabits per second, thus the CPU-end is not being challenged in the initial installation.

Data is moved between the I/O Unit and the Backing Store in 128-bit parcels (quarter-swords). Data is moved between the I/O Unit and the peripheral sections in 32-bit segments, and control information under the command of the monitor mode scalar instruction is communicated between the Scalar Processor and the peripheral section in 64-bit pieces.

3.5.2 Error Checking

All data passed through or buffered in the I/O Unit contains seven bits of SECDED information for every 32-bits of data. SECDED checking and generation are performed in the PDC (peripheral device controller) so that error correction can cover the total path from the Swap Unit out to the peripheral sections.

In the event that a single-bit error is detected by the PDC, the error is corrected and the PDC memory counter for the failing address plus the syndrome bits are locked up for sampling by the MCU. An error flag is sent to the MCU indicating which PDC discovered the error.

(continued)

3.5.2 (Cont.)

If a double-bit (uncorrectable error) is discovered, the error data is latched up, the MCU is flagged, and the PDC attempts to retry the data transfer between the Backing Store and the PDC. If after a number of attempts (set by the installation) the error cannot be recovered, an error message is sent down the network data trunk by the PDC, a code word is sent to the scalar monitor on the 64-bit trunk, and the I/O Unit idles that particular channel. If the MCU discovers more than one channel failing it will force a job abort of the computation in progress and cause the monitor to enter a channel diagnostic mode. All stations attached to the trunk are alerted to the problem and will take appropriate action, including switching to an alternate channel.

Data transferred onto the data trunk carries one or more CRC (cyclic redundancy codes) for error checking. If the PDC receives a faulty transmission, it requests a retry of the full block on the trunk for a number of times. If the block cannot be transmitted the PDC will signal the transmitting station and attempt to retry on an alternate trunk (each PDC can be attached to up to four trunks at a time). The MCU and the transmitting stations are all alerted to any errors, whether transient or fatal, and through software will take appropriate actions.

3.5.3 Addressing

All data transfers to and from the I/O Unit are by 128-bit parcels; however, the minimum block size transmitted from the Backing Store is defined for each channel, but cannot be less than 512 words. All data arriving at the I/O Unit is held in a large buffer. This buffer can house from 32,768 to 262,144 words depending on the bandwidth requirements of the peripheral subsystem on a given channel. Eight channels operating at full rate would require a full block to be held at a time per channel to minimize interference with other Backing Store requests. All data enters the homogeneous memory buffer, which is allocated by a local control in the I/O Unit. Since a variable amount of buffer can be allocated dynamically, the buffer can be of modest size, with various I/O channels given large portions as they need them for large transfers.

3.5.4 The PDC

See report Appendix C

3.5.5 The Trunk

See report Appendix D

3.6 Maintenance Control Unit (MCU)

The MCU is an autonomous Maintenance Control Unit connected to the computer via a CDC FMP I/O channel with access to special internal interfaces. These interfaces allow it to regulate information flow, control pulses, and monitor performances of the computer. The MCU consists of a control unit, line printer, card reader, and a disc drive and it provides for system dead-start and system performance monitoring. Special connections to the computer give the MCU the capability of monitoring system performance. Diagnostics and preventive maintenance are facilitated by this section.

There are three operating modes for the MCU.

1. The first mode is under operation of a diagnostic maintenance program to locate faults and malfunctions within the MCU.

2. The second mode of operation is running diagnostic routines on the FMP. The MCU loads diagnostics, ranging from a simple command test to a very sophisticated diagnostic catalog routine, controls and monitors the operations of the diagnostics, and displays the results of the tests via the display unit or line printer.

3. The third mode of operation is System Operation. Here again, the MCU loads the Operating System Software into the FMP and controls and monitors its operation. In this on-line mode of operation, the MCU concerns itself with: autoloading the central processor and first level stations, running on-line diagnostics, monitoring CPU faults, and restarting the central processor after hang-ups.

3.6.1 MCU/CPU Interface

The MCU connects to the FMP via a network trunk. It interfaces 16 buffers, internal to the FMP, called MCU/CPU channels - ATB being outgoing buffers and BTA being access channels. Tables 3.6-1 through 3.6-8 show the channels from the CPU to the MCU (ATB) and Tables 3.6-9 through 3.6-16 show the channels from the MCU to the CPU (BTA). Each table shows the channel bit number, and function of each bit for a channel.

TABLE 3.6-1 CHANNEL ATB1

Bit No.	Function
0	Bit 0 Current Instruction Address Register
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

TABLE 3.6-2 CHANNEL ATB2

Bit No.	Function
0	Bit 16 Current Instruction Address Register
1	17
2	18
3	19
4	20
5	21
6	22
7	23
8	24
9	25
A	26
B	27
C	28
D	29
E	30
F	31

(continued)

3.6.1.1 (Cont.)

TABLE 3.6-3 CHANNEL ATB3

Bit No.	Function
0	Bit32 Current Instruction Address Register
1	33
2	34
3	35
4	36
5	37
6	38
7	39
8	40
9	41
A	42
B	43
C	44
D	45
E	46
F	47

TABLE 3.6-4 CHANNEL ATB4

Bit No.	Function
0	Bit 0
1	1 Display Register - Displays the
2	2 register selected by bits C-F of
3	3 channel BTA1 in the MCU.
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

(continued)

3.6.1.1 (Cont.)

TABLE 3.6-5 CHANNEL ATB5

Bit No.		Function
0	Bit 16	Display Register - Displays the register selected by bits C-F of Channel BTA1 in the MCU.
1	17	
2	18	
3	19	
4	20	
5	21	
6	22	
7	23	
8	24	
9	25	
A	26	
B	27	
C	28	
D	29	
E	30	
F	31	

TABLE 3.6-6 CHANNEL ATB6

Bit No.		Function
0	Bit 32	Display Register
1	33	
2	34	
3	35	
4	36	
5	37	
6	38	
7	39	
8	40	
9	41	
A	42	
B	43	
C	44	
D	45	
E	46	
F	47	

(continued)

3.6.1.1 (Cont.)

TABLE 3.6-7 CHANNEL AT87

Bit No.	Function
0	Bit 48 Display Register
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
A	58
B	59
C	60
D	61
E	62
F	63

TABLE 3.6-8 CHANNEL AT88

Bit No.	Function
0*	Memory SECDED Fault or Instruction Stack Parity
1*	Microcode Parity Fault
2	Not Used
3*	Absolute Sword Bounds Hit
4*	Event Stop
5*	Single SECDED Error
6	CPU Clock - Used for gating data back to the CPU. The MCU cannot read this line.
7	Monitor Mode
8	Temperature - Dew Point Alarm
9	Not Used
A	Section Power Fail
B	60 Hz Input Power Fail, M.G.1
C	60 Hz Input Power Fail, M.G.2
D	Not Used
E	CPU Idle
F	CPU Stopped

* These lines indicate why the CPU has stopped.

3.6.1.2 Channels from MCU to CPU

TABLE 3.6-9 CHANNEL BTA1

Bit No.	Function
0	MAC Master Clear - Master Clear to Memory Interchange, and Main Memory only. This includes the I/O channels. This signal must be set a minimum of 3 microseconds.
1	Stop - CPU will stop before next instruction issue.
2 *	Step - Execute one instruction. Store the register file and the invisible package (job mode only); then stop. Faults must be cleared before the computer can be stepped.
3 *	Run - Start CPU from manual stop or fault stop. Faults must be cleared before computer can be started.
4 *	Store Register File - The Register File is stored starting at address 0000_{16} in monitor mode and address 4000_{16} in job mode.
5 *	Load Register File - The Register File is loaded starting at address 0000_{16} in monitor mode and address 4000_{16} in job mode.

* Computer must be stopped before executing these commands.

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-9 CHANNEL BTA1 (Cont.)

Bit	Function
6	CPU Master Clear - Master Clear to Scalar Unit, Stream, and Floating Point only. Memory Interchange, I/O Channels, and Main Memory are not included. This signal must be set a minimum of 3 microseconds.
7	Clear Fault Conditions - This signal clears the following conditions and allows the computer to be restarted with a run signal (bit 3): a. SECDED Double Error Condition b. MIC Memory Parity Fault c. Sword Bounds Hit d. The Bounds Hit Address is released. e. Reference to Illegal Address in Stream Microcode. f. Instructional Stack Parity Error
8	Clear SECDED Single Error, SECDED Fault Address and Syndrome Bits.
9	MCU Sync.- This signal is used in the CPU to gate the CPU data back to the MCU. When reading the display registers, the MCU Sync. signal must be set after the read signal is set.
A	Select SECDED Error Mode Two.
B	Read - Transfer selected register and CIAR into the Display Registers.
C D E F	Display Register Selection See Section 3.6.4.2

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-10 CHANNEL BTA2

Bit		Function
0 *		Latch Memory Size Code
1 *	Static ∧	Interrupt Gate - When this signal is a "1", time interrupts and external interrupts will only be processed between instructions.
2 *		Memory Size Degrade Code
3 *		000 = 2 Meg Memory
		001 = 2 Meg Memory, Force Section 1 --> Section 0
		010 = 2 Meg Memory, Force Section 2 --> Section 0
4 *		011 = 2 Meg Memory, Force Section 3 --> Section 0
		100 = 4 Meg Memory
		101 = 4 Meg Memory, Force Upper 4 Meg --> Lower 4 Meg
		110 = 8 Meg Memory
5 *		Select Mainframe Clock Freq.
		000 = Nominal
6 *		001 = Increase clock freq. (1)
7 *		010 = Decrease clock freq. (1)
		011 = Select variable freq. (adjustment on oscillator pak)
		100 = Increase clock freq. (2)
		101 = Increase clock freq. (3)
		110 = Decrease clock freq. (2)
		111 = Decrease clock freq. (3)
		NOTE: If clock frequency codes 4-7 are used, then code 3 is not available. Either codes 0-3 or 0-2 and 4-7 are available.
8 *		Delay Trailing Edge - Delay the trailing edge of all of the clocks on the panel which is specified by bits 11-15 of Channel BTA2. If bit 8 and bit 9 are set, only the odd or even clock, on a panel are moved depending on bit A.

* Computer must be stopped before executing these commands.

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-10 CHANNEL BTA2 (Cont.)

Bit		Function
9 *		Delay Leading Edge - Delay the leading edge of all of the clocks on the panel which is specified by bits 8-F of Channel BTA2. If bits 8 and 9 are set, only the odd or even clocks on a panel are moved depending on bit A.
A *	V Static	"0" - Move even clocks (see description for bit 8 or 9). "1" - Move odd clocks.

* Computer must be stopped before executing these commands.

Bit	Function		
B (2 4) C (2 3) D (2 2) E (2 1) F (2 0)	Panel Designator for Clock Margins - Bit 8 is the left-most bit of the designator. The designators are defined below.		
	Designator (16)	Panel(s)	
	00		
	01		
	02		
	03		
	04		
	05		
	06	(to be supplied later)	
	07		
	08		
	09		
	0A		
	0B		
	0C		
	0D		
	0E		
	0F		

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-10 CHANNEL BTA2 (Cont.)

Bit	Function
10	
11	
12	
13	
14	
15	
16	(to be supplied later)
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-11 CHANNEL BTA3

Bit No.	Function
0	Not Used.
1	Send an external flag on the channel specified by the Channel Select Code in bits 4-8. (*1)(*2)
2	Set Channel Disable on the channel specified by the Channel Select Code in bits 4-8.(*1)(*3)
3	Clear Channel Disable on the channel specified by the Channel Select Code in bits 4-8.(*1)(*3)
4 | 5 > 6 | 7 /	Channel Select Code. A code of 0 thru F selects a channel (16_{10} thru 16_{10}) for the operation specified in bits 1, 2 and 3.(*1) Bit 7 of BTA3 is bit 3 of the Channel Select Code.
8	Select All Channels (0_{10} thru 15_{10}) for the operation specified in bits 1, 2 and 3.(*1)
9	Stop on SECDED Single Error Detection.
A	Disable Stop on SECDED Double Error Detection.
B	Block External Interrupt
C	Disable Error Correction on all Read Buses.
D	Swap Register File Read on Exchange.
E	Not Used
F	Not Used.

(continued)

3.6.1.2 (Cont.)

(*1) The Channel Select Code bits 4-8 must be set before any commands are sent, and it must remain set until after the command has dropped.

(*2) The External Flag is transmitted to the device on the I/O channel corresponding to the code in bits 4-8. External Flag instructs the device to autoload.

(*3) The Channel Disables are transmitted to the I/O Unit. If the disable line for a channel is set, no backing store references will be allowed from that channel. Data transfers can proceed in and out of the channel buffer in an end-around type of operation.

TABLE 3.6-12 CHANNEL BTA4

Bit	Function
0	Checkword bit 0
1	1
2	2 — Used for toggling I/O
3 >	3 — Checkword bits 0-6
4	4
5	5
6	6
7	Block Write Enable on SECDED Error
8	Not Used
9	Not Used
A	Force Reg. File Store at bit address 20,000 on Initial Exchange
B	Force Instruction Stack Parity
C	Enable I/O Simulator
D	Initiate I/O Simulator on Channel Flag
E	Not Used
F	Not Used

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-13 CHANNEL BTA5

Bit	Function
0 1 2 > 3 4	Not Used
5 6 > 7	Bounds Limit Load Code 0 = Null 1 = Load Bits (35-42) Upper Bounds 2 = Load Bits (51-58) Upper Bounds 3 = Load Bits (43-50) Upper Bounds 4 = Null 5 = Load Bits (51-58) Lower Bounds 6 = Load Bits (35-42) Lower Bounds 7 = Load Bits (43-50) Lower Bounds
8 9 A B > C D E F	Bounds Address Bits Due to the operational characteristics of the maintenance interface, only one bit of the code can be changed at one time. Address bits must be loaded in such a manner as to leave the Load Code bits undisturbed. Address bits are transferred on the Leading Edge of a code change, the address bits must be set up before a code change occurs. Address bits are Loaded as follows, starting and ending with a Null Code: Code = 0 Null Code = 1 Set up Bits (35-42) Upper Bounds Code = 3 Set up Bits (43-50) Upper Bounds Code = 2 Set up Bits (51-48) Upper Bounds Code = 6 Set up Bits (35-42) Lower Bounds Code = 7 Set up Bits (43-50) Lower Bounds Code = 5 Set up Bits (51-58) Lower Bounds Code = 4 Null Bounds limits are absolute, physical half-word addresses. Bits (35-36) and (55-58) must be zero.

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-14 CHANNEL BTA6

Bit	Function	
0	Check bounds on memory reads	If bits 0 and 1 or bits 2 and 3 are zero, no bounds hits can occur.
1	Check bounds on memory writes	
2	Check bounds on CPU references	
3	Check bounds on channel references	
4	Stop CPU on bounds hit	
5	Enable bounds check - The bounds addresses and conditions must be set up before the enable is set.	
6	Count A - Monitoring Counter A is enabled while this line is a "1" and held clear when this line is a "0". The proper counter specification and bits 8-E of channel BTA6 must not change while this line is up.	
7	Count B - Monitoring Counter B is enabled while this line is a "1" and held clear when this line is a "0". The proper counter specification and bits 8-E of channel BTA6 must not change while this line is up.	
8	Clear counter (see code 6 in Section 3.6.4.2).	
9	Stop CPU on Counter A Increment	
A	Stop CPU on Counter B Increment	See Section 3.6.4.1.3

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-14 CHANNEL BTA6 (Cont.)

Bit	Function
B	Enable Carry into A1
C	Enable Carry into A2
D	Enable Carry into B1 ---See Section 3.6.4.1.2
E	Enable Carry into B2
F	"0" - Load Counter A Event Selects and Gates (Channel BTA Bits 0-F). "1" - Load Counter B Event Selects and Gates (Channel BTA Bits 0-F). This bit should be set to the proper counter before the count specification is set into Channel BTA7.

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-15 CHANNEL BTA7

Bit	Function
0	Event Select for Counter A1 and B1 -See Section 3.6.4.1 for codes
1	
2	
3	
4	
5	
6	Event Select for Counter A2 and B2 See Section 3.6.4.1 for codes
7	
8	
9	
A	Not Used
B	Selected Job Gate
C	Monitor Mode Gate
D	Job Mode Gate
E	Data Flag 56 Gate
F	Data Flag 57 Gate

MCU Event Counter Gates --See Section 3.6.4.1.1

(continued)

3.6.1.2 (Cont.)

TABLE 3.6-16 CHANNEL BTA8

Bit	Function
0	8-bit function select code. Bit 0 is the left-most bit of the code. See event code 12_{16} in Section 3.6.4.1.
1	
2	
3	
4	
5	
6	
7	
8	8-bit function mask. Bit 8 is the left-most bit of the mask. See event code 12_{16} in Section 3.6.4.1.
9	
A	
B	
C	
D	
E	
F	

3.6.2 MCU/Microcode Memory Interface

Upon power up of the FMP all microcode memory contents are undefined since that memory is built of RAM circuits with volatile storage. Each of the FMP microcodes can be loaded by an MCU function which is sent over the FMP I/O channels from one of the processors acting as the MCU. A special trunk address identifies the special I/O channel which does not transfer data to the Backing Store, but instead provides control information for the FMP, and retrieves status information from the FMP from one or more of the internal maintenance channels contained within the FMP. One of the maintenance functions is the loading of microcode to each of the microcode memories. Each block of microcode received by the MCU interface is checked for data errors (using the CRC code in the trunk message) and sent to its respective microcode memory system. Each block is preceded by a unique 16-bit address which identifies the particular microcode destination.

3.6.2.1 Microcode Units and Addresses

(to be supplied later)

3.6.2.2 Microcode Error Checking

Under control of the MCU interface control signals, a microcode memory can be loaded with data from the trunk. The data carries its own parity bits (one per word) which are generated by the assembler at the time the microcode is created. This block can be read out of each microcode memory sequentially by the MCU interface so that the memory can be checked. Each word read is parity checked and if an error occurs the location of the failing word is unloaded by the MCU interface via the P counter of that microcode.

During normal startup procedures, each microcode memory is loaded in turn with its unique microcode and the entire contents are swept out on an MCU-controlled, sequential read operation to verify the integrity of that memory.

(continued)

3.6.2.2 (Cont.)

During operation of the FMP, each microcode access is parity checked. If a parity error occurs in any microcode, the MCU is signalled via the network trunk and the FMP CPU is stopped as soon as possible. The location of the error P counter and the address of the failing microcode unit are then provided to the MCU interface for transmission to the MCU processor on the trunk.

3.6.2.3 MCU Interface Channel Bits

(to be supplied later)

3.6.3 Microcode Memory Channel Programming

(to be supplied later)

3.6.3.1 N/A

3.6.3.2 N/A

3.6.3.3 Typical Microcode Interface Function Codes

For all channel functions the address that accompanies the function and the null function are ignored. The following 3-bit function codes control the microcode memory:

TABLE 3.6-17 B TYPICAL FUNCTION CODES (MIC. MEM.)

Bit 0	Bit 1	Bit 2	Function
0	0	0	Null - Automatically sent by the MCU interface as the second half of any other function.
0	0	1	Read Memory - Read a block of microcode memory from the current microcode "P" address.
0	1	0	Write Memory - Write a block of microcode memory from the current microcode "P" address.
0	1	1	Not normally used but will perform the same as a EOP.
1	0	0	Data - Automatically sent with the data during a write microcode memory operation.
1	0	1	Read Status - Read the current microcode status. See Section 3.6.3.5 for explanation.
1	1	0	Write Switches - The switches provide control of microcode execution. See Section 3.6.3.4.
1	1	1	EOP - End of Operation clears the interface of all previous functions and also clears the counter that controls the data fan-in and fan-out to/from the channel.

3.6.3.4 Microcode Switches

Microcode switches are 1-bit control terms used to control the microcode memory. Each switch is one bit of the Write Switch Control Word. The 110 function code (write switch) causes the microcode memory to store the Write Switch Control Word in a register. The MCU interface receives this data from the I/O trunk and sends it to the microcode control. The following is a definition of each switch function and a description of its use.

1. Switch Function Definitions

TABLE 3.6-18 MICROCODE SWITCH FUNCTIONS

Bit	Function
0	Go Microcode - Strobing this bit will cause microcode to start execution at the current microcode "P" address.
1	Kill - Setting this bit will stop any microcode instructions executing at the time the bit is set. The instruction will come to a normal halt with "P" pointing to the next word to be executed. Execution can be resumed by setting bit 0.
2	Sense Switch - Any microcode program can sense the condition of this switch for program control (used mainly by diagnostics).
3	P to 0 - Strobing this bit will force the "P" register to zero. Kill should be set either previously or in the same word so as to come to a normal halt.
4	Clear Checkpoint - Strobing this bit will clear the check point flip-flop.
5	Drop Control-Setting this bit disables control of the CPU and the I.C.s from microcode. This will prevent undefined CPU operation due to a microcode memory test.

(continued)

3.6.3.4 (Cont.)

TABLE 3.6-18 MICROCODE SWITCH FUNCTIONS (Cont.)

Bit	Function
6	Change Status Word 2 Definition - Bits 8-F of Status Word 2 become bits 0-7 of an IC register. See Section 3.6.3.5.
7	Enable control of the register logical pipe from microcode.
8	Function for Scalar Microcode not yet defined.
9	Sweep Scalar Microcode
A	Write Scalar Microcode - Must be set to Write. Scalar Microcode, disables microcode write enables.
B	1 = Enables Scalar Microcode to sweep PM00 0 = Enables Scalar Microcode to sweep PM01
C-F	Functions for Scalar Microcode not yet defined.

(continued)

3.6.3.4 (Cont.)

2. Use of Switch Functions

1. Switch Functions 0, 3 and 4 are one-shot functions. This is accomplished by having the required bit set in the even 16-bit word of a transfer and clear in the odd 16-bit word. If the bit is set into both halves of a 32-bit transfer, for instance, the function will be performed in that transfer but will possibly be ignored if sent in the next transfer.

2. Switch Functions 0 and 3 are delayed by one cycle so that other functions sent in the same data word have time to propagate; i.e., "kill" and "P to 0" together are legal as are "sense switch" and "go microcode". Other combinations are also legal.

3. Switch Functions 1, 2, 5, 6, 7, 9, A and B are latching functions that are caught and held until another function is sent. Note, however, that a single function consists of two or more data transfers -- each transfer clearing and loading over previous data transfers so that a switch that is meant to be valid both during and after the function must be sent in both halves of a 32-bit data transfer and any latching function that is supposed to remain valid through another "send switches" function must be sent again with that function, again present in both halves of the 32-bit data word.

3.6.3.5 Stream Microcode Status

The input of status to the MCU can be of any number of words; but all words after the first word will be word 2 of the status.

The input of status does not have any effect on microcode or microcode controls.

Upon the receipt of a 101 (read status) channel function code, the MCU interface will load the channel with the following status words.

TABLE 3.6-19 MICROCODE STATUS

Bits (Word 1)	Meaning
0	Checkpoint - Software uses this bit to indicate to the MCU that the microcode has reached some predefined status found an error or reach some predefined address for debugging, for example.
1-4	Flags - The current state of flags 0, 1, 2, 3.
5-F	P - The current state of the P (microcode address) register.*

Bits (Word 2)	Meaning
0	Run - This bit will be used to indicate the microcode is executing.
1-4	J1 - The current state of the least significant 4 bits of the J1 register.
5-F	J2 - The current state of the J2 register. (See bit 6 of the switch function control word).

*The contents of P do not indicate the address at which microcode has stopped until the second minor cycle after the RUN bit has gone to zero. Thus it is necessary to read the status word twice, once to determine that microcode is not running, and once to read P.

3.6.3.6 Interface Sequences

After selection of the MCU interface the following are examples of possible control sequences.

TABLE 3.6-20 INTERFACE SEQUENCES

Step	Code	Sequence (Stream Units Write Microcode)
A	111	EOP - To clear the interface. Initiate (bit 0) should not be sent with any EOP function.
B	010	Write Mode - The address with this function is ignored; the write will proceed from the current P address.
C	100	Data - Data sent to microcode must (except on the last transfer) be sent in integer multiples of microcode words. One microcode word is 14 16-bit transfers. Data will be lost and/or rearranged if this is not observed.
D	111	EOP
E		Repeat from Step B as many times as necessary to complete transfer of the block of data.
Step	**Code**	**Sequence (Stream Units Read Microcode)**
A	111	EOP
B	001	Read Mode - The address is ignored.
C		Input the data. The same caution as in Write Microcode Step C applies. Data starts from the current microcode P address.
D	111	EOP
E		Repeat from Step B as many times as necessary.

Note: If the last operation performed in a sequence is an EOP, the next sequence does not have to start with another EOP.

(continued)

3.6.3.6 (Cont.)

After selection of the MCU interface the following are examples of possible control sequences.

TABLE 3.6-20 INTERFACE SEQUENCES (Cont.)

Step	Code	Sequence (Stream and Scalar Write Switches)
A	111	EOP
B	110	Set Switch Mode - The address is ignored.
C	100	Data - Although one 32-bit transfer is the normal data length, there is no restriction on data length if the extra data length can be useful - repeated starts for instance.
D	111	EOP

Step	Code	Sequence (Stream Read Status)
A	111	EOP
B	101	Set Status Mode - The address is ignored.
C		Input Data - All data after the first word is status word 2.
D	111	EOP

Note: If the last operation performed in a sequence is an EOP, the next sequence does not have to start with another EOP.

(continued)

3.6.3.6 (Cont.)

TABLE 3.6-20 INTERFACE SEQUENCES (Cont.)

Step	Code	Sequence (Stream and Scalar Write Switches)

Step	Code	Sequence (Write Scalar Microcode)
A	111	EOP - To clear the interface
B	010	Write Mode - Bits 0-8 of the second 16 bits of the address, selects daughter boards 0-8, respectively. The first 16 bits of the address are ignored. The write will proceed from the current P address.
C	100	Data - Bits 0-3 are Write Enables and bits 4-15 are Data. The microcode address is incremented by one for each 16 bit quantity sent by the MCU.
D		Repeat step C until the selected Auxiliary Board has been loaded (normally 1024 16-bit words).
E	111	EOP
F		Repeat from step B to load other Auxiliary Boards.

Note: If the last operation performed in a sequence is an EOP, the next sequence does not have to start with another EOP.

3.6.3.7 Writing or Sweeping Scalar Microcode Memories

The scalar microcode consists of 5 memories; PM00, PM01, HM00, DM00 and GM00. Although each operates independently during CPU instruction execution, they are all addressed simultaneously during writing or sweeping operations.

3.6.3.7.1 Scalar Microcode Memory Write Operations

For write operations, the write enables at each auxiliary board control which auxiliary board and which address within an auxiliary board is to be written. Since 12 bits of data are written at a time, the write enables are also responsible for choosing which 12-bit portion of a microcode address is to be written.

Under the control of the write enables and auxiliary board select, one auxiliary board is written at a time. The address registers on the auxiliary boards will first be set to 00_{16} and then cycled thru FF_{16} and then back to 00_{16} while writing one-fourth (or twelve bits) of an auxiliary board. The write enable will then change to address the next twelve bits of the particular auxiliary board and the address register will again cycle through all addresses. This operation will occur four times on each of the 9 auxiliary boards.

It is possible, by bringing up all 9 auxiliary board selects and all write enable bits, to write all bits of all auxiliary boards in one write of 100_{16} words (except PM00/PM01). Each 12-bit segment of the 48-bit word will be duplicated. This would be done only as a maintenance aid for pattern generation during either write or sweep operations.

3.6.3.7.2 Scalar Microcode Memory Sweep Operations

Sweeping of the scalar microcode memory is an operation to be done to detect a parity error on any of the 9 microcode auxiliary boards. The operation simply consists of referencing all 9 auxiliary boards simultaneously with the same address register. Since there is one parity bit per auxiliary board per microcode memory, any parity error or errors will be isolated to the failing auxiliary board or boards.

The control signals necessary to perform the sweep operation are Sweep (switch function bit A), Enable PM00/PM01 (switch function bit B) and Clear Fault. Sweep should be enabled during the entire sweep operation. Enable PM00/PM01 selects PM00 or PM01 microcode memories and Clear Fault will clear any parity errors caused by the sweep operation. If Clear Fault is sent while sweep is still set, not only will the parity fault condition be cleared but sweeping will continue. However, since the sweep address, upon a parity fault, is 3, 4 or 5 addresses ahead of the actual parity fault address, sweeping immediately after a parity fault will "skip" 3, 4 or 5 addresses respectively. For example, if the parity fault address is 22 on PM00 then addresses 23, 24 and 25 will be skipped.

The register used to reference all of the auxiliary boards during the sweep operation is cleared before and after the sweep operation. Thus, sweeping starts with address zero and, because of the time delay in detecting parity errors, will end beyond that address which caused the parity error. For example, if the parity error occurred on HM00 at address 125 then the address displayed at the MCU will be 130 and is therefore 5 ahead.

The following list specifies how far ahead of the parity fault address the sweep address will be.

GM00	5 ahead
DM00	4 ahead
HM00	5 ahead
PM00	3 ahead
PM01	3 ahead

3.6.4 Monitoring System Activity by the MCU

The MCU monitors the output of two display registers as its main medium of monitoring system activity. One display register contains the output of the Current Instruction Address Register (CIAR). The other display register contains the output of the register selected by the MCU. A 4-bit code sent from the MCU selects which register the display register will present. In addition to monitoring the display register, the MCU can also monitor the microcode memory status and other CPU status.

3.6.4.1 Monitoring with Counters

For monitoring purposes, the CPU has four 16-bit counters. Each of these counters can be connected to an event line selected by a command from the MCU. See figures 3.6-1 and 3.6-2. A list of events which can be counted and their corresponding select codes is given in Table 3.6-21. For ourposes of discussion, one pair of 16-bit counters is referred to as Counters A1 and A2. The other pair is labeled B1 and B2. Counter A and Counter B are completely independent and cannot be tied together; however, they do share the same input event lines and gate lines. The counters can be read by selecting them for input into the MCU display register. They can also be combined in various ways to form one or two 32-bit counters. This reconfiguration is accomplished via the carry lines from the MCU. The counters are enabled by a number of hardware and software gates selected with a mask from the MCU. The MCU has the option of stopping the CPU count condition. This option is exercised by use of the stop lines.

(continued)

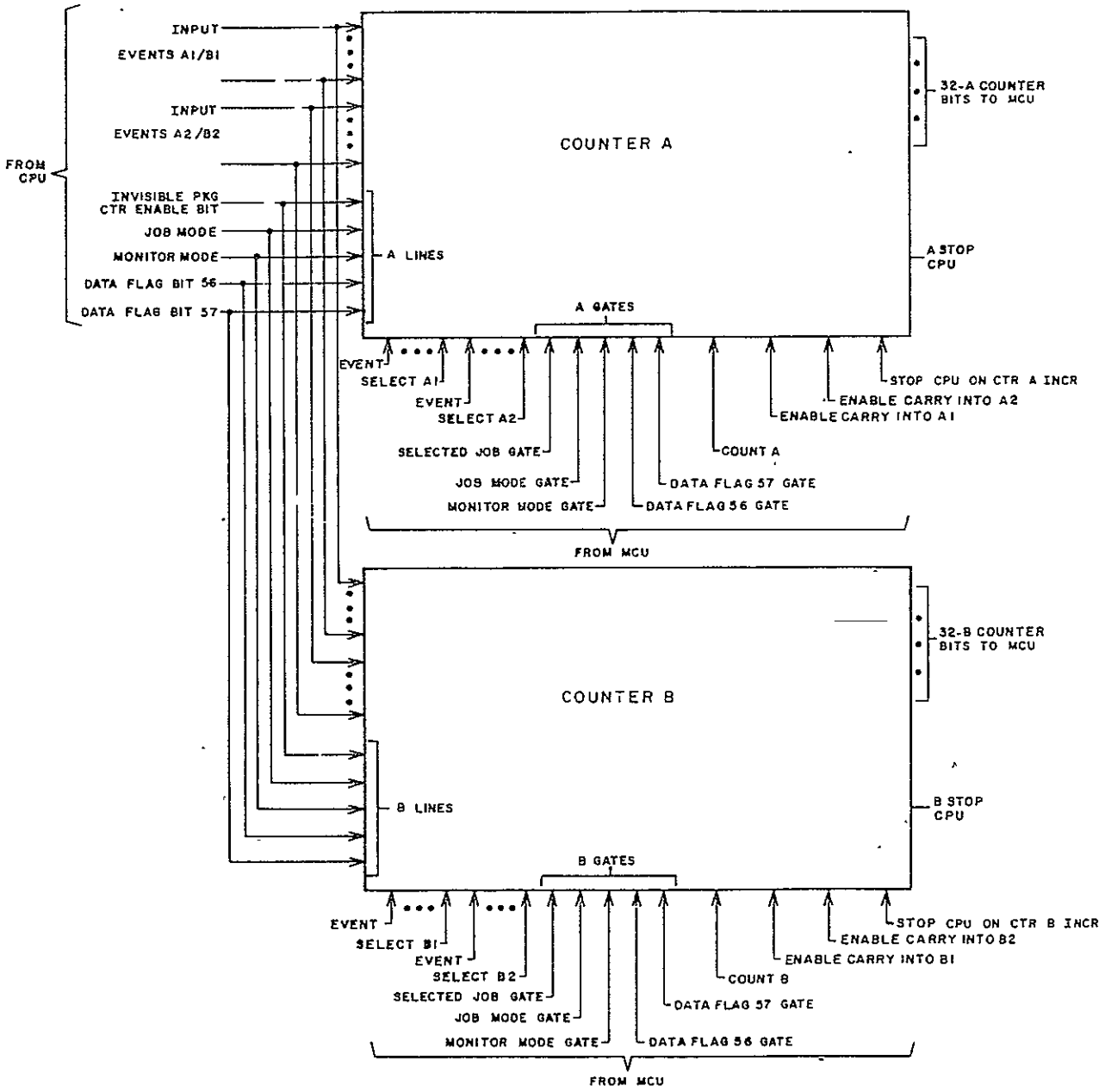


Figure 3.6-1 Block Diagram of Counter Logic Lines (continued)

3.6.4.1 (Cont.)

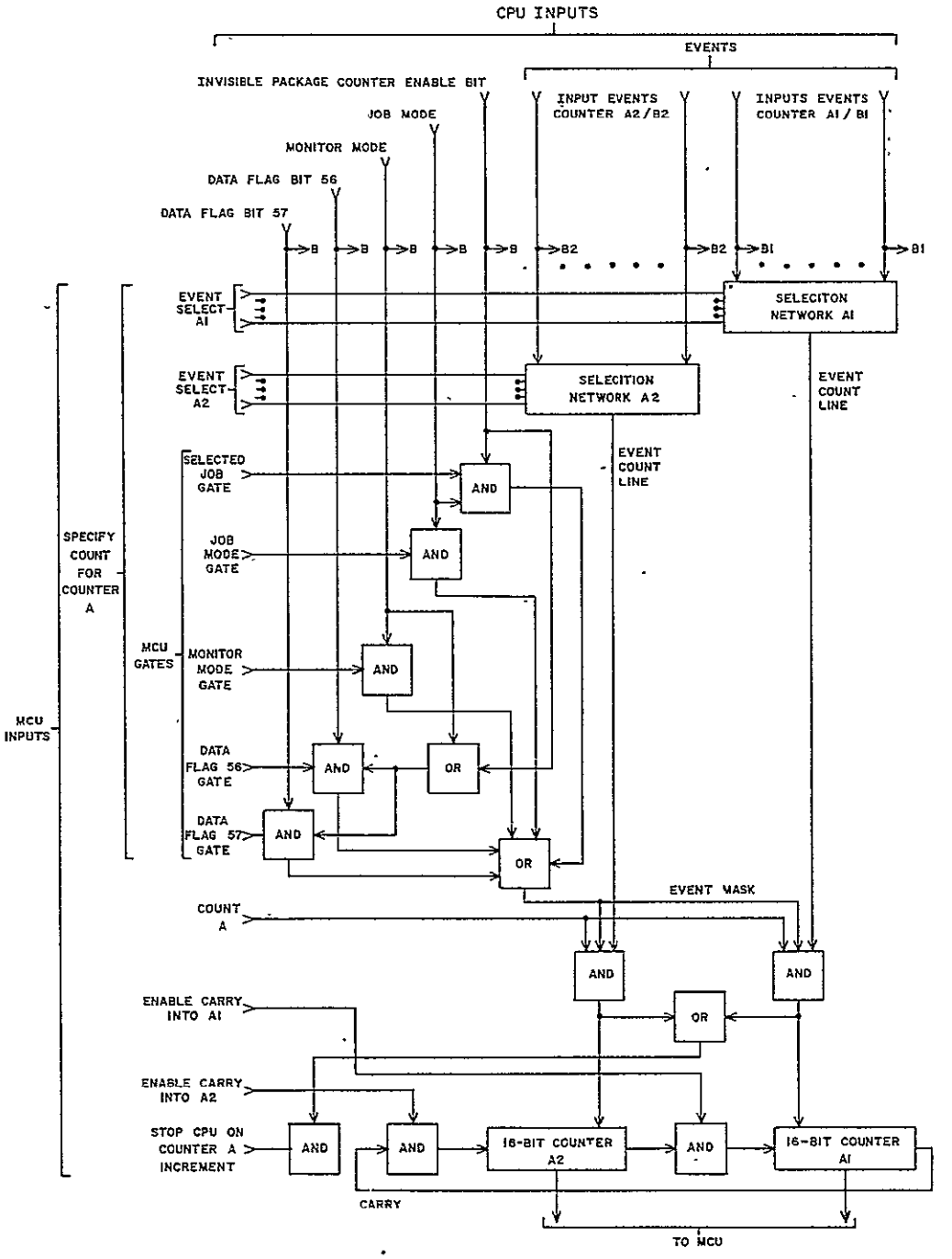


Figure 3.6-2 Block Diagram of Counter A
(continued)

3.6.4.1 (Cont.)

TABLE 3.6-21 COUNTER EVENTS

*Codes 16 Counter A1/B1	*Codes 16 Counter A2/B2	EVENTS
01		Number of branches out of instruction stack.
	01	Number of branches in instruction stack.
04		Number of times microcode field MON = 1 is selected.
	04	Number of shortstop path usages.
05		Not Used
	05	Not Used.
09		Number of normal channel backing store requests.
	09	Number of normal channel backing store requests accepted.
0A		Number pf CPU memory requests.
	0A	Number of CPU memory requests accepted.
0B		Total number of memory requests.
	0B	Total number of memory requests accepted.
11		Number of minor cycles from selected instruction issue to next, non-selected issue. The counter will begin counting when an instruction whose function code meets the conditions described in code 12 below, is loaded into IR0. It will stop counting when the first following instruction which does NOT meet the conditions is loaded into IR0.

*These are 5-bit codes, expressed in hexadecimal.

(continued)

3.6.4.1 (Cont.)

TABLE 3.6-21 COUNTER EVENTS (Cont.)

*Codes 16		EVENTS
Counter A1/B1	Counter A2/B2	
12		Number of times a particular function code or a particular category of function codes is executed. The count condition is determined by an 8-bit select code and an 8-bit mask sent to the CPU on MCU channel BTA8. If the select code bits and the corresponding instruction function code bits are equal wherever there is "1" in the mask, the counter will be incremented. If the mask contains all zeros, all instructions will be counted.
	12	Time - 1 MHz.
13		Time between selecting microcode monitor field, MON=2 and selecting MON=3.
	13	Number of cycles where data is not available at the output of a functional unit once data has been requested for all input streams. This time does not include the time required for initial setup (preceding the input of the last operands to a functional unit). This count thus permits the programmer to analyze the amount of time required for startup memory accesses, pipeline/functional unit length, and memory conflicts for a specific instruction.

*These are 5-bit codes, expressed in hexadecimal.

3.6.4.1.1 MCU Count Gates and CPU Lines

The counters are incremented when the selected event occurs, the count line is up, and one or more of the following gate-line conditions is satisfied:

1. The Event Counter Enable bit is set in the invisible package of the job currently being executed and the Selected Job Gate from the MCU is set. This allows counts to be made during selected jobs only.

2. The CPU is in job mode and the Job Mode Gate from the MCU is set.

3. The CPU is in monitor mode and the Monitor Mode Gate from the MCU is set.

4. Data flag bit 56 (or 57) is set in the Data Flag Register of the CPU and the data flag 56 (or 57) gate from the MCU is set and the CPU is in monitor mode.

5. Data flag bit 56 (or 57) is set in the Data Flag Register of the CPU and the data flag 56 (or 57) gate from the MCU is set and the Event Counter Enable bit is set in the invisible package of the job currently being executed.

There is one set of gate-line enable logic for Counters A1 and A2 and one set for Counters B1 and B2; therefore, Counter A may be enabled by different gates than Counter B.

In summary the CPU lines are:

1. Data flag bit 56.

2. Data flag bit 57.

3. Monitor mode.

4. Job mode.

5. Job enable of monitoring counters from invisible package.

There is a corresponding MCU gate for each of the above.

3.6.4.1.2 Carry Lines

There is one enable carry line associated with each 16-bit counter. Enable carry line A1 enables the carry into Counter A1 from Counter A2. Enable carry line A2 enables the carry into Counter A2 from A1. There are equivalent lines for the B Counter. A zero on carry lines A1 and A2 allows the Counters to operate as two 16-bit counters. Only half of the total number of events are available at the selection network for one Counter A1 or A2; therefore, if a 32-bit count is desired, either counter may have the lower bits of count. For example, if an event is enabled to Counter A1 and a 32-bit count is desired, then enable carry line A1 must equal "0" and enable carry line A2 must be a "1". In this example, Counter A1 will have the least significant bits and Counter A2 will have the most significant.

3.6.4.1.3 Stop Lines

There is one stop line associated with each counter pair, one for the A Counters and one for the B Counters. When the stop line is a "1", an event incrementing either 16-bit counter will stop the comouter. Mode line "Event Stop" is returned to the MCU (bit 4, channel ATB8) to show why the CPU has stooped. The MCU, after sending a "Clear Fault Signal", may restart the CPU.

3.6.4.1.4 Counter Setup

Typically, the four counters would be set up by the MCU as follows:

1. Set the following bits as required:

   a. Stop CPU on A Increment (bit 9, channel BTA6)
   b. Stop CPU on B Increment (bit A, channel BTA6)
   c. Enable carry into A1 (bit B, channel BTA6)
   d. Enable carry into A2 (bit C, channel BTA6)
   e. Enable carry into B1 (bit D, channel BTA6)
   f. Enable carry into B2 (bit E, channel BTA6)

2. With bit F, channel BTA6, a zero, set event and mask selection for Counter A into channel BTA7.

3. Set bit F, channel BTA6 to a "1".

4. Set event and mask selection for Counter B into channel BTA7.

(continued)

3.6.4.1.4 (Cont.)

5. If A1/B1 event code 12 for function counting has been selected, set channel BTA8 to the desired function and mask.

6. Set count line A or B (bit 6 or 7, channel BTA6) as desired.

The counters will now be counting events and will continue to count until their respective count lines are drooped.

3.6.4.2 Display Registers

There are two 64-bit display registers that can be monitored by the MCU. One display register is used for the Current Instruction Address Register (CIAR) and the other is used for a register that has been selected by the MCU. The register is selected by a 4-bit code transmitted on bits C-F of channel BTA1. Any unlisted bits (such as bits 0-16 for code 3) are undefined.

The MCU must send a read signal to enable the CIAR and the selected register into the display registers. The read signal has been defined as bit 8 on channel BTA1 and its leading edge simultaneously transfers both registers into the display registers. The register select code must be set up by the MCU before the read signal is transmitted to the CPU.

The CIAR is received on channels ATB1 - ATB3 of the MCU and may read while the CPU is running. The selected register is received on channels ATB4 - ATB7 of the MCU. See Section 3.6.1.1 for bit assignments. The selected register on channels ATB4 - ATB7 may only be read when the CPU is stopped.

(continued)

3.6.4.2 (Continued)

The select codes and corresponding registers are listed below:

TABLE 3.6-22 DISPLAY REGISTER SELECT CODES

Code 16	Register(s)	Bits
0	Current Instruction Register	0-63
1	Data Flag Register	3-15 19-31 35-47 31-58
2	Invisible Package Address (Absolute Sword Address)	0-22
3	External Interrupt Register	15-31
	Monitor Interval Timer	15
	Channel 0	16
	1	17
	1	17
	2	18
	3	19
	4	20
	5	21
	6	22
	7	23
	8	24
	9	25
	10	26
	11	27
	12	28
	13	29
	14	30
	15	31
	Channel Read Active - Write Active	32-63
	Channel 0	32-33
	1	34-35
	2	36-37
	3	38-39
	4	40-41
	5	42-43
	6	44-45
	7	46-47
	8	48-49
	9	50-51
	10	52-53
	11	54-55
	12	56-57
	13	58-59
	14	60-61
	15	62-63

(continued)

3.6.4.2 (Cont.)

TABLE 3.6-22 DISPLAY REGISTER SELECT CODES (Cont.)

Code 16	Register(s)	Bits
4	SECDED Fault Read Bus Code	0-2
	I/O Bus = Code 0	
	R1 Bus = Code 1	
	R2 Bus = Code 2	
	R3 Bus = Code 3	
	Scalar Bus = Code 4	
	RNS Bus = Code 5	
	Instruction Stack Parity Fault	4
	MIC Memory 0 Parity Fault	5
	MIC Memory 1 Parity Fault	6
	Scalar MIC Parity Fault	7
	Double Secded Error. Syndrome Bits must be checked to determine if address and Bus Code are valid.	8
	Syndrome Bits	9-15
	Parity Fault on Auxiliary Board 0	16
	Parity Fault on Auxiliary Board 1	17
	Parity Fault on Auxiliary Board 2	18
	Parity Fault on Auxiliary Board 3	19
	Parity Fault on Auxiliary Board 4	20
	Parity Fault on Auxiliary Board 5	21
	Parity Fault on Auxiliary Board 6	22
	Parity Fault on Auxiliary Board 7	23
	Parity Fault on Auxiliary Board 8	24
	PM01 Enabled for Parity Checking	25
	Scalar Microcode Address -Bit 0	26
	Scalar Microcode Address -Bit 1	27
	Scalar Microcode Address -Bit 2	28
	Scalar Microcode Address -Bit 3	29
	Scalar Microcode Address -Bit 4	30
	Scalar Microcode Address -Bit 5	31
	Scalar Microcode Address -Bit 6	32
	Scalar Microcode Address -Bit 7	33
	NOTE: All Fault/Error conditions are cleared by the "Clear Fault" signal from the MCU except the SECDED Error and the Syndrome bits. These are cleared/ released by the "Clear Single Error" signal from the MCU.	

(continued)

3.6.4.2 (Cont.)

TABLE 3.6-22 DISPLAY REGISTER SELECT CODES (Cont.)

Code 16	Register(s)	Bits
4	(continued)	
	SECDED Fault Address (Absolute physical bit address, significant to the half-word level) The address of the first SECDED error is retained in this register. The SECDED Fault Address is released by the Clear Single Error Condition Signal from the MCU.	34-63
5	Bounds Hit Address (Absolute physical bit address, right justified) The address of the first bounds hit is retained in this register. The bounds hit address is released by the Clear Fault Condition Signal from the MCU. The bounds checking is performed on half-word boundaries only.	0-31
6	Counter A1 Counter A2 Counter B1 Counter B2 If bit 8 of channel BTA6 in the MCU is a "0", both counters will be cleared after the read signal is received and after both counters are transferred into the display register. If bit 8 is a "1", the counters will not be cleared. To ensure proper initialization of the counters, the count lines must be made zero prior to the new count selection.	0-15 16-31 32-47 48-63

3.6.4.3 Logic Fault Monitoring

There are two types of logic faults detected in the computer. They are memory SECDED and MIC memory parity, When a logic fault is detected, the computer stops between instructions. The types of fault may be sensed on channel AT88. (See Section 3.6.1).

After sensing the logic fault, the MCU must clear the fault via bit 7 of channel BTA1. The MCU must determine the appropriate response to the fault and has the option of restarting the CPU by setting bit 3 of channel BTA1.

3.7 Swap Unit

Figure 3.7-1 gives an overall block diagram of the Swap Unit. This unit performs swapping operations for data from the Register File to Main Memory, from Main Memory to the Register File during exchange operations and register file swap operations (7D instruction). The Swap Unit also performs block swap bperations between Main Memory and the Backing Store. Finally, the Swap Unit provides the I/O interface to the Backing Store.

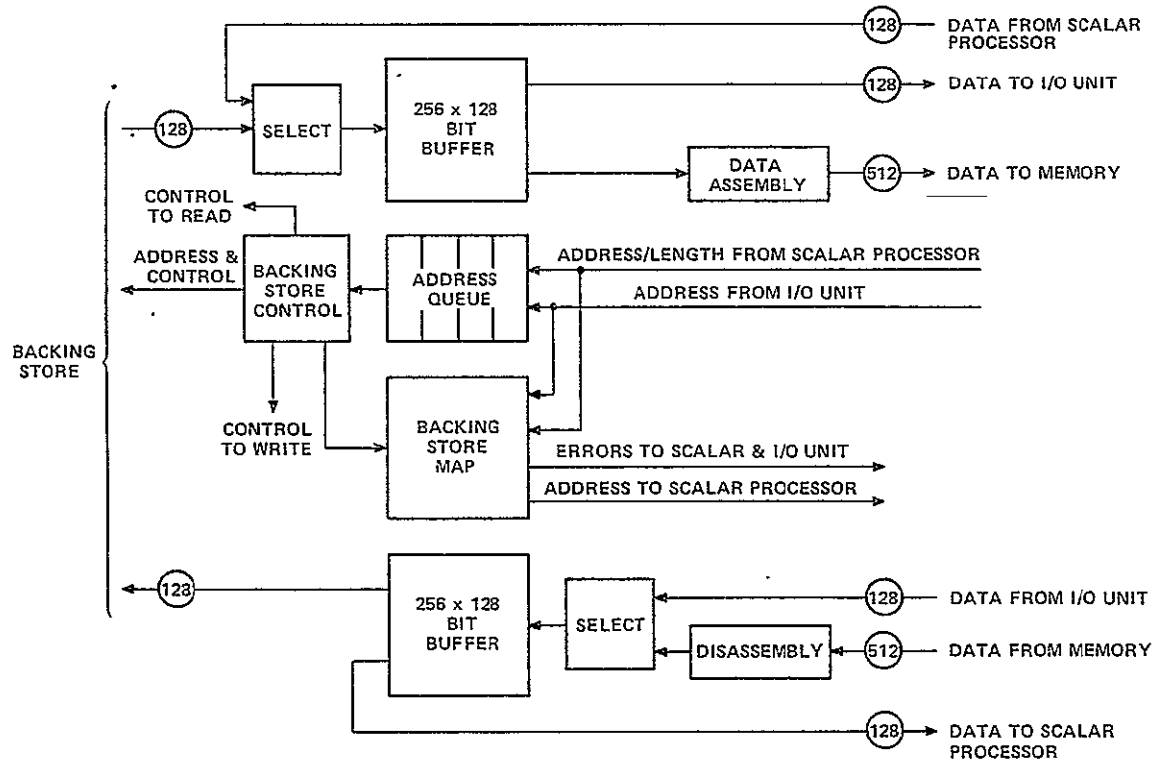


Figure 3.7-1 FMP Swap Unit

3.7.1 Data Movement

The Swap Unit moves swap data between register files and memories at the rate of 128 bits per minor cycle. Data transmitted to/from Main Memory is transferred in 512-bit (plus SECDED) segments. Thus a set of assembly and disassembly buffers are provided to agglutinate or decompose data into/from 512-bit units from/to the transfer quantity of 128 bits.

3.7.2 Error Checking

SECDED is carried on all trunks (including the register file swap trunk) and in the data buffers. SECDED originating in the Main Memory is composed of seven bits for every 32 data bits. SECDED is carried this way (seven bits for every 32 bits) throughout the FMP with the exception of the Backing Store. SECDED within the Backing Store consists of 11 error code bits for each 512 data bits; the conversion between seven bits for every 32 and 11 bits for every 512 must be accomplished at the Backing Store interface.

In the event that a single-bit error is detected by the Swap Unit, a flag is sent to the MCU and the counters pointing to the data plus syndrome bits are locked up for sampling by the MCU. Note that in such cases the first error only will be locked up in the MCU interface.

When a double-bit error is detected, the Swap Unit is halted immediately, stop flags are sent to the Scalar, Map and Vector Units, and the MCU is alerted. Location and nature of the error are locked up as in the case of the single-bit error.

3.7.3 Addressing

All transfers involving the Backing Store occur in blocks of 32,768 64-bit words. All exchanges between the Register File and Main Memory occur in multiples of 128-bit quarter-swords, beginning on a quarter-sword boundary. When a reference to a particular Backing Store block is made, data transmission from that block begins as soon as the particular block is selected, rather than waiting for the first word of the block to appear at the output or input ports of the Backing Store. This is necessary to reduce the latency time inherent in block-organized CCD memory systems. With a block length of 32,768 words the latency time to wait for the first word can be as long as 3.2 milliseconds. This can be reduced to near zero by beginning transfers immediately.

The ability to begin Backing Store transfers immediately requires address control information to be exchanged between the Backing Store and the Swap Unit. In particular, the current address being read by the Backing Store must be sent to the Swap Unit so that Main Memory write addresses can be adjusted to match the starting word location. The same is true for the Read from Memory, Write to Backing Store.

3.7.4 Address Queue and Backing Store Map

The Swap Unit is capable of four different swap instructions, one in process and three waiting to be executed. This permits the object code in the Scalar Processor to release a series of swap instructions local to a segment of computation and go on to something else. As addresses and swap control information arrive at the Swap Unit, the corresponding blocks in the Backing Store are set busy in the backing store map. This prevents accidental overlap of swap operations. A similar busy is set for Main Memory by the Memory Interchange to prevent erroneous conflict situations to arise.

(continued)

3.7.4 (Cont.)

I/O operations are funneled through the Swap Unit, by checking the busy map. I/O operations are permitted to occur only with busy blocks (which have been set busy by the monitor). At the completion of an I/O operation, the I/O processor can clear the busy bit or instruct the monitor to clear it.

In the event that two swap requests are made to the same block of Backing Store, such that the second request encounters a busy block, the request will be rejected and a data flag bit set to indicate that an error condition has arisen. The programmer may choose to sample the data flag bit or to enable an automatic data flag branch to an error handling routine.

3.7.5 Control Signals

(To be defined later)

3.7.6 Microcode Control Terms

(To be defined later)

3.7.7 Interface Signals

(To be defined later)

3.8 Backing Store

The Backing Storage subsystem provides a massive, on-line memory with fast access and high block transfer bandwidths. The memory is engineered into 32-million-word modules, so the minimum configuration is one 32-million-word cabinet. The maximum configuration is governed by available space, but with the 65K chips now available in CCD (charge coupled devices) the practical maximum appears to be 256-million words. The allowable address space permits an expansion up to 1-billion words of Backing Storage as technology developments make such volumes feasible.

3.8.1 Data Movement

Data is read/stored from/into the CCD memory on 512-bit paths. Every 400 nanoseconds 512 bits are read in parallel from a selected rank of memory chips. The data is latched into 512-bit holding registers and disassembled into 128-bit segments for transmission to the Swap Unit every tenth minor cycle. Writing of the Backing Store proceeds in reverse order with 128-bit segments being transmitted and assembled every tenth minor cycle, and the resulting 512 bits being written to the Backing Store every 400 nanoseconds.

3.8.2 Error Checking

The Swap Unit transmits and receives 128 bits, plus 7 bits of SECDED for each 32 data bits, between the Backing Store and itself while 11 bits of SECDED are stored and retrieved with each 512 bits of data in the Backing Store. The conversion from 7 SECDED bits per 32 data bits to 11 SECDED bits per 512 data bits (and vice versa) is done in the Backing Store Unit at the assembly/disassembly interface. While the new SECDED code is generated, previous SECDED is checked with appropriate error correction and/or flagging taking place.

3.8.3 Addressing

A read or write operation is initiated by the Swap Unit sending a basic block address and read or write signal to the Backing Store. In return the Backing Store Unit transmits the next word address available for readng and writing. The Swap Unit then adjusts its memory addresses so that data transfer can begin immediately.

3.8.4 Control Signals

(To be defined later)

3.8.5 Interface Signals

(To be defined later)

3.9 Timing Information

The FMP is in preliminary design phase so only the most preliminary timing estimates are available. All estimates are given in CPU minor clock cycles. The period of this clock cycle is predicted to be 10 nanoseconds, but with extant technology can be no worse then 16 nanoseconds.

3.9.1 Scalar Processor Timing

The table in Section 3.9.1.2 is designed to provide scalar timing data for the instruction sequences in FMP. All timing data is expressed in minor cycles.

3.9.1.1

Multi-operand instructions are typically expressed as overhead + (number of cycles per operand) (number of operands).

Scalar instructions are expressed as described below.

The ISSUE portion of the table gives the minimum number of minor cycles between the issue if the specific instruction listed in the left column and the issue of the next instruction in the sequence. Various operand or memory conflicts (as discussed later) can cause additional delay beyond this minimum time.

The Issue portion of the tables is sub-divided when appropriate into three categories as defined below:

NB -- No Branch
ISB -- In Stack Branch
OSB -- Out of Stack Branch to first quarter-sword. This time must be increased by 1, 2 or 3 if the Branch address is in the 2nd, 3rd or 4th quarter-sword respectively.

The non-branch instructions use the entry under NB or No Branch. Example 1 illustrates a simple, no conflict, Branch sequence.

(continued)

3.9.1.1 (Cont.)

Example 1

	Instr.	R S T	Comments
A)	60	- - -	Register designators which are not pertinent to the example are represented by a -.
B)	25	- - -	Branch condition not met.
C)	65	- - -	
D)	25	- - -	Branch condition met, branch out of stack to instruction E in 3rd quarter-sword.

Sequence Timing

Instr. A issues at minor cycle 0
Instr. B issues at minor cycle 1
Instr. C issues at minor cycle 12
Instr. D issues at minor cycle 13
Instr. E issues at minor cycle 42

The RESULT AVAILABLE portion of the table contains information necessary to time instruction sequences with operand dependencies. The first column, SS or shortstop, contains entries for those instructions which use the Scalar Floating-Point Unit. These are the instructions which may use the shortstop feature to provide an input operand. This entry is the number of minor cycles after issue that the result operand will be available at the shortstop for use with a following instruction.

If instruction A issues at minor cycle X, any following instruction, B, needing the result of A must issue no later than minor cycle X+SS to utilize the shortstop. A floating-point instruction needing the result of A, can be issued before X+SS and wait at the input of Floating Point for the shortstopped result of A. This allows other non-floating-point

(continued)

3.9.1.1 (Cont.)

instructions to issue. The resulting time of an instruction that issues and waits for shortstop will be as if it had issued at the ideal time to match shortstop. A subsequent instruction requiring access to Floating-Point will not issue any earlier than (X+SS) + 1.

If instruction B issues later than cycle X+SS, thus missing the shortstop; instruction B must wait until at least X+RF. At this time the desired operand will be available from the Register File. Examole 2 illustrates operations using shortstop.

Example 2

	Instr.	R	S	T	Comments
A)	60	-	-	12	
B)	60	-	-	-	
C)	60	-	-	-	
D)	60	-	-	-	
E)	60	-	-	-	
F)	60	12	-	14	
G)	60	14	14	-	
H)	7F	-	-	-	
I)	60	-	-	15	
J)	60	14	-	16	
K)	60	16	15	-	
L)	60	-	-	-	

Sequence Timing

Instr. A issues at 0
Instr. B issues at 1
Instr. C issues at 2
Instr. D issues at 3
Instr. E issues at 4
Instr. F issues at 5 -- Thus exactly matching shortstoo
Instr. G issues at 6 -- Issues but must wait at the inout of Floating Point for the result of instruction F to be available

(continued)

3.9.1.1 (Cont.)

Instr. H issues at 7		
Instr. I issues at 11	--	Cannot issue until instruction G catches shortstop and proceeds. Thus 3 minor cycles not used
Instr. J issues at 13	--	Missed result of instruction F at shortstop thus waiting until operand is available from Register File
Instr. K issues at 14	--	Issues and waits at input to Floating Point
Instr. L issues at 19	--	Instruction K is treated as if issued at 18 and the L at 19

The last column under RESULT AVAILABLE (MEM) contains entries for those scalar instructions (13, 32, 5F, 7F) which store a result into Main Memory. The time listed is the minimum time until the operand is in memory and available for use. The time may also be increased by 4 minor cycles if the desired memory bank is busy.

The UNIT BUSY portion of the table concerns instructions issued to either the Divide/Convert (D/C) Unit or the Load/Store Unit (L/S). The Divide/Convert Unit executes the 10, 11, 4C, 4F, 53, 6C, 6F and 73 instructions. This unit is the only portion of Scalar Floating-Point which is not completely pipelined; thus the appropriate unit busy time listed in the table must elapse before a third instruction can be issued to the Divide/Convert Unit. Floating-Point instructions other than these eight may be issued to Floating-Point while the Divide/Convert Unit is busy. A second instruction from the set of eight can be issued, but will be held in front of the Floating-Point Unit and issuing of non-floating-point instructions will continue.

(continued)

3.9.1.1 (Cont.)

The Load/Store Unit executes the 12, 13, 32, 5E, 5F, 7E and 7F instructions. There are six address registers in the Load/Store Unit which enable requests to be stacked and executed in the proper order. The 12, 5E and 7E instructions each require one register and can be executed (in the absence of memory conflicts) at the rate of one load per minor cycle. The 5F and 7F instructions each require two address registers and can be executed at one store per two minor cycles. The 13 and 32 instructions each require two address registers and can be executed at one per 14 and 15 minor cycles, respectively.

The Load/Store Unit is then capable of streaming Load/Store instructions (other than the 13 and 32) at one minor cycle per load and two minor cycles per store assuming no Memory or Register File conflicts. For example, a stream of N loads will execute in N + 14 minor cycles from the issue of the first load until the operand from the last load is available in the Register File. A stream of N stores will execute in $2N + \lfloor \frac{N}{13} \rfloor$ minor cycles from issue of the first store until issue of the last store.

Example 3

	Instr.	R	S	T	Comments
A)	60	-	-	-	
B)	7E	-	-	-	
C)	13	-	-	-	
D)	13	-	-	-	
E)	60	-	-	-	
F)	7E	-	-	-	
G)	7E	-	-	-	
H)	7E	-	-	-	
I)	13	-	-	-	

(continued)

3.9.1.1 (Cont.)

Sequence Timing

Instr. A issues at 0
Instr. B issues at 1
Instr. C issues at 2
Instr. D issues at 4
Instr. E issues at 6
Instr. F issues at 7
Instr. G issues at 8
Instr. H issues at 19 Instr. H must wait for address register to become free from Instr. C.

Instr. I issues at 35 Instr. I must wait for address register.

There are three additional Operand Dependencies which must be considered.

1. Source operand conflict -- an instruction requiring the result of a previous instruction as an inout operand waits until the operand becomes available.

2. Output operand conflict -- an instruction output to the same Register File location as a previously issued, but slower instruction, waits until the previous instruction stores its result in the Register File.

3. Register File Write conflict -- an instruction cannot issue if its result arrives at the Register File at the same minor cycle as the result of a previously issued but slower instruction.

Table 3.9-1 pertains to instructions having greater than 1 minor cycle issue time.

The first column lists the appropriate instructions. The second column indicates the minor cycle of issue that a specific operand is required. The third column indicates the availability of shortstop for that specific operand.

(continued)

TABLE 3.9-1. ORDER THAT DESIGNATORS ARE READ FOR MULTIPLE ISSUE INSTRUCTION AND IF THEY CAN CATCH SHORTSTOP OF A PREVIOUS INSTRUCTION

INSTRUCTION		DESCRIPTION	SHORTSTOP
13	1st	R&S&T	No
	2nd	T	No
20-27	1st	T	No
	2nd	R&S	Yes
2F	1st	S	No
	2nd	T	No
	3rd	T	No
31&35	1st	S&T	No
	2nd	R	No
	3rd	R	No
32	1st	S	No
	2nd	T	No
36	1st	S&T	No
	2nd	R	No
	3rd	R	No
5F	1st	R&S	No
	2nd	T	No
6D	1st	R&S	Yes (R&S)
	2nd	T	No
7F	1st	R&S&T	No
	2nd	T	No
B0-B5.X00X-X	1st	B&Y	No
	2nd	X&A	No
	3rd	Z	No
	4th	X&A&C	No
B0-B5.X01X-X	1st	B&Y	No
	2nd	X&A&C	Yes (X+A)
	3rd	Z	Yes
B0-B5.X10X-X	1st	B&Y	No
	2nd	X&A	Yes
B0-B5.X11X-X	1st	B&Y	No
	2nd	X&A	Yes
B6	1st	R	No

3.9.1.1 (Cont.)

Examole 4

	Instr.	R	S	I	Comments
A)	60	-	-	12	
B)	36	10	-	12	Specifies an out of stack branch to Instruction C in the 2nd quarter sword
C)	60	-	-	35	
D)	80	40	35	-	Specifies an in stack branch to Instruction E
	--	-	-	-	
E)	60	-	-	-	

Sequence Timing

Instr. A issues at 0		
Instr. B issues at 8		B must wait for Result from A to be stored into the Register File
Instr. C issues at 32		
Instr. D issues at 36		Result from Instruction C available from Shortstop at time 37 allows issue at 36
Instr. E issues at 48		

3.9.1.2 Basic Instruction Timing

TABLE 3.9-2 SCALAR INSTRUCTION TIMES

Instructions	Issue NB	Issue ISB	Issue OSB	Result Avail. S.S.	Result Avail. R.F.	Result Avail. MEM	Unit Busy L/S	Unit Busy D/C
00	Waits for external or real-time interrupt							
04	20							
06	4							
08	20							
09	Job to Monitor							
		--			--	--		
		--			--	--		
	Monitor to Job							
0A	20							
0E		20	34					
10	1	--	--	22	25			9
11	1	--	--	53	56			25
12	1	--	--	--	15		1*	
13	2	--	--	--	--	23	14*	
20	1	--	--	--	--	--		

* MUST ADD 5 MC FOR REGISTER RELEASE

(continued)

R A D L

3.9.1.2 (Cont.)

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

	Issue			Result Avail.			Unit Busy	
Instructions	NB	ISB	OSB	S.S.	R.F.	MEM	L/S	D/C
21	1	--	--	3	6	--		
28	1	--	--	3	6	--		
2C	1	--	--	3	6	--		
2D	1	--	--	3	6	--		
2E	1	--	--	3	6	--		
2F	7	8	23.	---	7	--		
30	1	--	--	3	6	--		
31	7	8	23	--	7	--		

* MUST ADD 5 MC FOR REGISTER RELEASE

(continued)

3.9.1.2 (Cont.)

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

Instructions	Issue NB	Issue ISB	Issue OSB	Result Avail. S.S.	Result Avail. R.F.	Result Avail. MEM	Unit Busy L/S	Unit Busy D/C
							G-bits	2&3
32.00X-X	2	--	--	--	--	(24 (	15*))	0 1
32.01-X		9	24	--	--	<(24 (	15*))	>1 0
32.1X-X	20	21	36	--	--	(24 (	15*))	1 1
33.XXXXX0XX								
33.XXXXX1XX								
34	1	--	--	3	6	--		
35	7	8	23	--	7	--		
36,R=T,S=0	5	--	--	--	5	--		
36,R=T,S≠0		9	24	--				
36,R≠T		8	23	--	5	--		
37	32			--	--			
38	1	--	--	1	4	--		
39	30			--	--			
3A	20			--	--			
3B	26	--	--		--	--		
3C	1	--	--	5	8	--		
3D	1	--	--	5	8	--		

* MUST ADD 5 MC FOR REGISTER RELEASE

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

	Issue			Result Avail.			Unit Busy	
Instructions	NB	ISB	OSB	S.S.	R.F.	MEM	L/S	D/C
3E	1	--	--	1	4	--		
3F	1	--	--	1	4	--		
40	1	--	--	5	8	--		
41	1	--	--	5	8	--		
42	1	--	--	5	8	--		
44	1	--	--	5	8	--		
45	1	--	--	5	8	--		
46	1	--	--	5	8	--		
48	1	--	--	5	8	--		
49	1	--	--	5	8	--		
4B	1	--	--	5	8	--		
4C	1	--	--	30	33	--		13
4D	1	--	--	1	4	--		
4E	1	--	--	1	4	--		
4F	1	--	--	30	33	--		13
50	1	--	--	5	8	--		
51	1	--	--	5	8	--		
52	1	--	--	5	8	--		
53	1	--	--	29	32	--		13

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

	Issue			Result Avail.			Unit Busy	
Instructions	NB	ISB	OSB	S.S.	R.F.	MEM	L/S	D/C
54	1	--	--	5	8	--		
55	1	--	--	5	8	--		
58	1	--	--	1	4	--		
59	1	--	--	5	8	--		
5A	1	--	--	3	6	--		
5B	1	--	--	3	6	--		
5C	1	--	--	5	8	--		
5D	1	--	--	5	8	--		
5E	1	--	--	--	15	--	1*	
5F	2	--	--	--	--	10	2*	
60	1	--	--	5	8	--		
61	1	--	--	5	8	--		
62	1	--	--	5	8	--		
63	1	--	--	1	4	--		
64	1	--	--	5	8	--		
65	1	--	--	5	8	--		
66	1	--	--	5	8	--		
67	1	--	--	1	4	--		

* MUST ADD 5 MC FOR REGULAR RELEASE

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

	Issue			Result Avail.			Unit Busy	
Instructions	NB	ISB	OSB	S.S.	R.F.	MEM	L/S	D/C
68	1	--	--	5	8	--		
69	1	--	--	5	8	--		
6B	1	--	--	5	8	--		
6C	1	--	--	54	57	--		25
6D	2	--	--	4	7	--		
6E	1	--	--	3	6	--		
6F	1	--	--	54	57	--		25
70	1	--	--	5	8	--		
71	1	--	--	5	8	--		
72	1	--	--	5	8	--		
73	1	--	--	53	56	--		25
74	1	--	--	5	8	--		
75	1	--	--	5	8	--		
76	1	--	--	5	8	--		
77	1	--	--	5	8	--		
78	1	--	--	1	4	--		
79	1	--	--	5	8	--		
7A	1	--	--	3	6	--		

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

Instructions	Issue NB	Issue ISB	Issue OSB	Result Avail. S.S.	Result Avail. R.F.	Result Avail. MEM	Unit Busy L/S	Unit Busy D/C
7B	1	--	--	3	6	--		
7C	1	--	--	3	6	--		
7E	1	--	--	--	15	--	1*	
7F	2	--	--	--	---	10	2*	
B0.X00 X-X	8	9	24	--	8	--		
B0.X01 X-X	3	--	--	5	5+8**	--		
B0.X10 X-X	11	12	27	--	--	--		
B0.X11 X-X	2	--	--	6	9	--		
B1.X00 X-X	8	9	24	--	8	--		
B1.X01 X-X	3	--	--	5	5+8**	--		
B1.X00 X-X	11	12	27	--	--	--		
B1.X11 X-X	2	--	--	6	9	--		
B2.X00 X-X	8	9	24	--	8	--		
B2.X01 X-X	3	--	--	5	5+8**	--		
B2.X10 X-X	11	12	27	--	--	--		
B2.X11 X-X	2	--	--	6	9	--		

* MUST ADD 5 MC FOR REGISTER RELEASE.

**Output to be stored in Register C is available at 5 cycles and Y at 8 cycles. Y may be used from the Shortstop at time 5. C can not be shortstopped.

TABLE 3.9-2 SCALAR INSTRUCTION TIMES (Cont.)

Instructions	Issue NB	Issue ISB	Issue OSB	Result Avail. S.S.	Result Avail. R.F.	Result Avail. MEM	Unit Busy L/S	Unit Busy D/C
B3.X00 X-X	8	9	24	--	8	--		
B3.X01 X-X	3	--	--	5	5+8**	--		
B3.X10 X-X	11	12	27	--	--	--		
B3.X11 X-X	2	--	--	6	9	--		
B4.X00 X-X	8	9	24	--	8	--		
B4.X01 X-X	3	--	--	5	5+8**	--		
B4.X10 X-X	11	12	27	--	--	--		
B4.X11 X-X	2	--	--	6	9	--		
B5.X00 X-X	8	9	24	--	8	--		
B5.X01 X-X	3	--	--	5	5+8**	--		
B5.X10 X-X	11	12	27	--	--	--		
B5.X11 X-X	2	--	--	6	9	--		
B6	7	8	23	--	--			
BE	1	--	--	1	4			
BF	1	--	--	1	4			
CD	1	--	--	1	4			
CE	1	--	--	1	4			

**Output to be stored in Register C is available at 5 cycles and Y at 8 cycles. Y may be used from the Shortstop at time 5. C can not be shortstopped.

3.9.2 Vector Processor Timing

All vector processing times are stated in terms of overhead (the time required to start up a vector operation) and vector throughput in results per minor cycle. Total time for a vector operation is then stated as O+N/R where O is the overhead, R is the rate of results per minor cycle, and N=M times the ceiling of L/M; L is the vector length and M is 8 for 64-bit operands or 16 for 32-bit operands. Ceiling is the APL operator which returns the maximum integer value of the argument.

Vector overhead is variable depending on a number of conditions. Its component parts are:

1. Issue time---The Instruction Issue Unit requires a certain number of cycles to translate and scan over the vector instructions and the included 32-bit packets.

2. Transmission of control information from the Scalar Unit.

3. Map Unit setup----The time required to form addresses and initiate the memory requests within the Map Unit.

4. Transmission of memory request.

5. Memory access time.

6. Data transmission to Map Unit.

7. Data transmission through Map Unit.

(continued)

3.9.2 (Cont.)

8. Data transmission to Vector Unit.

9. Time through vector pipelines.

10. Transmission to Map Unit.

11. Data path through Map Unit.

12. Transmission to memory.

When two vector operations appear in consecutive instructions in the Issue Unit the issue time, transmission of control to the Map Unit, and part of the buffer setup time are overlapped. Thus the overhead in such cases can be reduced.

Table 3.9-3 gives Vector Processor times for operations involving the Vector Units. See section 3.9.3 for timing of vector operations executed within the Map Unit and section 3.9.4 for Swap Unit timing.

TABLE 3.9-3 VECTOR PROCESSOR TIMES

Function	Modifier	Source	Destination	Overhead Time*	Sustained Data Rate/Cycle
Vector	Normal	Memory	Memory	28 + 2P	512 bits
		Memory	Buffer	20 + 2P	
		Buffer	Memory	20 + 2P	
	v	Buffer	Buffer	12 + 2P	
	Broadcast	Memory	Memory	28 + 2P	
		Memory	Buffer	20 + 2P	
		Buffer	Memory	20 + 2P	
v	v	Buffer	Buffer	12 + 2P	v
Buffer Load	None	Memory	Buffer	14 + 2P	512 bits

*P = Packet Count in the instruction header.

3.9.3 Map Unit Timing

The Map Unit functions of MERGE, MASK, COMPRESS, SCATTER and GATHER operations are incorporated physically within the single Map Unit. These operations have only memory as a source of operands but all except SCATTER can deliver results to either memory or the vector buffers. Table 3.9-4 gives timing information for these Map Unit functions.

TABLE 3.9-4 MAP UNIT TIMES

Function	Mode	Modifier	Destination	Overhead Time*	Maximum Data Rate/Cycle
GATHER	Word	None	Memory	36 + 2P	1 operand**
		None	Buffer	28 + 2P	
		Stride	Memory	26 + 2P	
		Stride	Buffer	22 + 2P	
	Record	None	Memory	36 + 2P	512 bits***
		None	Buffer	28 + 2P	
		Stride	Memory	26 + 2P	
		Stride	Buffer	22 + 2P	
SCATTER	Word	None	Memory	30 + 2P	1 operand**
	Word	Stride	Memory	28 + 2P	
	Record	None	Memory	30 + 2P	512 bits***
	Record	Stride	Memory	28 + 2P	
COMPRESS	N/A	N/A	Memory	26 + 2P	8 input operands**
			Buffer	22 + 2P	8 input operands**
MASK			Memory	24 + 2P	512 bits
			Buffer	20 + 2P	
MERGE			Memory	26 + 2P	8 output operands**
			Buffer	22 + 2P	8 output operands**

*P = Packet count in the instruction header.
**Either 32-bit or 64-bit word.
***Rate assumes that swords are moved on sword boundaries; if not, maximum rate is 256 bits per cycle.

3.9.4 Swap Unit Timing

Swap Unit startup consists of the issue cycle = 1, transmission to the Swap Unit = 1, and Swap Unit setup = 5 cycles. Once begun, the swap operation moves data at the rate of 512 data bits every 400 nanoseconds.

4.0 QUALITY ASSURANCE PROVISIONS - Not Applicable

5.0 PREPARATION FOR DELIVERY - Not Applicable

6.0 NOTES

6.1 Intercom

CDC FMP has an intercom system which is utilized primarily for maintenance purposes. The system can be enabled by simply plugging the required number of headsets into the desired intercom jacks. There are intercom jacks located in each section and in the MCU. Up to four headsets may be on-line at any time.

6.2 System Start-up

The Start-up sequence for the system is as follows:

1. Bring up system power.
2. Autoload MCU.
3. Master clear the system from the CPU.

   This master clear:

   a. Initializes the CPU - clears all control flip-flops, data flags, interrupts and error flip-flops.
   b. Sets monitor mode in the CPU (Job Mode FF cleared in step A).

4. Load microcode into the Stream Unit and the Scalar Unit from the MCU.

5. The MCU sends an external flag to the I/O stations required on-line. The stations, on receiving this flag, will autoload and enter an idle loop waiting for a channel flag from the CPU. An alternative approach is to manually autoload each of the stations desired on-line.

6. The MCU loads the operating system kernel into Main Memory, then interrupts the CPU. The CPU recognizes the interrupt and executes a partial exchange to start execution in monitor mode. This exchange is the same as a normal job to monitor exchange except the contents of the Register File are not stored. Program execution starts at the address contained in monitor's register six just as it does after a normal I/O interrupt.

APPENDIX C

PROGRAMMED DEVICE

CONTROLLER DESCRIPTION

Appendix C

PROGRAMMED DEVICE CONTROLLER DESCRIPTION

A Programmed Device Controller (PDC) is a unit which adapts data channels or peripheral controllers to a serial data trunk. By means of the PDC and the serial trunk, a set of processors, or processors and peripherals, may be conveniently and efficiently interconnected.

Figure C-1 illustrates a processor to processor data link construct using the serial trunk for connectivity and PDC's for adapting the CPU channels to the trunk. A CPU can present messages (and data) to the PDC, and can accept messages (and data) from the PDC. The PDC is an agent for inserting messages on the trunk and for selecting messages from the trunk. In this case the PDC is neither the originator of an activity nor the recipient, but rather the means for conveying the bit stream defining the activity.

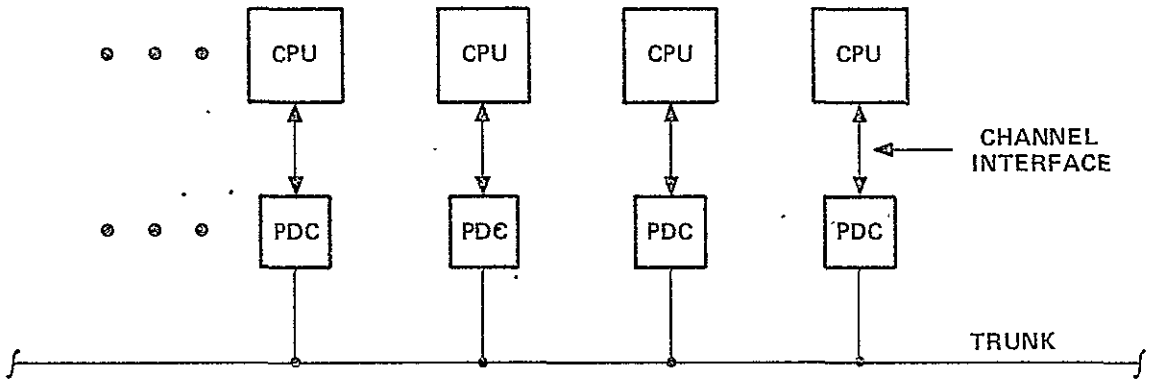


Figure C-1. Processor To Processor Data Link

Figure C-2 illustrates an interconnect of host processors and peripheral units. The function performed by the PDC for the host processors here is the same as that in the processor-to-processor data link: to act as the agent for message (and data) delivery, being neither the originator nor the recipient of a message. The PDC adapting a peripheral unit to a trunk, on the other hand, is itself the originator or recipient of messages. As such, it has the function of message interpretation and execution as well as the message delivery function.

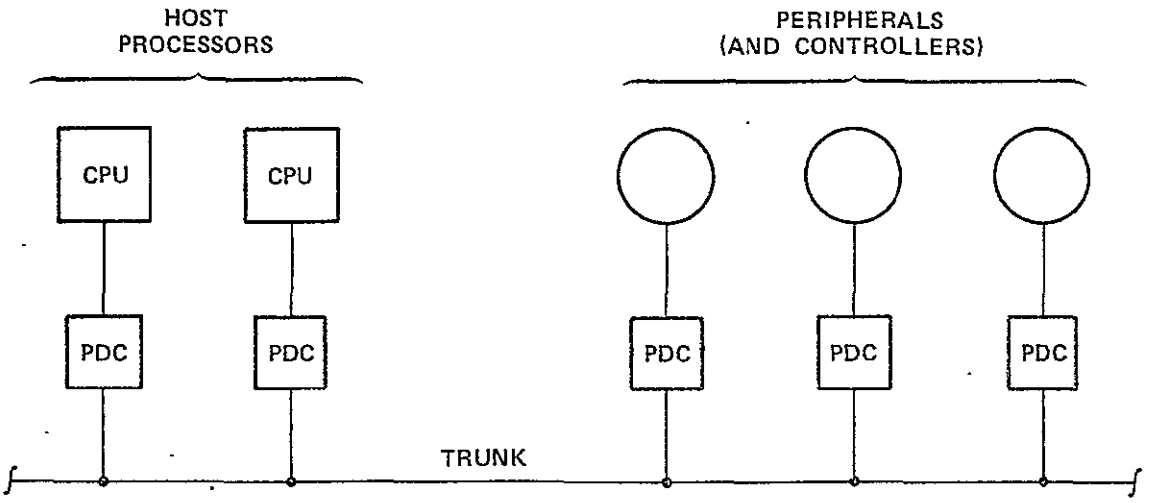
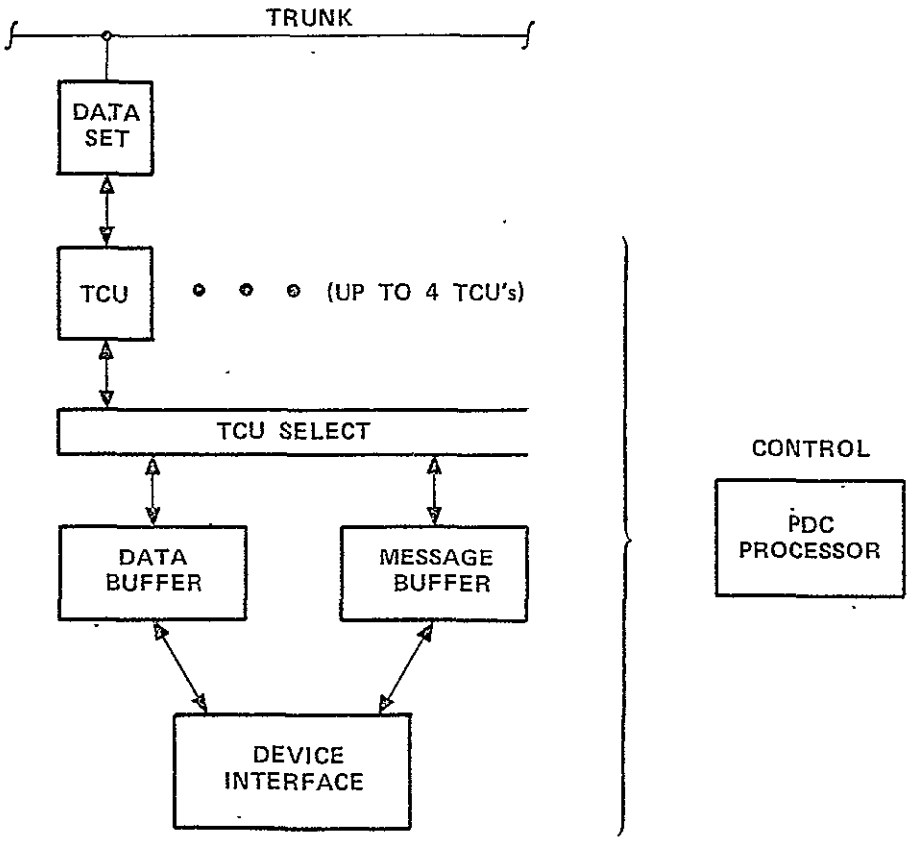


Figure C-2. Processor/Peripheral – Subsystem Network

This type of PDC, having an additional function to satisfy, requires resources in addition to those of the processor-adapting PDC, namely execution time and memory; the design must be capable of such extension.



The PDC consists of transmission control units, message and data buffers, a device interface, and a processor to manage these resources in a way which satisfies the PDC functions.

1.0 TRANSMISSION CONTROL UNIT (TCU)

The TCU decouples the message transmission protocol from the functional definition of the message. The message transmission protocol includes,

- Trunk protocol – a bit-oriented protocol similar to SDLC
- Contention resolution
- Access control
- Message closure

and defines how a message is moved from a TCU buffer interface, down the trunk, to another TCU buffer interface, and how a message disposition status is returned.

2.0 BUFFERS

The PDC buffer decouples the data rate of the attached device (processor channel, peripheral interface) from the 50-megabit data rate of the trunk. The buffer has two parts, a message section and a data section.

Messages have a predefined format and are of fixed length. Data transfers, from the viewpoint of the attached unit, can be of any length. The PDC blocks and unblocks long data fields for transmission on the trunk. The data buffer functions as a circular buffer. When half full, the PDC begins transmitting the data block on the trunk, while at the same time the device continues outputting data to the PDC (other half of the data buffer). The receiving PDC performs a similar function, placing the received data blocks into its data buffer in a circular fashion. Thus the sending PDC waits for its attached device to deliver a block (half buffer) of data, "bursts" this data on the trunk, and waits for the next block. The trunk, is available to other communicating PDC's between bursts.

Messages, as they arrive off the trunk, are placed into the message buffer one after the other until either the message buffer becomes full, or a message arrives with associated data. If the message buffer is full, the TCU returns a BUSY response to the originating TCU. If the message is accepted by this PDC, an ACK is returned to the originating PDC. The PDC will accept one message with associated data at a time. No additional messages will be accepted until the process defined by the data message and its data has been concluded.

3.0 DEVICE INTERFACE

This is the logic necessary to match the device control and data characteristics to the PDC. A channel interface includes data assembly/disassembly, resync, ready/resume, and whatever control line and function translation is required. This logic also includes voltage/impedance (V/Z) matching. The interface to a peripheral controller is essentially the same as a channel interface.

The major difference between a PDC interfacing with a processor channel and one interfacing with a peripheral controller is the question of control. A PDC is passive to a channel, and active to a controller. Thus a channel PDC reacts to the commands of the channel; the state of this PDC is controlled by the channel. The situation is reversed for the peripheral controller PDC. Here the state of the peripheral controller is controlled by the PDC; the PDC, in effect, appears to the peripheral controller to be a channel.

4.0 PROCESSOR

This is the programmable element in the programmable device controller. The software executed by the processor includes a basic set, which is found in all PDC's, and an application-oriented set, i.e., the channel PDC, peripheral controller PDC (and its variations).

5.0 BASIC SOFTWARE SET

- Message transmission control
  1. Interprets response to message
     a. ACK – Message received. Transmission is completed. Set status for channel.
     b. BUSY – Message was not accepted. Transmission is completed. Set status for channel.
     c. No Answer – Contention, transmission error, access violation, or failed PDC (also nonexistent). Perform a retry operation.

     Results (a, b, or c) determine next action.
- Buffer management
  1. Message queue – FIFO. Manage queue pointers. Set up address registers for both the channel and the TCU interfaces.
  2. Data buffer management. Set up address and length registers.
- Data blocking
  1. Synchronize the transmission of data blocks on the trunk with data motion on the channel.
- PDC state control
  1. Busy/available (for messages), transmit/receive data, connected/disconnected, autoload.

6.0 CHANNEL PDC SOFTWARE SET

- Channel Interface
  1. Output message, output data – interpret selects.
  2. Input message, input data – interpret selects.
  3. END OF OP – synchronous with final data message.

4. STATUS

5. ABORT – send message to destination PDC similar to the end of operation message.

6. MASTER CLEAR

   a. Hardware – reset pointers.

   b. Software (function) – Abort data input/output if in progress.

7.0 PERIPHERAL CONTROLLER PDC SOFTWARE SET

- Channel extension

  1. Convert command words to channel functions.

  2. Input/output data.

  3. Data blocking/deblocking.

- Device driver

  1. Manage queue of read/write requests.

  2. Convert requests to the appropriate set of channel command words.

  3. Error retry/recovery.

  4. All of channel extensions.

- Higher level function

  1. File system

     a. Message translation

     b. Resource management

     c. Catalogue

     d. Access control

     e. All of channel extensions, all of device driver

APPENDIX D

SERIAL TRUNK

CONTROL PROCEDURE

Appendix D

SERIAL TRUNK CONTROL PROCEDURE

The Programmable Device Controller Communication-Control Procedure (PDCCP) is a bit-oriented, code-independent, modular data link (trunk) control protocol. It is designed for a conference multipoint interconnect, and is meant to be line and link compatible, to the greatest extent practical, with Control Data Communication Control Procedure (CDCCP). CDCCP is presently under development as a Control Data Corporate Standard and is not yet available for publication.

CDCCP, which is intended for use in "datacomm" networks, i.e., those using relatively low-speed common carrier facilities, expressly forbids multipoint interconnects, whereas the serial trunk system allows them. This constitutes the principal difference in the design philosophies of the two protocols, and arises because of the widely divergent application requirements.

This document defines in detail the frame structure used in all PDCCP transmissions. It describes the structure, formatting, and significance of the various fields in the frame as well as frame delimiting flags and frame check sequences.

1.0 FRAME STRUCTURE

1.1 GENERAL

The vehicle for all command, response, and information transmission is called a frame. A frame is a sequence of contiguous bits bounded by and including opening and closing flag sequences. There are two types of valid frames, as discussed below.

1.1.1 TYPE-I Frame

A valid TYPE-I frame is a minimum of 64 bits in length, including flags, and must conform to the following structure:

F, T, FUN, S, P, I, FCS, F

where

F = Flag Sequence
T = Destination Address Field
FUN = Function Field
S = Source Address Field
P = Parameter Field
I = Information Field (optional)
FCS = Frame Check Sequence

Frames containing only link control sequences form a special case where no I field is present.

The TYPE-I frame structure is illustrated in Figure D-1. Each element of the frame is detailed under Section 1.2.

1.1.2 TYPE-II Frame

A valid TYPE-II frame is a minimum of 80 bits in length, including flags, and must conform to the following structure.

F, T, FUN, AC, S, P, I, FCS, F

where

AC = Access Code Field

and all other elements are identical to the elements of the TYPE-I frame.

Frames containing only link control sequences form a special case where no I field is present.

The TYPE-II frame structure is illustrated in Figure D-2. Each element of the frame is detailed under Section 1.2.

1.1.3 Frame Type Limitations

Command frames may be either TYPE-I or TYPE-II frames. Response frames are always TYPE-I frames. On a given trunk, all command frames must be of the same type.

1.2 FRAME ELEMENTS

1.2.1 Flag Sequence (F)

All frames open and close with the flag sequence. This sequence has the binary configuration 01111110, that is, a zero-bit followed by six one-bits, followed by a zero-bit.

The opening flag serves as a position reference for the address and control fields, and initiates transmission error checking. The closing flag serves as a position reference for the frame check sequence.

Transmitters must send only complete 8-bit flags. All receivers attached to the data link must search continuously, on a bit-by-bit basis, for the flag sequence. Thus, the flag sequence provides frame synchronization.

An F may be followed by a frame, another F, or an idle line. An F which closes a frame may also be used as the opening F on a following frame. Any number of F's may be transmitted between frames.

Since the F sequence brackets and synchronizes the frame, it must be prevented from occurring in any field of the frame. This is accomplished by the zero-insertion technique described below.

Each transmitter must insert a zero-bit following five contiguous one-bits anywhere between the opening and closing flag sequences. The insertion of the zero-bit thus applies to the address, control, information, and FCS fields and effectively prevents the fortuitous transmission of the F sequence 01111110.

Figure D-1. TYPE-I Frame Structure

Figure D-2. TYPE-II Frame Structure

Each receiver after detecting the opening flag (start of frame) continuously monitors the received bit stream and removes any zero-bit which follows a succession of five contiguous one-bits. Note that zero insertion at the transmitter follows the computation of FCS and that zero-deletion at the receiver precedes the FCS check process.

Receivers must be capable of recognizing the following sequences as containing one or more flags.

a. FCS 01111110 T FUN

←Flag→

The flag can be detected as a valid closing flag for one frame and a valid opening flag for the next frame whether or not the first frame was addressed to this receiver. This is a combined opening and closing flag.

b. FCS 011111101111110 T · FUN

←Flag→

←Flag→

Although transmitters must send only complete 8-bit flags, receivers will detect this sequence as 2 flags.

c. 01111110 XX . . . . . XX 01111110

where X is any combination of bits not comprising a flag. The number of X bits can range from 0 upward.

1.2.2 Destination Address Field (T)

The Destination Address Field (T) immediately follows the opening flag of a frame and precedes the function field.

Two addressing modes are defined for this addressing field by the state of the most significant bit (bit-zero):

1.2.2.1 Unique Destination Address

Bit zero = 0. The remaining 7 bits uniquely identify the destination.

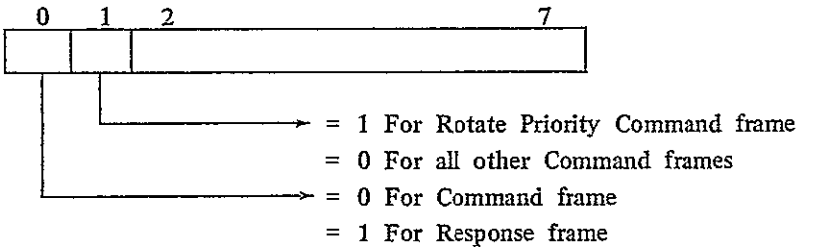
1.2.2.2 Global Destination Address

Bit zero = 1. The frame is directed to all units on the trunk.

1.2.3 Function Field (FUN)

The Function Field (FUN) is located immediately following the destination address field and preceding the source address field (TYPE-I) or the access code field (TYPE-II). The function field is used to convey the commands and responses necessary to control the data link.

The two most significant bits of the function field are reserved for link control as follows:



Note that bit 1 is reserved for command frames, but not for response frames.

The remaining six (6) bits of the function field are available for specified commands and responses.

1.2.4 Access Code Field (AC)

The Access Code Field (AC) immediately follows the function field in the TYPE-II frame; this field does not exist for the TYPE-I frame.

The AC is a "key" which must match the "lock" on the receiving unit in order that the frame be accepted. If the match is not made, the frame is discarded.

1.2.5 Source Address Field (S)

The Source Address Field (S) immediately follows the function field in the TYPE-I frame or the access code field in the TYPE-II frame.

The Source Address Field identifies the unit which sent the frame.

1.2.6 Parameter Field (P)

The Parameter Field (P) immediately follows the Source Address field, preceding the information field.

The parameter field provides control or status for the control message or response message respectively.

1.2.7 Information Field (I)

The data link control is completely transparent to the contents of the I field. The I field may, therefore, consist of any number of bits, in any code, related to character structure or not and limited only by system requirements. The I field is unrestricted as to length but it should be recognized that typical length is contingent on system requirements and limitations beyond the link level. Factors limiting I field length may include channel error characteristics, PDC buffer size, and the logical properties of the data.

The fortuitous occurrence of a flag or abort sequence within the I field is prevented by the zero-insertion technique described in paragraph 1.2.1.

An I field with a length of zero is specifically permitted.

1.2.8 Frame Check Sequence (FCS)

Each frame includes a 16-bit frame check sequence (FCS) immediately following the I field (or the P field if there is no I field) and preceding the closing flag. The FCS field serves to detect errors induced by the transmission link thus validating transmission accuracy. The 16-bit FCS results from a mathematical computation on the digital value of all bits (excluding inserted zeros) in the frame including the destination address, function access code, source address, parameter and information fields.

The process is known as cyclic redundancy checking using the CCITT Recommendation V.41 generator polynomial of $X^{16} + X^{12} + X^5 + 1$. The transmitter's 16-bit remainder value is initialized to all ones before a frame is transmitted. The binary value of the transmission is premultiplied by X^{16} and then divided by the generator polynomial. Integer quotient values are ignored and the transmitter sends the complement of the resulting remainder value, high-order bit first, as the FCS field.

At the receiver the initial remainder is preset to all ones and the same process is applied to the serial incoming bits. In the absence of transmission errors the final remainder is 1111000010111000 (X_0 thru X_{15} respectively).

The receiver will discard a frame in error. Subsequent retransmission of the errored block is under control of error recovery procedures.

1.3 ADDITIONAL CONVENTIONS

1.3.1 Interframe Time Fill

Interframe time fill may be transmitted to maintain the link in an active state. Time fill may also be used to avoid timeouts and to hold the authority to transmit.

When used, interframe time fill must be a series of contiguous flags which are contiguous to the closing flag of one frame and the opening flag of the next frame.

1.3.2 Abort

Abort is the process by which a PDC, in the act of transmitting a frame, decides before the end of that frame to terminate in an unusual manner which will cause the receiver to discard the frame.

Aborting a frame is accomplished by transmitting at least seven consecutive one-bits with no zero-insertion. Receipt of seven contiguous one-bits is interpreted as an abort.

1.3.3 Invalid Frame

An invalid frame is defined as one not properly bounded by an opening and closing flag or one which is too short, e.g., less than 64 (TYPE-I) or 80 (TYPE-II) bits between flags. An aborted frame is an invalid frame. A PDC will ignore an invalid frame.

1.3.4 Order of Bit Transmission

The order of transmission for all fields is most significant bit first.

2.0 DEVIATIONS IN PDCCP FROM CDCCP

2.1 FRAME TYPES

CDCCP defines one frame type; PDCCP defines two (one incorporates an access code, one does not).

2.2 FIELD DEFINITIONS

PDCCP defines three additional fields:

S — Source Address Field
P — Parameter Field
AC — Access Code Field

2.2.1 Address Field Definition

The Address Field (A) of CDCCP is redefined as the Destination Address Field (T) in PDCCP as follows:

A (CDCCP)	T (PDCCP)
N Octets in length where N $\geqslant$ 1	Single Octet
Two Addressing Modes	Single Addressing Mode
Group Addressing	No Group Addressing
Global Address = 11111111	Global Address = 1XXXXXXX
Null Address = 00000000 (ignored by all stations)	No Null Address
Address Field refers to Source or Destination	Address Always Destination

2.2.2 Control — C (Function — FUN) Field

C (CDCCP)	FUN (PDCCP)
N Octets in length where N $\geqslant$ 1	Single Octet

←————Content Definitions Totally Different————→

2.3 MINIMUM LENGTH OF FRAME

CDCCP	PDCCP
48 Bits	64 Bits (TYPE-I) 80 Bits (TYPE-II)

2.4 ORDER OF BIT TRANSMISSIONS

CDCCP	PDCCP
Address, Control, Responses, and Sequence Numbers are Low Order Bit First (Bit 2^0 first) Data (I Field) Any Order FCS = Most Significant Bit First	All Fields Most Significant Bit First

3.0 TRANSACTIONS

3.1 GENERAL

A transaction is a dialogue between two units on a trunk. At least one of the units must be active (a PDC). Active units can transmit command or response frames. Passive units can transmit only response frames. The multiplexed loop controller is an example of a passive unit.

A dialogue can be viewed as a set of command and response transmissions. Consider two units, A and B:

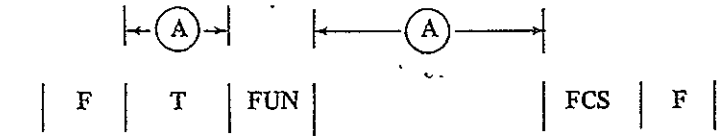
IDLE TRUNK
A transmits command frame(s) to B
Reserved trunk
B transmits response frame(s) to A
Reserved trunk
A transmits command frame(s) to B
o
o
o
o
B transmits response frame(s) to A
IDLE TRUNK

A dialogue, therefore, consists of one or more sets of command/response transmissions.

Each PDC contains one to four trunk interfaces. These interfaces are called Trunk Control Units (TCU). The TCU interfaces the trunk and the PDC buffer. The TCU selects frames from the trunk and presents frames to the trunk.

The TCU analyzes certain frame fields as part of the select process. Likewise the TCU generates certain fields when presenting frames to the trunk. These TCU operations, as they pertain to the PDCCP frame, are discussed in detail in the following sections.

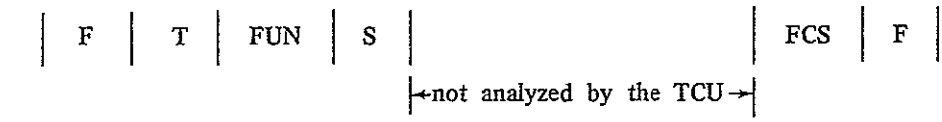
3.2 TRANSMIT COMMAND FRAME – TYPE-I OR TYPE-II



F = Flags
FCS = Frame Check Sequence
FUN = Bit 0 and 1 forced to zeros. Bits 2-7 come from the buffer.

(A) = not generated by the TCU

3.3 RECEIVE COMMAND FRAME – TYPE-I

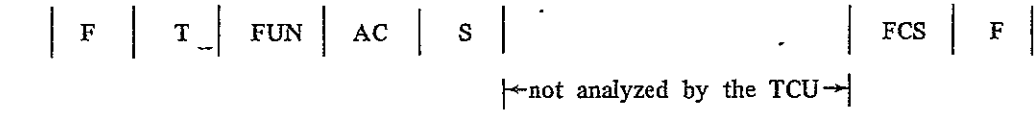


T = The transmitted address which must match this unit's address.

FUN = The function field which must identify this frame as a command frame (bit 0 = 0). The TCU recognizes a small set of functions.

S = The source field of the first correctly received frame; it is saved by the TCU

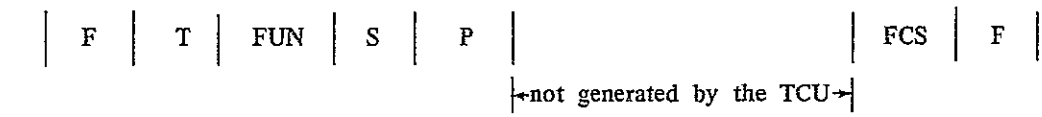
3.4 RECEIVE COMMAND FRAME – TYPE-II



AC = The transmitted access code which must match the physical access code of this unit.

All other fields are recognized identically to the TYPE-I frame of Section 3.3.

3.5 TRANSMIT RESPONSE FRAME



T = The value saved when receiving a command frame. See Section 3.3.

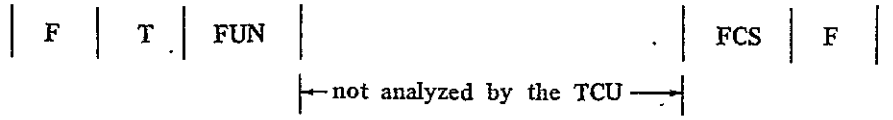
FUN = The response, including bit 0 = 1.

S = The physical address of this unit.

P = The parameter field; contains unit status..

T, FUN, S, and P are inserted by the TCU.

3.6 RECEIVE RESPONSE FRAME



T = The transmitted address which must match this unit's address.

FUN = Bit 0 must be set.