Introduction
to the
Fountainhead (FHP) System

Chapter 1
Introduction

This paper introduces the Fountainhead Project (FHP), a new architecture developed by Data General for introduction in 1979. FHP represents a family of high capacity systems with multiprocessor capability which take advantage of advanced concepts in both architecture and technology.

PROJECT ORGANIZATION

As the organization chart of Figure 1-1 shows, the FHP hardware team consists of five groups:

a) Physical - this group is responsible for the electrical and mechanical aspects of the FHP systems.

b) Firmware - This group will create all of the microprograms, and all of the tools entailed in that task.

c) CPU - This group, with the I/O group, performs the systems design, and is fully responsible for the memory and job processor portions of the processors.

d) I/O - In addition to participating in the systems definition effort, this group is responsible for all design associated with I/O.

e) Maintenence - This group is responsible for the diagnostic and fault isolation logic, the maintenence strategy, and the diagnostic programming unique to each system.

Coordinating these groups is a documentation and systems engineering program. A similar organization exists for the software effort. Working hand-in-hand, these twin organizations hope to make a significant impact on the data processing marketplace.

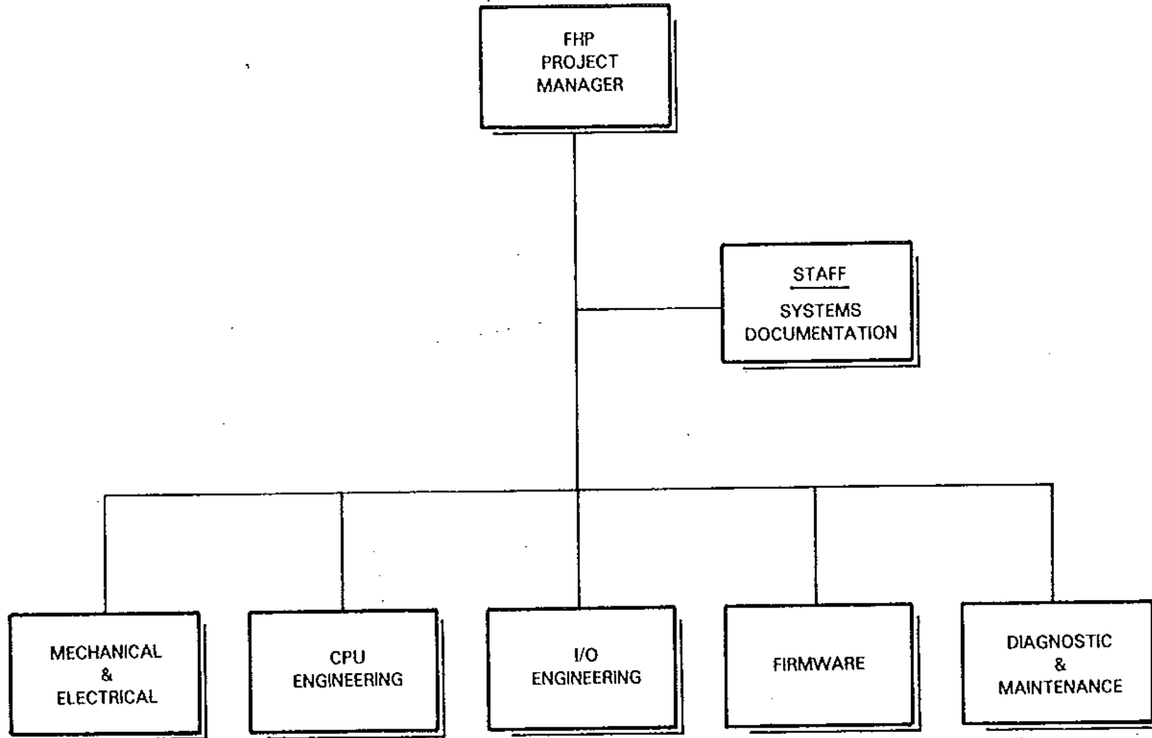

Figure 1-1- FHP ORGANIZATION CHART

MARKET

The FHP will top Data General's product line and cover the top of the DGC critical market. Figure 1-2 shows a graphical representation of the marketplace in terms of relative computer performance (in 360 equivalent KOPS) plotted against cost. Two main areas of the graph contrast the classical minicomputer area and the classical mainframe curve.

The location of the FHP within this marketplace is indicated. Note the high performance-to-cost ratio. This position is one which DGC hopes will capture a large, and expanding share of the end-user market.

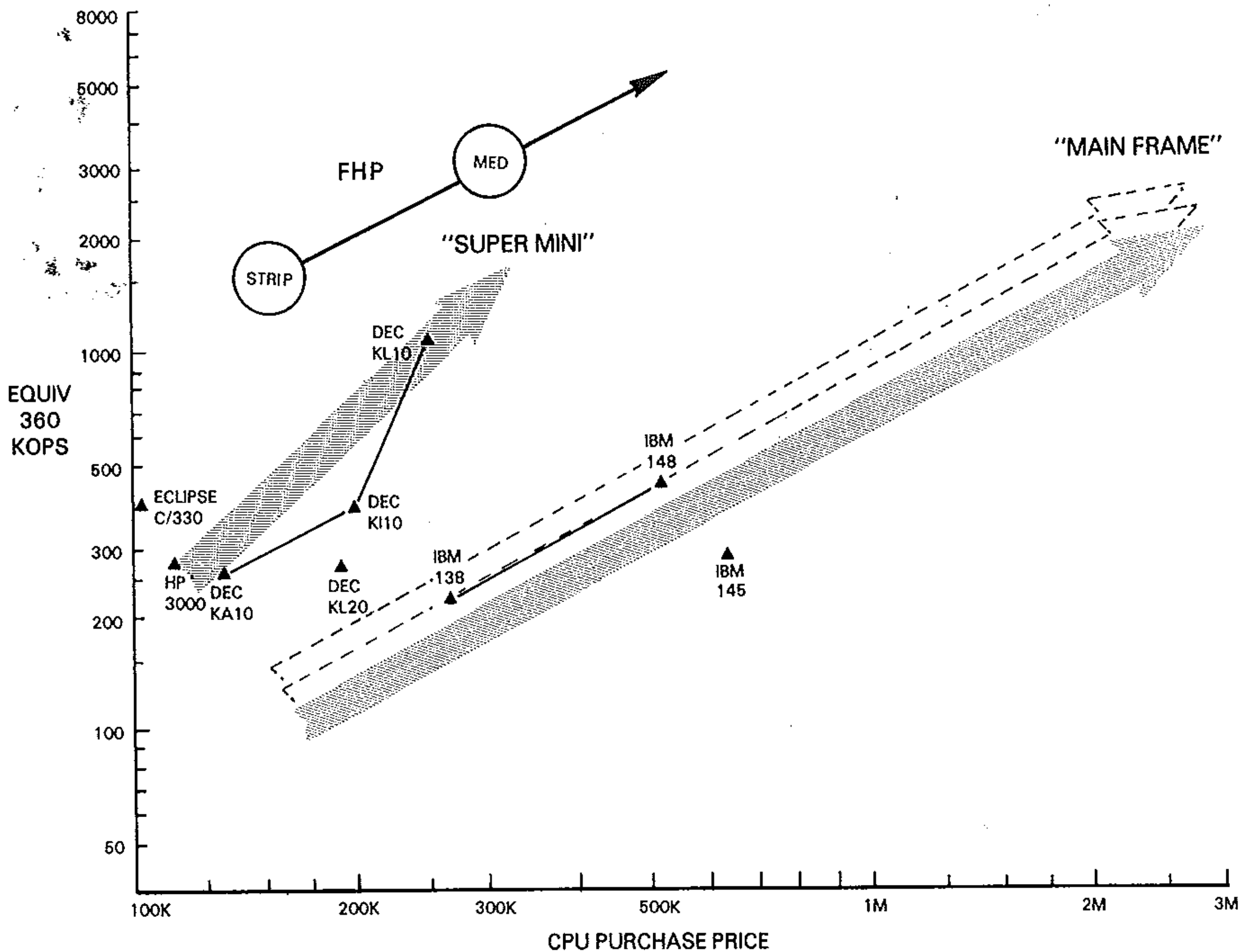

Figure 1-2- CRITICAL MARKET PERFORMANCE

S-LANGUAGES

A major source of architectural inefficiency in present day computers lies in the disparity between the language the programmer writes in and the language the computer runs in. Most modern processors use an *instruction set* implemented on a control store. Programs written in higher level language are translated into this *machine language* by a compiler. The extent of the inefficiency is such that in most systems, any repeated segments of the operating system are written directly in assembly language.

The FHP represents a departure from the standard method described. In place of a single machine language, the FHP supports a series of machine languages, called S-Languages, each one optimized for the higher level language it supports. These are implemented in firmware on a writeable control store, so each program runs on the effective architecture that supports it best. Refer to Figure 1-3 for a graphical description of this simplified explanation.

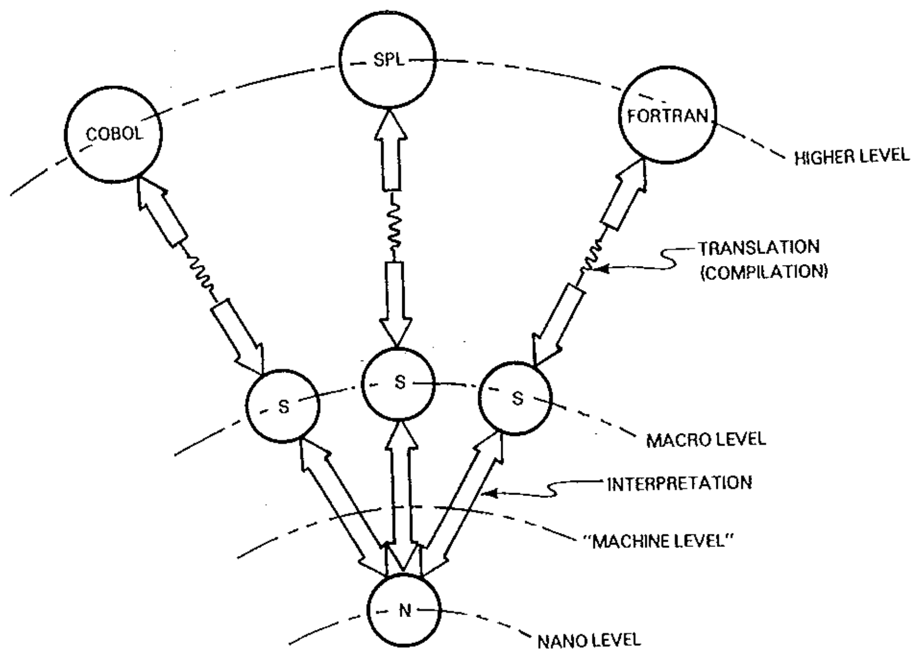

Figure 1-3- S-LANGUAGE GRAPHICAL REPRESENTATION

As Figure 1-3 shows, the basic machine is created by a language-independent nano-level control pattern -- the lowest level of semantic intensity. The nano-level is that machine level at which the hardware control lines are individually specified. In essence, the basic machine is defined by the microword field definitions. Because each language runs on nano-code tailored for its peculiarities, program execution times are optimized. Note, in particular, that in FHP we have no definition or concept of a *machine level* or micro-level. In a conventional system, the intermediate level must exist.

In addition, a language created for just this purpose will be used for operating system and executive functions as well as compiler writing, thereby minimizing software overhead.

PHYSICAL ORGANIZATION

FHP will be a family of processors, initially consisting of three sizes, two of which are scheduled for rapid implementation. This discussion will center on the larger of the two -- the *medium* -- and will note the differences in the smaller and earlier system -- the *strip*.

Figure 1-4 illustrates the physical layout of the medium system, the basic subunit of which is the bay. Only three types of bays comprise the system; the classical division of memory, job processor, and input/output has been followed here. Bays, which are physical as well as logical entities, consist of a series of modules.

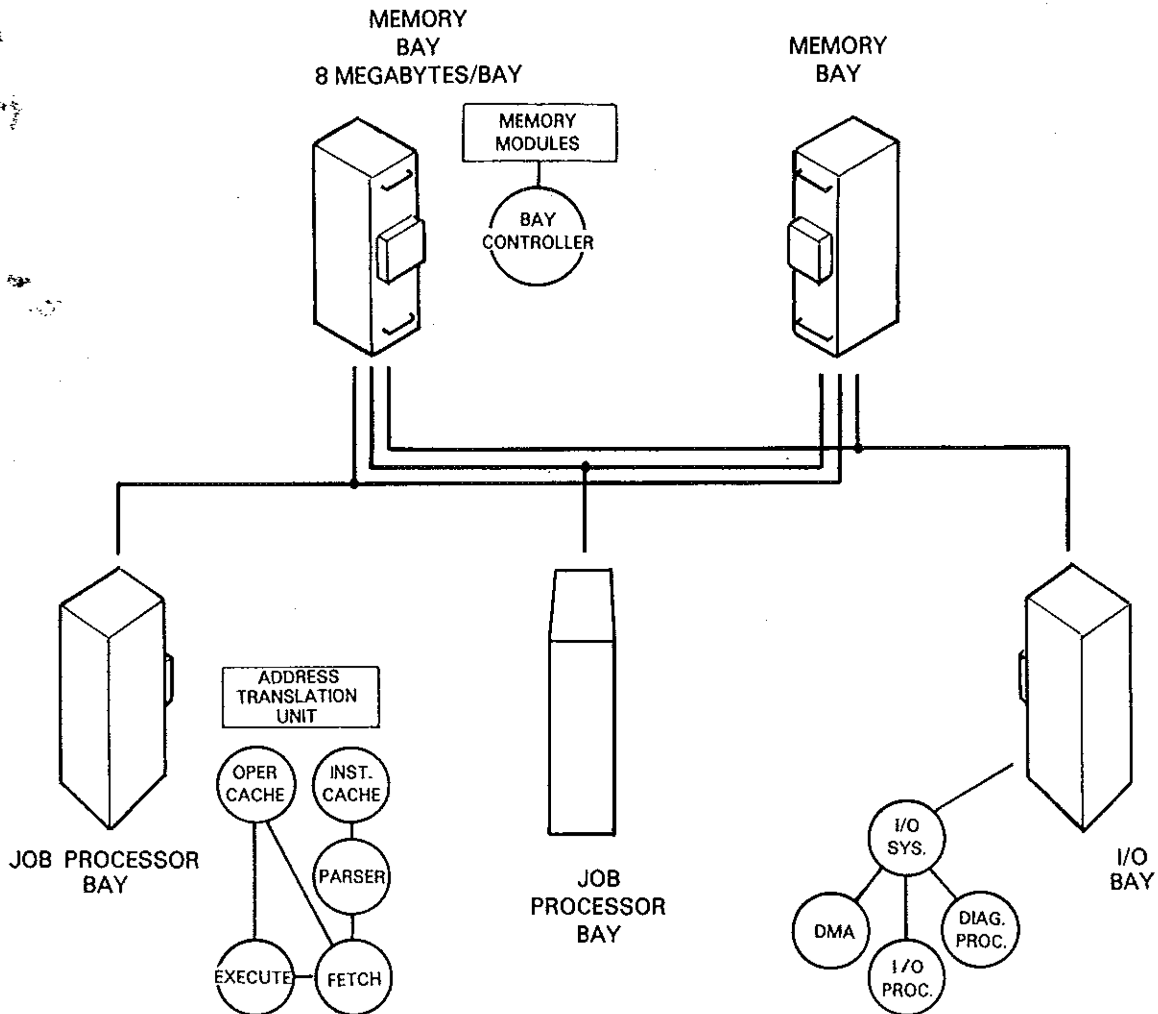

Figure 1-4- MEDIUM SYSTEM

Memory

Each memory bay consists of eight million bytes of block-addressable storage, divided into four banks of eight modules each, and a bay controller. The medium system will have two bays, and therefore total 16 million bytes of direct storage. Each memory access obtains a block of four 64-bit words.

Job Processor (JP)

Arithmetic and control functions for the system are handled by the Job Processor (JP) Bay. This is the most complex of the three bay types. It contains a pair of cache memories, one for procedure, one for data, feeding a tripartite pipelined central processor. The central processor begins with a parser, which accepts an instruction bit stream from the cache and extracts the command and data information, and sends them to the appropriate functional unit. Next in line, the fetch/interpreter accepts address information, verifies security, generates the logical address from the information given, performs addresss modification as required, and directs the address to the appropriate storage element. Some instructions may also be executed entirely within the fetch/interpreter or parser.

Most instructions are completed in the execution unit, which accepts operands and instructions and produces results (which are returned, if desired, to storage based on further information provided by the fetch/interpreter). Single pass arithmetic and logical operations are performed on operands up to 64 bits.

All units are individually microcoded, and have a cycle time of 120ns. In the medium system, two of these job processor bays may be used.

Input/Output

I/O functions are accomplished in the third bay. This bay uses an I/O processor (in the class of an Eclipse) to control all of the information transfers to/from the other bays to/from the peripheral devices. It has the responsibility of establishing all direct memory access operations, and shares with the Job Processsor the responsibility for keeping the security of the system intact. Toward this end, it performs the translations from logical to physical address spaces required for I/O operations.

In another external area, a diagnostic processor will monitor all errors and failures within the system and display relevant information to the operator. It will have the ability to exercise all critical lines in the FHP, and to take some corrective actions. This processor is the cornerstone in the FHP commitment to a maintainable system with minimal down-time.

Strip System

The strip system will be fully compatible with the medium, but will be smaller. All of the elements described above will be present, but scaled to fit into a single bay (rather than 5). As an example, the memory will consist of a single bank with its controller. The other units will be similarly scaled down in capability, but all functionality will be retained.

Chapter 2
A More Detailed Look At S-Languages

The concepts surrounding S-languages are key to the FHP, and crucial to an understanding of what we are trying to accomplish.

DEFINITION

An S-language is a macro-level language which, hopefully, carries as much as possible of the semantics of the parent language. Each S-language relates directly to a parent higher level language. Several S-languages can exist for a single higher level language, but only one is necessary (or, probably, desirable) for a processor.

STRUCTURE

In an S-language machine, each language operates on an abstract machine -- a machine that appears to the language to be processing exactly what the language requires. In fact, the same physical machine is used to run all of these different S-language programs. Microcoded emulators create the environment required by each S-language to run efficiently.

Creation of an S-language entails a detailed study of the parent language to determine those structures which appear most frequently, and would therefore be good candidates for inclusion. Classically, most S-languages have been written around a structure of three variables and one operation specification per instruction "word." This structure is based on the realization that the concept of accumulator is alien to the higher level language, and is therefore transparent to the S-language. The classic instruction, therefore, takes two operands, performs an operation between them, and stores the result in a specified place. Several other structures are possible, depending on the way the high level language structures its data, specifies its operations, organizes its loops, and performs similar operations. Options are also available to personalize any number of instructions to other formats if linguistic frequency analysis (or whatever) convinces the designer that introduction of such a format would enhance program execution.

Usage on FHP

Ultimately an FHP computer will be capable of running with many S-languages in its repertoire, each swapping into writable control store as required. Presently, we have four S-languages under development. Three of these implement FORTRAN, COBOL, and BASIC. The fourth implements a language that is also under development on this project, called System Programming Language (SPL). All system programs will be written in this new language by DGC programmers.

Chapter 3
Some Architecture Philosophy

At the time this is written, several critical aspects of the architecture are in development, so it would be pointless to list specifics. A superior approach in any event is to give you a simplified picture of just what we are trying to accomplish architecturally, to impart a feel for the kind of design we are looking for.

S-LANGUAGE SYNTAX

Any computer requires instructions of some form. In the FHP, these take the form of *nibble* -structured bit streams. For those unfamiliar with the term, a nibble is, reasonably enough, half a byte. Our instruction word begins with an operation code that may be one, two, three or four nibbles long. The exact number will vary with the S-language and with the frequency of appearance of the operation in the language. Frequently specified operations will get shorter operation codes, thereby optimizing memory space usage. Exact length of the operation code will be determined by the parser upon an examination of the leading bits of the instruction.

Following this portion of the instruction will come a series of address specifications, detailing the locations from which the operands are to be extracted and to which the answer is to be returned. On occasion, depending on the S-language, there may be more than two operands to be combined, or a register may be designated as a temporary storage. Our intent, at all times, is to optimize the execution time and minimize overhead functions. Address designations are typically two, three, or four nibbles long, and may include modification information (indexing or indirect addressing) to be passed to the fetch unit. Literal value designations may also be used.

OBJECTS AND UID's

An object is a collection of information in the computer grouped together for programmer convenience. In the FHP, each object is assigned a Unique IDentifier (UID) which it retains forever. Essentially a 128-bit wide name, the UID number remains with the object, distinguishing it exactly from any other object ever created on any FHP system. An object can contain up to 4 billion bits. UID guarantees unambiguous transportability of software from one FHP to another.

SECURITY

Much emphasis has been placed on full system security as an integral part of the FHP architectùre. The problem addressed is to allow each specific user conditional access to a particular portion of a data base depending on who the user is, what job he is doing at that moment, what type of access he requires, and what overall permission he has been granted. Thus, a user will not have a simple access to read or write a file. Rather, the exact access will depend also on the context in which that access request is made (the *domain* of the process).

ADDRESS GENERATION

Addresses in the FHP are to the bit level. An instruction "word" may be of arbitrary length and begin on any 4-bit boundary. Efforts are made to create compact instruction and data structures to optimize memory use.

Because speed is a crucial parameter, procedure or data resident in memory and required by a program must be accessed in the shortest possible period of time. To attain this speed, a series of *accelerators* must be created. Typically, these will consist of a series of look-up tables and compacted address representations.

The exact nature of the address structure is still under development. Architecturally, the goal is to define a structure that permits "shorthand" versions of address referents and object descriptions as often as possible without precluding the use of fully expanded information to be accessed when desired.

OTHER AREAS

Many other items come within the scope of he architecture, such as interrupt structure, accounting, and kernel structure. Throughout the definition, we have emphasized the integrity, universality, speed and cost constraints in roughly that order. For an up-to-the-minute description of the architecture see *Preliminary Architecture Manual.*

Chapter 4
A More Detailed Look at Physical Organization

We can take a look at the physical implementatiom of the three bays projected for the medium system as they exist today. Again, because the nature of an emerging architecture is quite dynamic, it is unlikely that these designs will be reproduced exactly in the actual product. They do give a good feel for the proposed nature of the machine, and should be considered in that light.

MEMORY

A block diagram of the memory appears in Figure 4-1. Illustrating the complete bay, the diagram can be thought of as having three parts, the storage area, the port (address and data) area, and the control area.

The storage area consists of eight modules (four mod-pairs) in each of four banks. MOS technology is used for these modules, each of which contains 32K 64-bit words. Control functions center around a Bay Controller, which enables requests to find the appropriate bank and ensures successful sequencing of the internal bus, and a Buffer Management Table (BMT), which ensures successful sequencing and handles the multiple copy problem. Error detection and single bit error corection is performed in the ERCC module, while a bus arbitor resolves incoming bus conflicts.The other modules in the bay act as address and data bus driver/receivers, and physical terminators for compatible transmission.

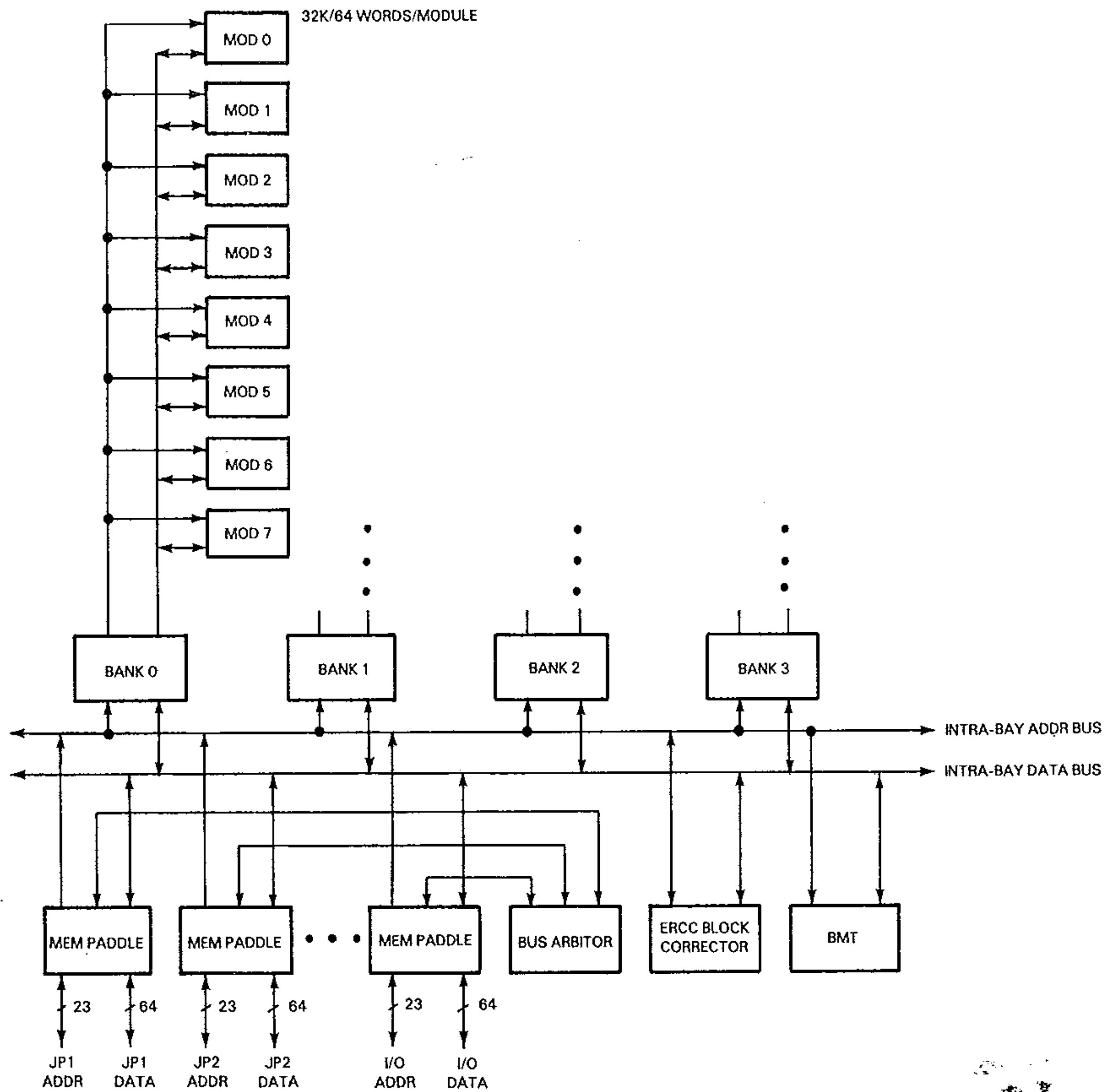

Figure 4-1- MEMORY BLOCK DIAGRAM

JOB PROCESSOR

Figure 4-2 shows a rough block diagram of the JP bay. Pipelining and parallelism are used extensively to provide the speed required of the processor.

Overview

Operations commence with the parser, which issues a request to the instruction cache for the instruction pointed to by the program counter. The cache locates and properly aligns the bit stream, and sends it to the parser. There, the stream is parsed into the operation code, which is decoded, and a series of addresses, which are shipped to the fetch unit. The parser also determines starting microaddresses for the actions to be performed by the fetch and execute units. The fetch acts on the information typically by verifying the security of the request, creating a logical address from the information, and sending operand requests to the operand cache. This portion of the cache replies by sending operands to the execute unit. There they are combined in accordance with the microroutine stored at the starting address supplied by the parser, and the result delivered to a register, and ultimately to the cache. Pipelining ideally occurs when the parser is working on Instruction 3, the fetch is obtaining operands for Instruction 2, and the execute is performing the operation called for by Instruction 1.

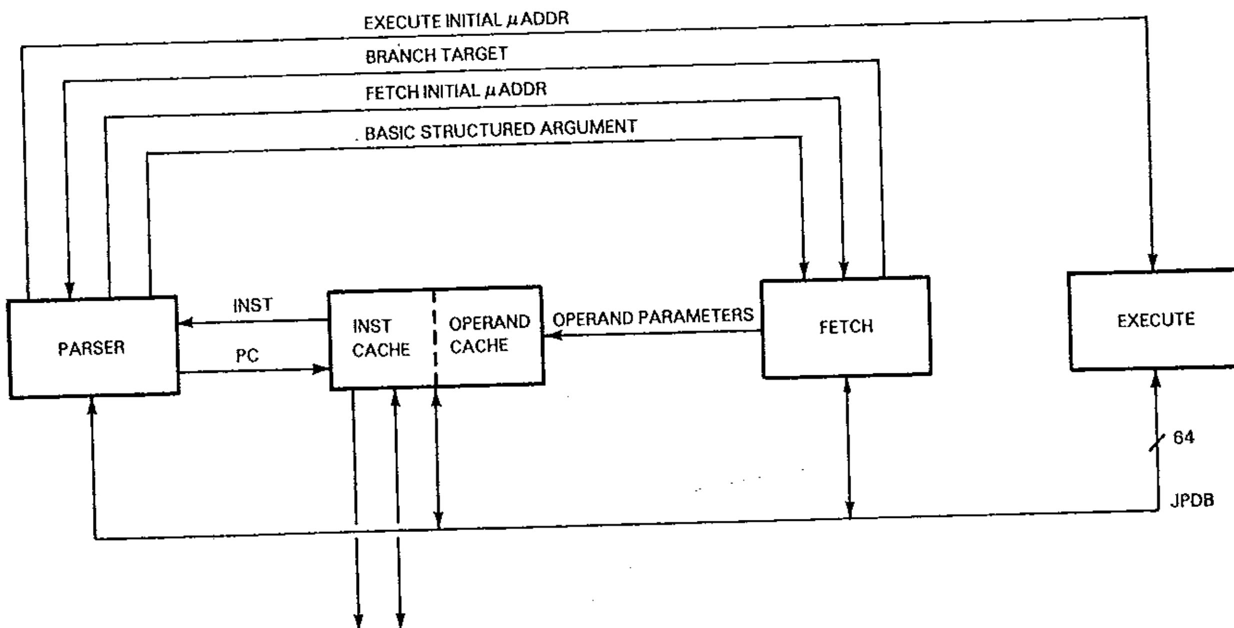

Figure 4-2- OVERALL BLOCK DIAGRAM

Cache

The cache bears the burden of interfacing between the block oriented memory and the nibble and bit structured JP. Also, as in any cache, the primary function is accelerating transmissions between the two major functional units by buffering a certain amount of information from main memory in the hope that it will be available for use by the JP. Translation from logical to physical address space is also the responsibility of the cache, as is the task of verifying that an address for a subsequent (pipelined) instruction is not a write address for a previous one.

The physical structure of the cache is highly dependent on the architecture and the manner in which address translations are accomplished. Figure 4-3 shows a possible implementation of the cache, in which the interfaces to the memory, the Job Processor Data Bus, and the instruction bus are noted.

That part of the cache concerned with providing the instruction output, consisting of a small associative store, an address translation unit, and a pending block file, comprises the item termed the *instruction cache.* That larger area with essentially the same structure represents the *operand cache.* The remaining hardware shown is used as a physical interface to the main memory.

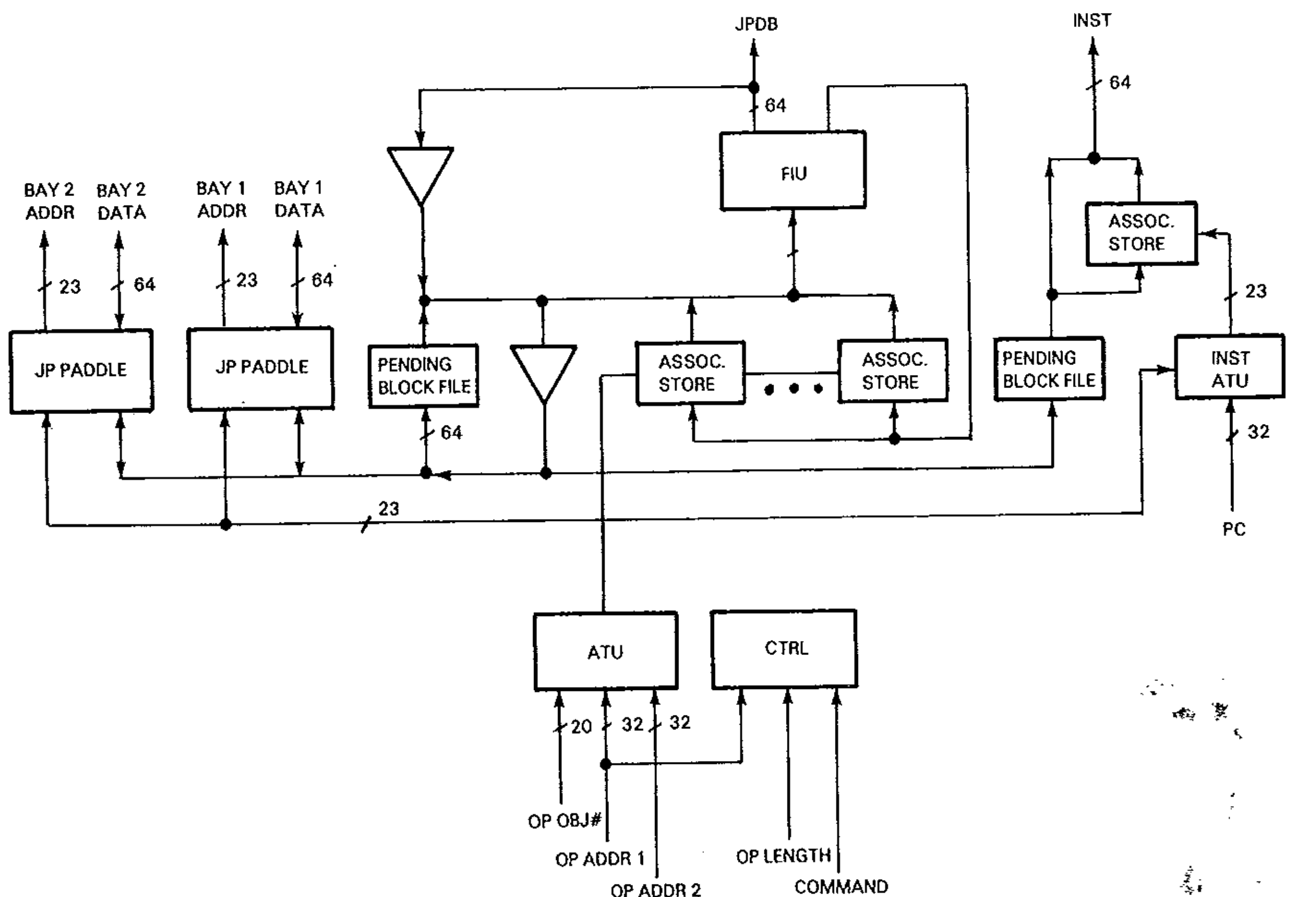

Figure 4-3- CACHE BLOCK DIAGRAM

Parser

The parser accepts the intruction bit stream as it comes from the instruction cache and separates operation code and operand address information. It transmits the operand address information to the fetch, then determines correct starting microprogram addresses for both units and sends them over dedicated busses.

When no branch is required, the parser immediately proceeds to the next instruction, which is determined by adding the length of the previous instruction to the former program counter value. In cases where a conditional branch is specified, the parser awaits the results of the branch decision before continuing. Figure 4-4 depicts a possible implementation of the parser.

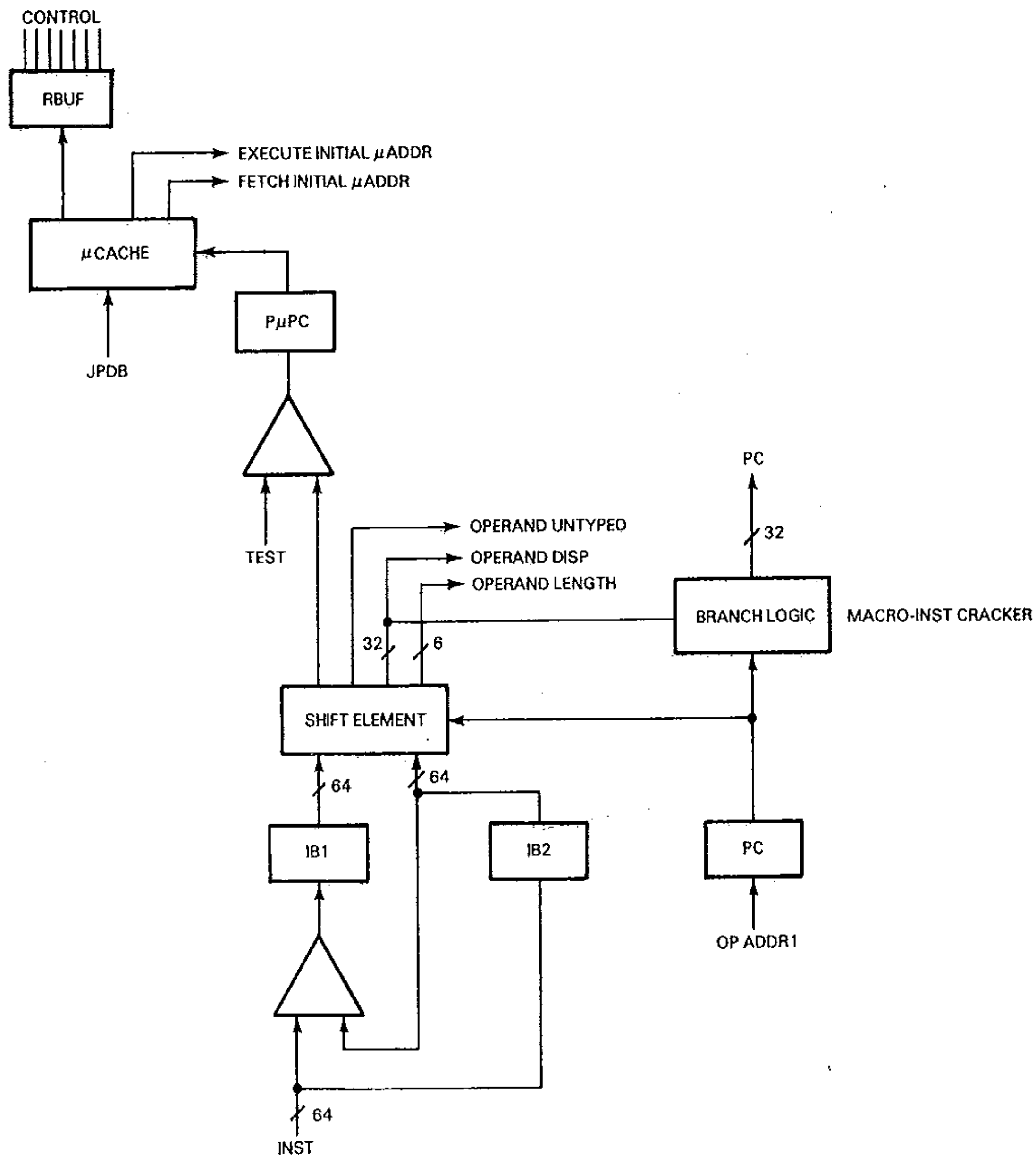

Figure 4-4- PARSER BLOCK DIAGRAM

Fetch/Interpreter

As the heart of the address generation structure, the fetch implementation is dependent on the actual address and Access Control conventions adopted by the FHP architecture. Figure 4-5 shows an implementation of the fetch using Object Register addressing, in which a register holds a starting address and a displacement to a specific entity within the Object. This is one of a number of methods under investigation.

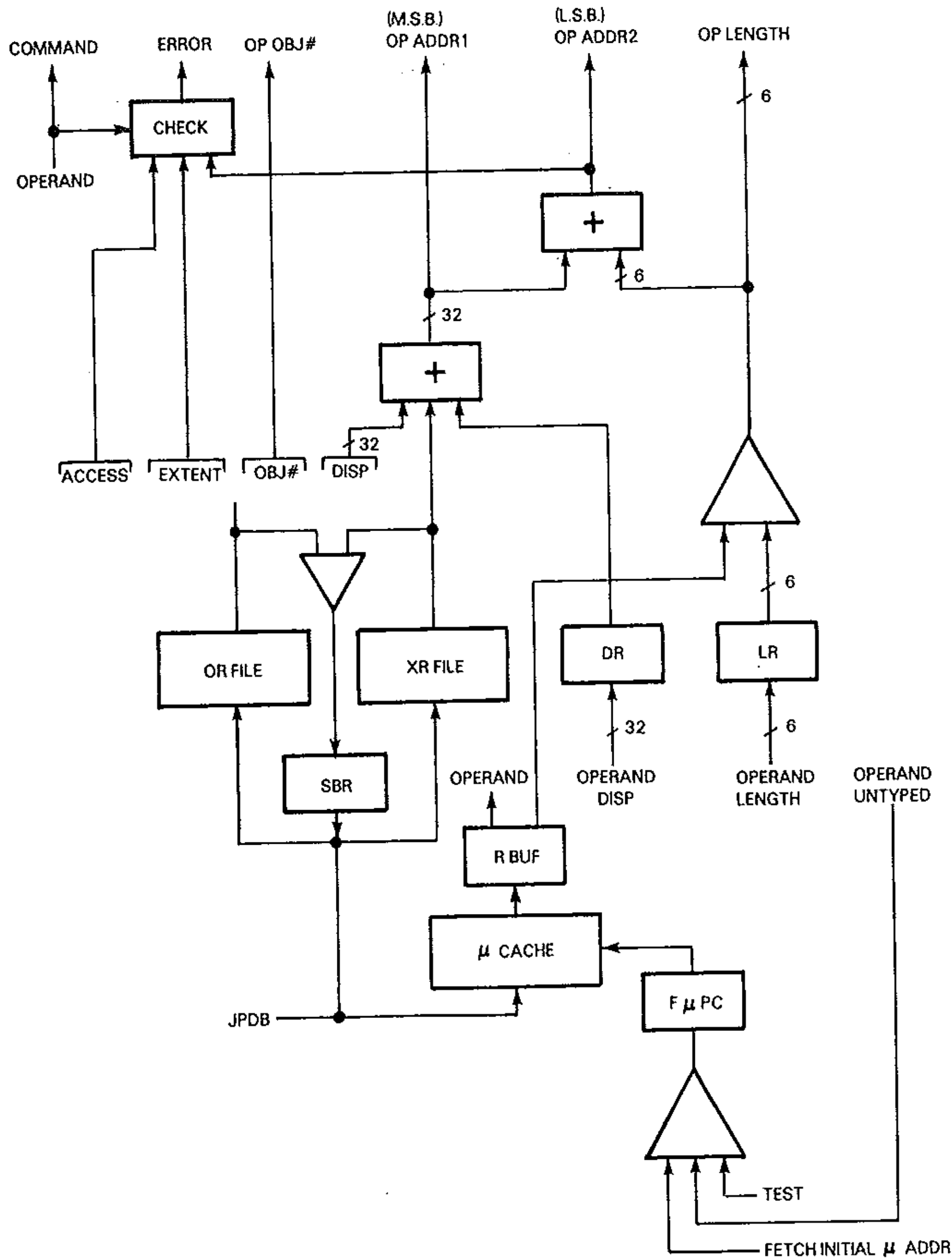

Figure 4-5- FETCH BLOCK DIAGRAM

While this scheme may change, the essential areas of the fetch can be seen. As in all units, the fetch contains a microprogram memory that gives it its sequence of operations. The interface to the Job Processor Data Bus leads to a simple adder where address oriented additions are performed. Any more difficult arithmetic functions are always done in the execution unit. The size and length of the operand is determined, as is the starting address in memory. All of this information is then made available to the cache, which gets the operands for the execution unit.

The function of the interpreter, which also appears as part of this section, is to provide a micro- programmable mechanism by with the specific peculiarities of an S-language operand and data structure can be readily imposed onto the hardware.

Execution Unit

Here, all of the actual calculations are performed. The execute unit is a microprogrammed, bus structured unit capable of high speed arithmetic and logical combinatorial operations.

Figure 4-6 shows a version with several available options that will not be included with all models. Essential portions of the unit include the operand queue, which buffers the operands until all are present and the execute unit has finished its previous operation, a standard arithmetic element, and some form of temporary storage, in addition to the microprogram memory. Options depicted include a special purpose decimal arithmetic element, and another for enhancing the speed of floating point operations. Results are sent directly to the cache, which already contains the necessary information to place the answer in the appropriate storage location.

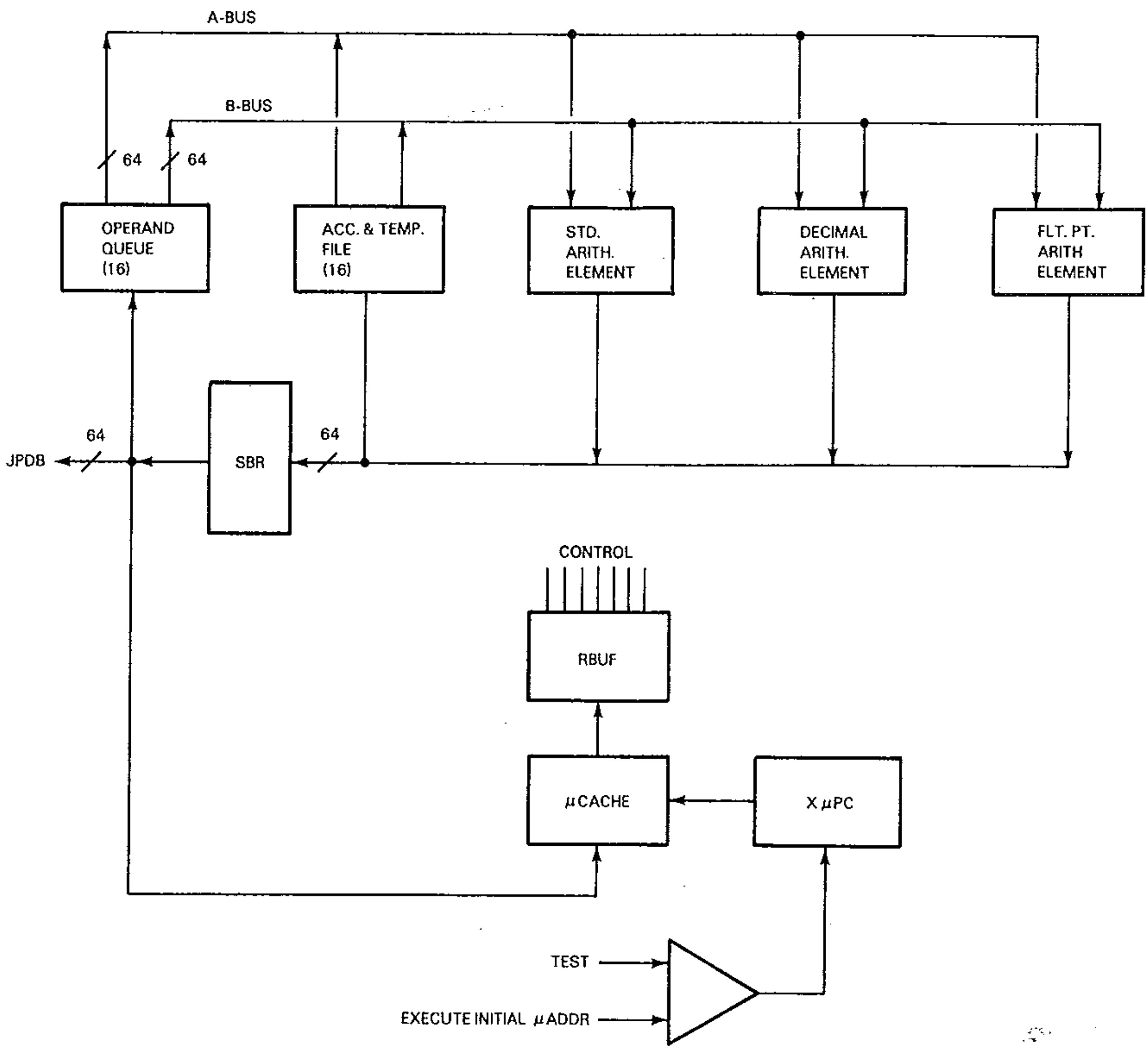

Figure 4-6- EXECUTE BLOCK DIAGRAM

INPUT/OUTPUT

The input/output section of the machine consists primarily of an input/output processor and a network of transmission lines and peripheral devices. I/O oriented address translation and security are functions of this processor, as well as the establishment of information transfers from and to external devices in a direct memory access mode. Figure 4-7 shows a projected general block diagram of the I/O system.

Present projections for the I/O include provision for a high speed bus up to 40 feet long, capable of operation as a standard Nova data channel, or as part of a custom defined FHP/channel-controller complex.

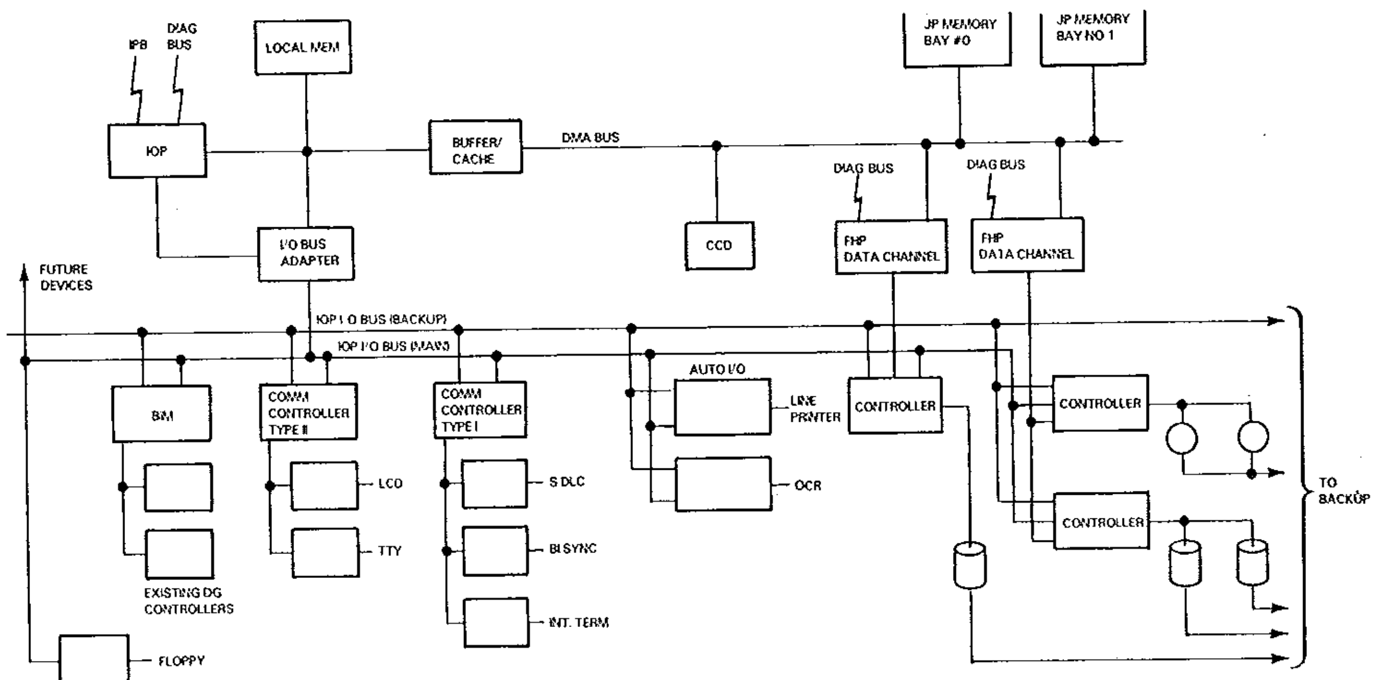

Figure 4-7- I/O SYSTEM ARCHITECTURE

DIAGNOSTIC PROCESSOR

A second center of computational power external to the central processor lies in the diagnostic processor. This unit controls a separate bus that leads to a diganostic island in every module in an FHP system. From this single processor, an operator can test all combinations of lines within a processor, and failures can rapidly be narrowed down to the module level.