FHP Product Overview
ARCHITECTURAL GOALS

The Fountainhead Project (FHP) was started in July, 1975. One of its major tasks was to define a new architecture for Data General computers. This effort has had four major goals throughout:

- Provide a very large logical address space.
- Support controlled sharing of information in a distributed, multi-user environment.
- Provide efficient, cost-effective execution of programs written in high-level languages.
- Allow a broad range of effective implementations.

THE FHP PROGRAM ARCHITECTURE

To meet these goals, the project has designed a program architecture that consists of three major components: data services, instruction services, and process services.

DATA SERVICES

Memory

- Universal, shared address space
- Object-oriented addressing
- Up to $2^{30}$ independent objects
- Objects named by Unique Identifiers (UIDs)
- Up to $2^{31} - 1$ bits per object
- Bit-granular addressing
- Total addressability greater than $10^{32}$ bytes
- Single-level storage: files accessible as working storage
- Extent checking on every reference

Protection

- Objects as protected containers of data
- One protection mechanism for all information (files, working storage, code, etc.)
- Subjects as units of authority; composite of principal, domain, process, and tag
- Instantaneous validation of memory accesses
- Access Control List protection model
- Multiple, non-hierarchical domains
- Support for Extended Type Objects

INSTRUCTION SERVICES

Instructions

- Simple instruction streams made up of 16-bit syllables
- Multiple-operand instructions, including:
  - One-, two- and three-address arithmetic
  - One-, two- and three-address compare-and-branch

Operand Addressing Modes

- Tailored to high-level languages
- Architectural Base Registers:
  - Frame Pointer
  - Static Data Pointer
  - Procedure Base Pointer

N-address procedure invocation
- S-languages tailored to needs of high-level languages
- Wide variety of data types:
  - Signed and unsigned integers of any width
  - Single and double precision IBM floating point
  - Packed and zoned decimal up to 18 digits
  - Bit and character strings of arbitrary length
  - Pointers that can denote any bit in any object
- One 16-bit syllable per operand reference
- Granular reference syllables capture base, displacement, indirect scalars
- Name Table references for structured data: packed items, array elements, strings, substrings, and based records

**PROCESS SERVICES**

**Call and Return**

- Universal argument passing mechanism
- Universal stack frame format
- Universal Return operator
- Neighborhood Call for normal subroutine calls
- General Call for changing environments

- Cross-domain case of General Call allows processes to change access rights securely; supports secure subsystems and abstract types

**Miscellaneous Services**

- Powerful debugging support
- Dynamic Linking and Associated Addressing
- Condition management and signalling
- Nonlocal transfer of control

**ARCHITECTURAL BENEFITS**

A user sees the benefits of the FHP program architecture in terms of obstacles that have been removed. The user can program effectively in a more structured and straightforward manner, so his productivity is enhanced. Specific benefits include:

- Support of a large, shared, object-oriented address space. No address space management of any kind. No restriction on per-process address space. No obstacles to sharing or protecting information. Transparent multiprocessing made possible.
- Support of high-level language programming with excellent performance. No need to resort to assembly language.
- Support of structured design and top-down programming with high-speed call and return. No need to violate structured design principles to accommodate inefficient modularization tools.
THE SPRINT HARDWARE

The SPRINT computer is the first implementation of Data General's new FHP architecture. The FHP SPRINT computer supports a major subset of the architecture, including S-languages, Namespace Addressing, and object-oriented addressing and protection.

Features

- Standard Schottky TTL logic circuitry
- 110-nanosecond cycle time
- Supports up to 16 megabytes of semiconductor memory
- Can address the entire FHP address space (2^108 bytes)
- 32-bit data paths to and from the memory system
- Semiconductor caches accelerate common operations, including data transfers, address generation, and protection checks
- A separate I/O Processor supports two I/O channels: a Data Channel (DCH) and a Burst Multiplexer Channel (BMC)
- A Data General MPT/100 is the System Console and diagnostic processor
- A diagnostic "scan chain" monitors the computer's logic circuitry

The SPRINT computer consists of four independent modules: memory system, Job Processor, Input/Output System, and the System Console and diagnostic processor.

The computer can support a full complement of peripherals. The System Console and a real-time clock are standard. A moving head disk, tape drive, and programmable interval timer are required.

The demonstration machine has 4 megabytes of semiconductor memory, a 2-megabyte fixed-head disk, a 192-megabyte disk, and two 277-megabyte disks. It also has two magnetic tape units, a lineprinter, a DCU/200, 16 CRTs, and an MCA.

Memory System

The memory system can support up to 16 megabytes of bit-addressable semiconductor memory. It uses four-way interleaved arrays and an 8K-byte data cache. The memory arrays have Error Recovery and Correcting (ERCO) circuitry.

The Job Processor and I/O System share the memory system. They transfer data to and from memory through separate 32-bit buses. The memory system has three data-request ports. The Job Processor uses two of the ports: one to request S-language instructions, the other to request data. The I/O System uses the other port to request data and instructions.

Job Processor

The Job Processor (JP) implements the addressing, protection, and S-language features of the FHP architecture. The Job Processor has a basic cycle time of 110 nanoseconds. It uses pipelining techniques and several caches to accelerate common operations. The JP executes most S-language instructions in 660 nanoseconds to 2 microseconds.
The Job Processor has three separate modules: a prefetcher, a Fetch Unit (F-unit), and an Execute Unit (E-unit). The prefetcher supplies a steady flow of S-language instructions to the F-unit. The F-unit determines what operations are to be performed, and generates operand addresses. It generates most addresses in 220 nanoseconds. The F-unit uses several caches, including an argument base cache, name cache, address translation cache, and a protection cache. The F-unit has an 80K-byte Writable Control Store that contains the system's S-language interpreters.

All arithmetic calculations are performed by the independent E-unit. The E-unit executes an integer addition in 220 nanoseconds and a floating-point addition in 550 nanoseconds. It has its own 8K-byte Writable Control Store. The E-unit can work on an arithmetic calculation while the F-unit is generating a new address.

**Input/Output System**

The Input/Output System (IOS) has an independent I/O Processor (IOP). The IOP is based on a Data General S/140. Floating-point instructions are executed in microcode. It runs a modified version of the Advanced Operating System (AOS). The IOP supports two data channels: the Data Channel (DCH) for programmed I/O and the Burst Multiplexer Channel (BMC) for direct-to-memory data transfers. Both channels support standard Data General device controllers. The IOP gets its data and instructions from the memory system over a 32-bit bus. The IOP also has its own 32K-byte Writable Control Store.

**System Console**

SPRINT uses a Data General MTP/100 as a System Console. The System Console has a CRT display, keyboard, and two flexible disk drives. System operators use the console to load and communicate with the operating system.

The console's microNOVA also acts as diagnostic processor. The console is connected to a diagnostic scan chain that provides "windows" into the main computer's logic circuitry. Maintenance personnel can troubleshoot the system by testing and setting hardware registers from the console.
BIOS is the SPRINT operating system. It provides:

- a large, object-oriented, one-level virtual memory;
- multiprogramming, with a sophisticated Medium-Term Scheduler;
- powerful, stream-oriented, logical I/O;
- an AOS-like, hierarchical file system;
- advanced protection features;
- a complete interactive environment; and
- dynamic linking.

**CONFIGURATION**

A SPRINT system includes a Job Processor (JP), which supports the FHP architecture, and an Input/Output Processor (IOP), which supports the ECLIPSE instruction set with enhancements for communication with the JP. These two CPUs share a common main memory. The smallest recommended memory size is 2 megabytes. BIOS will support up to 64 megabytes of main memory.

**INPUT/OUTPUT SYSTEM (IOS)**

BIOS has two major components: the IOS and the Job Processor Operating System (JPOS). Our current IOS is a specially tailored version of the standard AOS system and runs on the IOP. The major modifications that have been made to AOS to create IOS are:

- a communication mechanism for passing messages between the JP and IOP;
- additions to the file system so that files can be referenced by JP programs as FHP objects;
- low-level disk I/O support for the JP virtual memory mechanism; and
- the ability to run JP programs from the AOS CLI.

SPRINT/BIOS can be presented in one of two ways. Currently it can be viewed as a standard AOS/ECLIPSE system with an attached (FHP) job processor. All AOS program and support services can be made available on the IOP, with additional services available on the JP. An alternative approach is to restrict user access to the IOP, making AOS services available only through product variances. There may be marketing and support advantages to this approach.

**JPOS**

JPOS is the component of BIOS that runs on the Job Processor. It in turn has two major components: KOS and EOS. KOS (kernel operating system) runs on bare JP hardware and provides basic operating system facilities: virtual memory (FHP objects), processes (for multiprogramming), and communication with the IOP. For efficiency’s sake, significant support for KOS is provided by JP microcode. For example, the lowest level of processor multiplexing is performed by microcode.

EOS (extended operating system) extends the primitive KOS facilities to provide a “typical” operating system to users, including a file system, logical I/O, and process services. In the current BIOS, EOS is highly constrained because it must provide consistency between the JP and IOP environments.

The rest of this brief highlights JPOS services.

**Virtual Memory**

The FHP architecture includes a one-level virtual memory, composed of 2⁶⁰ independent objects. Each object can be up to 256 megabytes long. This virtual memory is implemented in hardware, microcode, and operating system software.

**Multiprogramming**

JPOS multiprograms the JP, to make more effective use of hardware resources. The multiprogramming is primarily oriented towards a batch environment, but good response is provided for a small number of interactive processes. JPOS controls multiprogramming with a sophisticated Medium-Term Scheduler, which is described in the FHP product brief, “The Medium-Term Scheduler.”

**Logical I/O**

JPOS supports a media-independent input/output protocol called Logical I/O. It provides a fixed set of operations against different kinds of data sources (e.g., user data files, terminals, printer files). Logical I/O operations are grouped into access methods, which reflect the organization of the data source. Currently two access methods are supported: sequential and direct. Data is transferred in units called logical records. There are three logical record formats: fixed,
variable length, and data sensitive.

**File System**

JPOS provides complete access to the AOS file system, thus allowing JP and IOP programs to share data. There are mechanisms to ensure consistency, if the accessing from the two CPUs is separated in time.

**Protection**

The FHP architecture has advanced protection features, including a powerful Access Control List mechanism for protecting the contents of objects, and nonhierarchical protection domains, which can be used to encapsulate Extended Type Managers. JPOS currently implements a subset of these protection features: simplified Access Control Lists and four protection domains implemented as rings. The two innermost rings are allocated to the operating system, the next ring to database software, and the outermost ring to user programs.

**Interactive Environment**

JPOS will include a complete interactive environment, based on the CP (Command Processor). The CP provides a common command interface for all SPRINT interactive utilities; it is described in detail in a separate product brief.

**Dynamic Linking**

JPOS supports dynamic linking, which allows the binding between separately compiled program units to be delayed until execution time. Essentially, dynamic linking allows the bind step of the normal "compile-bind-run" sequence to be deferred during program development.

**SUMMARY**

BIOS provides a complete set of "typical" operating system facilities, as well as support for the unique features of the FHP architecture.
GENERAL DESCRIPTION

One of the major advantages of the FHP architecture to data base management is its single-level, universally addressable memory. Data base systems on traditional architectures with per-process address spaces must provide memory mapping on top of that provided by the operating system to map pages of a data base into their address space. This results in a duplication of effort and overhead by the data base system and operating system.

In FHP, all memory mapping is performed by the virtual memory system. Programs can directly reference any data in a system without explicit system calls to perform I/O. Data base pages in main memory can be referenced with little more overhead than references to local program variables.

The accompanying graph illustrates the efficiency of this memory system in improving data base system performance. By simply increasing the physical memory of a system, we obtained direct and dramatic improvements in data base performance. The extra memory is automatically allocated to the data base system's "buffer pool," without any modifications needed to the data base system.

TEST DESCRIPTION

The test involves an RTAM data base consisting of 120,000 records of 200 bytes each, with an index over those records having keys of 20 bytes each. The total size of the data base is over 32 MB. The records are distributed randomly throughout the pages of the data base in an order different than that of their keys.

This test was performed with memory configurations of 2 MB, 4 MB, and 6 MB.

The illustration shows the results.

CONCLUSION

The single-level memory of the FHP architecture allows for an efficient implementation which can take great advantage of increased real memory.
FEATURES

The common FHP compiler components consist of programs and data structures common to compilers written by RTP/ISD. The programs are:

- Parse Table Builder
- Schema Editor
- Schema Manager
- Parser
- Logger
- Optimizers
- Schema Builder
- Trunk Optimizer
- Code Generator
- Assembler
- Collator
- Page Formatter

The data structures are:

- Base Tokens
- Parse Tables
- Parse Tree
- Schema
- Intermediate Tree Language
- Trunk
- Protocode
- Procedure Object
- Log

In the following section, a more complete description of the compiler is given. In this description, the common compiler components are consistently printed in italics to show how many there are and what a vital part of the compiler they are.

GENERAL DESCRIPTION

Each of the processes—parsing, the semantic phase, and generating code—are described in detail.

Parsing

The compiler writer describes the syntax rules of the language in a logical representation which is input to the Parse Table Builder. This common component edits the rules for consistency and produces Parse Tables and Base Tokens as well as a listing of the rules.

The compiler writer uses the Base Tokens in constructing the Lexer. The Lexer reads the user's source program as a stream of characters and breaks that stream into meaningful words or tokens as the user intended.

These tokens are passed to the Parser which uses the Parse Tables to reconstruct and assemble the original statements written by the user into Parse Trees. If a syntax error is found by the Parser, an error message is passed to the Logger.

The Parse Trees and other information about the program are stored in a data base called the Schema. Access routines for the Schema are supplied by the Schema Manager. The Schema Editor is an aid in preparing invariant Schema input for generating code.

Semantic Phase

The Parse Trees are read by the semantic phase. Semantic rules unique to each language are used to translate the Parse Trees into an Intermediate Tree Language. The semantic phase accesses the parse trees via the Schema Manager.

If the user's source program violates any semantic rule of the language, an error message is passed to the Logger.

Optimizers examine the program at this stage. Optimization includes eliminating unexecuted or redundant code and rearranging code for better execution.

Code Generating

The optimized program is translated into Protocode or a Procedure Object. This is done by the Code Generator and the Assembler. The Code Generator accesses the working program using the Schema Manager.
The Code Generator uses rules for this language which are stored in a part of the Schema called the Trunk. The compiler writer builds the Trunk using the Schema Builder and improves it using the Trunk Optimizer.

Finally, listings are produced. All of the error messages stored by the Logger in the Log now are rearranged by the Collator into the sequence in which the user wrote the program rather than the sequence they were generated by the compiler. The Page Formatter then creates the finished listing file.

**SUMMARY**

The advantages to such extensive use of components in compiler development are as follows:

- Better concentration of effort on compiler performance by having a few experts “tune” the process for everyone.
- Improved reliability because so much of the compiler is common to every compiler and, thus, is tested by each compiler project; an error corrected in one is corrected in all.
- Faster, better compiler development: each new compiler is available sooner, but has the same performance characteristics as all of the other compilers.
- A common user interface and uniform printed output are provided to all compiler users.
- Portability is enhanced through less reliance on unique code.
- All of the information needed for debugging and performance measuring utilities is captured in the Schema.
FEATURES

FHP FORTRAN is a compiler that fully conforms to the FORTRAN 77 standard. It also has several extended features that are natural and easy to use. The extended features include:

- enhanced character manipulation facilities,
- inline procedures,
- inline intrinsic functions,
- source text inclusion,
- additional I/O keywords to exploit system features,
- extended data types including INTEGER*1, INTEGER*2, and COMPLEX*16,
- bit manipulation facilities through standard ISA functions,
- interlanguage procedure invocation, and
- recursive procedure invocation.

Several features have been added to ease conversion from the FORTRAN 66 standard to the FORTRAN 77 standard as well as conform to various industry standard features. These features include:

- data initialization in type statements,
- type statements with optional length indicators,
- 66-style DO loops, and
- Hollerith constants.

FHP FORTRAN is highly compatible with the Data General MV/8000 F77 compiler, allowing a smooth transition between systems.

COMPILER PERFORMANCE

Compiler performance has been steadily increasing (as illustrated in “Compiler Performance Chronology” above) since its first introduction to an FHP prototype in October, 1980. Presently, the FHP FORTRAN compiler has competitive performance qualities; we plan to at least double the compile speed.

Working set and memory usage also have an effect on performance. In the past year, the working set of the compiler has been significantly reduced. This is also shown in the graph above. The average memory usage, versus working set limit, versus the number of page faults is shown in a separate graph, below.
AUDIENCE

FHP FORTRAN is designed for a wide range of users, from the student requiring easy-to-understand diagnostics to the experienced professional requiring a wide range of facilities and exemplary code quality. Both will find FHP FORTRAN easy to use.

DEVELOPMENT ENVIRONMENT

The program development environment of FHP FORTRAN users is greatly enhanced by the system-wide dynamic linking facility. Dynamic linking lets several independently compiled modules be invoked without the need for static binding. This eliminates a time-consuming and often costly step in the development environment.

FUTURE GROWTH

Several additional features are planned for the FHP FORTRAN system. These features include enhanced debugging facilities, vector capabilities, expanded aggregate data types, and data base interfaces.

SUMMARY

FHP FORTRAN is a competitive product, conforming to the FORTRAN 77 standard. It generates high-quality object code with easy-to-understand diagnostics.
**FEATURES**

This benchmark indicates excellent FHP performance. The features which contribute to this performance include:

- an asynchronous EBOX, which allows instruction execution and data retrieval to overlap;
- a FORTRAN S-language, which allows common functions to be coded in microcode;
- the FHP architecture, including namespace addressing and neighborhood call; and
- the FORTRAN 77 Compiler, which performs local optimizations and has an inline expansion capability.

**GENERAL PERFORMANCE**

The Whetstone benchmark is a widely accepted measure of a machine's FORTRAN computational power. It has run on numerous architectures. The test stresses the following functions:

1) Floating point arithmetic
2) Integer arithmetic and control
3) Intrinsic functions
4) Subroutine calling

The following graphs compare FHP's performance on this benchmark to a wide range of competitors. Following the graphs is an example of inline expansion.
<table>
<thead>
<tr>
<th>Calling Program</th>
<th>Subroutine</th>
</tr>
</thead>
</table>
| PROGRAM MAIN
REAL A,B,C,SQ
A = SQ(B) + SQ(C)
END | SUBROUTINE SQ(X)
REAL SQ,X
SQ = X * X
END |

**Translation without Inline Expansion**

<table>
<thead>
<tr>
<th>CALL SQ,B,temp-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SQ,C,temp-2</td>
</tr>
<tr>
<td>PADD temp-1,temp-2,C</td>
</tr>
</tbody>
</table>

**Translation with Inline Expansion**

<table>
<thead>
<tr>
<th>FMUL B,B,temp-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL C,C,temp-2</td>
</tr>
<tr>
<td>PADD temp-1,temp-2,C</td>
</tr>
</tbody>
</table>

**SUMMARY**

The FHP single-precision Whetstone number is very exciting. The combination of a sophisticated optimizing compiler with an inline expansion capability and a FORTRAN S-language results in performance double that of an MV/8000, PE 3240, PRIME 750, or VAX 11/780. Even without the inline capability, FHP performance is one and a half times that of these machines. The FHP double-precision inline Whetstone number is approximately one and a half times that of the above-mentioned machines.

**INLINE EXPANSION**

When writing programs, a programmer often wants to localize an algorithm (e.g., square a number) in a routine without incurring the cost of a CALL/RETURN on every invocation of that routine. Inline expansion provides this capability. Consider the example of "A = SQ(B) + SQ(C)" above, where SQ is a function which returns its argument squared. Without inline expansion, SQ must be called twice in the evaluation of the expression. With inline expansion, no call occurs since the code for SQ is placed in the main code stream.
FEATURES

- Implements ANSI '74 at the high-intermediate level and includes desirable extensions
- Produces very dense and efficient code using special COBOL S-instructions
- Through unlimited addressing, allows for arbitrarily large programs and unlimited data space
- Includes some COBOL '80 features, e.g., structured programming constructs
- Will provide access to a relational data base system (DBMS)
- Will allow enhanced user interaction, through screen management capabilities
- Will support enhanced debugging facility by interfacing with an interactive debugger

GENERAL DESCRIPTION

The COBOL language system provides FHP users with a high-level language, powerful data management facilities, and flexible program development aids. This COBOL was designed and implemented to provide language compatibility with other manufacturers' COBOL as well as with Data General's AOS COBOL.

The language meets the 1974 ANSI standards as defined in American Standard Programming Language COBOL, ANSI X3-23-1974. It implements the following modules at intermediate to high levels:

- Nucleus
- Table handling
- Sequential I/O
- Relative I/O
- Indexed I/O
- Sort/Merge
- Library
- Interprogram Communication

The input/output modules are implemented at the highest level to allow the user the greatest flexibility in designing and accessing his file systems.

The high-level sort/merge facility lets the user intermix ascending and descending key values and specified collating sequences.

EXTENSIONS

In addition to the standard character data types (alphanumeric and decimal), COBOL supports variable-length binary and packed decimal data formats. Other language extensions include DBMS interface capabilities, structured programming constructs, n-dimension arrays, screen management, and calls by reference or content.

PERFORMANCE

The COBOL S-language was designed to permit a close mapping of COBOL source statements onto executable instructions. The first result of this effort is high-density program code, as much as 50% smaller code files. The second result is the improved execution speed of a COBOL program. A COBOL program executing on FHP may be many times faster than one on a computer costing much more (as exhibited by a predicted U.S. Steel productivity index greater than that of an IBM 370/168).

DATA MANAGEMENT

Special language constructs enable FHP COBOL to use the powerful relational data base system offered by FHP. COBOL statements can manipulate the data base with a relational calculus as well as the traditional record-at-a-time access. Automatic mediation of concurrent access by multiple users is handled by DBMS.

MEMORY REQUIREMENTS

Due to FHP's unlimited address space, the user is free from any constraints on the size of his program or data storage area. Programs are automatically segmented in a way that optimizes their performance.

PROGRAM DEVELOPMENT AIDS

COBOL will be able to execute under the supervision of PROBE, an interactive debugger. PROBE
allows the programmer to set breakpoints within his program, view and/or modify data at the source level, and resume program execution.

**CURRENT STATUS**

The front end of the COBOL compiler is very near completion. Over 90% of the COBOL Compiler Validation Service (CCVS) tests will compile. As can be seen from the U.S. Steel demonstration, run-time testing is making steady progress. The DBMS, PROBE, and screen management extensions must be added and much more testing of the compiler and its generated code must be performed.

**SUMMARY**

COBOL offers all the advantages of source compatibility, both internal and external to Data General, as well as the performance advantages of the FHP architecture. Additionally, the ability to use a high-performance, relational data base and a interactive source-level debugger make FHP COBOL an extremely attractive product.
## DESCRIPTION

Since 1965, timing tests have been conducted on various computers that support COBOL compilers. The slowest system tested (IBM-1460) was selected to represent a unit of CPU productivity. All other systems are measured by dividing the time for the IBM-1460 by the time for the system being tested.

The U.S. Steel benchmark program consists of eleven timed loops, summarized below:

1) A series of mathematical calculations with the operands in DISPLAY format

2) The same calculations with the operands in a BINARY format

3) Comparisons using group data items

4) Comparisons using elementary data items

5) Moves using 100-character data items

6) Moves using figurative constants

7) Subscripted moves with the subscript in a DISPLAY format

8) Conditional moves, i.e., one or more tests precede the move

9) Iterative subscripted moves with the subscript in a DISPLAY format

10) Subscripted moves without subscript evaluation

11) 132-character print line, formatted with various moves such as decimal insertion and floating dollar sign

## FHP PERFORMANCE

COBOL performance on FHP is very exciting. The combination of a sophisticated compiler, an optimizing code generator, and the COBOL S-language results in COBOL performance which greatly exceeds any other machine in FHP's cost range. The above illustration shows the results of the U.S. Steel benchmark program for FHP and several selected computers.

As you can see from the illustration, FHP COBOL performance compares very favorably with that of the most powerful computers (regardless of price) on the market today.
FEATURES

- A full range of structured programming facilities.
- Syntax which promotes program readability.
- A complete set of primitive types including signed and unsigned integers, enumerations, sets, records, pointers, arrays, and strings.
- A powerful type definition facility including named and abstract types.
- Strong typing rules to prevent unexpected conversions.
- Support for modules. These are packages of related definitions (literals, types, variables and routines) which promote the logical organization and centralized definition of program entities.
- Import and export control over literals, types, variables and routines.
- Separate compilation of modules.
- Interface checking during compilation.
- Inline expansion facilities to promote abstraction without loss of code efficiency.
- Program control over the representation of types.
- Full capabilities for safe machine-level data access.
- Explicit control over generated machine code where needed.

- Excellent code generation including: peephole optimizations, simple code routing, powerful constant folding and address and storage layout optimization.

GENERAL DESCRIPTION

SPL is a high-order language which addresses the problems of constructing and maintaining large software systems. It is similar to, but simpler than, the Department of Defense's common high-order language, Ada. Thus, it provides many of the procedural and data structure features of languages like Pascal and PL/I as well as features to support modularization and abstraction.

COMPILING SPL

SPL programs are currently translated into executable code using a cross-compiler called SPLX, which runs under AOS/VS and generates FHP Procedures and Objects.

SPLX processes each module as a single unit. A module contains a list of names available outside the
module, a list of other modules which provide information, and the text of any literals, types, variables or routines which are defined in the module. From these, the SPL compiler generates a procedure object file and an interface data base called a schema. The SPL compiler uses this schema to obtain needed information while compiling other modules.

This feature of extracting information from the results of other compilations is fairly new in programming languages. Of the currently popular languages, only Ada has a similar facility. The benefits of this facility are as follows:

- It lets you write the definition of an entity in one place. That definition can be imported and used in many other contexts.

- It simplifies the packaging of interfaces and data structures which are known outside of a subsystem.

- It boosts compiler performance. The current compiler can copy the information it needs in about 10% of the time it would take to compile the source.

Running SPL

The SPL compiler produces a directly executable procedure object for each module. At run time, these procedure objects call each other and share variables with the aid of the operating system's dynamic linking facility. This extremely flexible method of running a program is best used while the program is under development. A change to the program can be accomplished by merely editing and recompiling the module which contains the bug. No bind step is required.

DEBUGGING SPL PROGRAMS

SPL programmers use the PROBE interactive debugger to analyze their executing programs. PROBE's source level breakpoints and data accessing commands are automatically available for any procedure object.

Occasionally a programmer needs finer control over debugging than source level commands provide. This control is available from PROBE and augmented by a stand-alone disassembler (DAS). It produces a listing of a procedure object's machine code, cross-referenced to the original source lines and data references.

With this disassembly in hand, a programmer can easily stop his program in the middle of evaluating an expression and see the values of intermediate results which are not named in his original source.

BINDING SPL

Once a program is working, it is desirable to bind it together into a single procedure object. This simplifies distribution of the program as well as improving its speed and space efficiency.

The SPL compiler has a companion intelligent binder called SQUASH, which:

- groups procedure objects together.

- statically links and optimizes the interconnection among the procedure objects. (Modifies Call instructions and optimizes data references.)

- reorganizes the modules' static data to optimize addressing.

- combines and updates the debugging information so that PROBE can find all the routines and data.

SUMMARY

SPL is a high-order language for building large software systems. The language and its compiler provide the control you need at every level from the macroscopic to the microscopic.
The System Programming Benchmark Suite was developed at RTP in 1980, and has been used as a learning and evaluation tool by all parts of the FHP project. In addition, the suite has been executed in PL/I on other Data General machines.

The suite consists of five test segments that represent the kinds of code that appear in large system programs, such as compilers, utilities, and operating systems. The tests manipulate integer, logical, pointer, and string data, and invoke routines heavily. No floating-point or decimal arithmetic is involved.

**BENCHMARK SEGMENTS**

- **BUDDLESORT**—simple array sort using exchanges.
- **TOPSORT**—topological sort using linked lists.
- **SQUEEZE**—text compression and expansion.
- **UCIFY**—lexical analysis and substitution.
- **ACKERMANN**—famous doubly recursive function.

**BUDDLESORT**
The BUBBLESORT routine sorts an array of integers by exchanging adjacent elements. It exercises subscripting of array parameters, integer comparison, and loop control. The SPL compiler's precise code generation, along with FHP's compare-and-branch and looping instructions provide significant speed and size advantages.

**TOPSORT**
The TOPSORT test is an exercise in linked list manipulation. It takes a table of partial ordering relations, builds a large list structure, and traverses it to produce sorted output. TOPSORT emphasizes pointer manipulation and indirection. Most of the time is spent in a repeatedly called function that scans a linked list. Within the inner loop, the translation is one-for-one. No overhead instructions are present.

**SQUEEZE**
SQUEEZE takes a stream of characters, compresses null bytes to a single bit, and emits nine bits for other characters. A second routine reverses the process. FHP's ability to deal with bit-oriented data of arbitrary alignment and size gives it a significant speed advantage over traditional architectures for this type of system programming.
UCIFY

UCIFY performs text filtering, replacing words in a source text stream according to user-supplied substitution rules. The basic structure is that of a lexical analyzer for a Pascal-like language. It exercises primarily the character string handling capabilities of an architecture, but also works calling and up-level referencing.

ACKERMANN

Ackermann's Function is a famous function studied in computer science courses, defined as:

\[
A(0, n) = n + 1 \\
A(m, 0) = A(m-1, 1) \\
A(m, n) = A(m-1, A(m, n-1))
\]

The ACKERMANN benchmark tests call, return, and argument passing. FHP's extremely fast Neighborhood Call feature, coupled with SPRINT's Argument Base Cache for accelerated argument transmission, make this segment exceptionally fast.

<table>
<thead>
<tr>
<th>Benchmark Comparison</th>
<th>MV/8000 PL/I 1.10 Oct 81</th>
<th>FHP Demo Goals Mar 81</th>
<th>FHP SPL 13h Oct 81</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLESORT</td>
<td>7.68</td>
<td>4.70</td>
<td>4.67</td>
</tr>
<tr>
<td>TOPSORT</td>
<td>17.02</td>
<td>19.50</td>
<td>18.85</td>
</tr>
<tr>
<td>SQUEEZE</td>
<td>15.99</td>
<td>3.20</td>
<td>2.84</td>
</tr>
<tr>
<td>UCIFY</td>
<td>20.07</td>
<td>19.00</td>
<td>14.80</td>
</tr>
<tr>
<td>ACKERMANN</td>
<td>3.98</td>
<td>2.80</td>
<td>2.74</td>
</tr>
<tr>
<td>Total Time</td>
<td>64.74</td>
<td>49.20</td>
<td>43.90</td>
</tr>
<tr>
<td>Total Size</td>
<td>3672</td>
<td>2600</td>
<td>2586</td>
</tr>
</tbody>
</table>
OBJECTIVES

- Compare FHP fixed point performance with MV/8000, VAX 11/780, DEC 2060, and Amdahl 470
- Compare use of subscripts, pointers, and bit strings on FHP
- Demonstrate a substantial FHP architectural advantage
- Exhibit structured access to architectural capabilities

BENCHMARK DESCRIPTION

PUZZLE is a recursive program that fits three-dimensional pieces of various shapes into a box. It exercises a computer's control flow, fixed-point operations, and array accessing capabilities. It was developed at Stanford University to compare various architectures and processor technologies, and it has been run on at least fifteen super-minis and mainframes.

RESULTS

The PUZZLE benchmark is ideally suited to register optimization. For example, the MV/8000 PL/I compiler optimizes the inner loops by keeping the iteration variable and the array base in registers. In spite of this, FHP runs standard versions of PUZZLE significantly faster than the MV/8000 and other super-minis; its architectural accelerations are an adequate substitute for registers even under unfavorable conditions.

Using a nonstandard coding of the benchmark that exploits its architectural addressing advantages, FHP is faster than even very large conventional machines. The fast bit-string version of PUZZLE requires no changes to the data structures or basic control flow of the algorithm, and the code is simpler and more natural than the standard versions. Furthermore, use of bit strings is not restricted to SPL. The current FORTRAN compiler permits the use of inline functions to manipulate bit slices.

PUZZLE demonstrates FHP's addressing strength, which can be exploited in a number of ways. Operations on bit strings can be used directly to implement bit-map graphics, routing algorithms, and packing algorithms. Many graph and signal processing algorithms can be translated into manipulation of bit arrays. A vector instruction set could provide a dramatic speedup of tight FORTRAN loops, using the same addressing support. Finally, the addressing power of the architecture is largely responsible for the outstanding performance of COBOL and RTAM on FHP.

DETAILED ANALYSIS

PUZZLE is often run in several variants. We translated the original PASCAL text of the benchmark into the FHP Systems Programming Language (SPL). We have produced three variants:

Subscripts

The PASCAL version of the benchmark used subscripts, and so does our first variant. However, we reduced the original double subscripts to single subscripts by linearization, a modification of the loop control performed by many optimizing compilers (including MV/8000 PL/I).

Pointers

In the second variant, we eliminated subscripts altogether by accessing the array through a pointer, and incrementing the pointer each time through the loop. A pointer version of the benchmark has been run on many of the other machines.

Since most architectures do not directly support any addressing mode more complex than a based reference, pointers are usually
significantly faster than subscripts. However, on FHP, the subscript and pointer versions run at almost the same speed. This is important, because typical application programs use subscripts; pointers are more difficult to use and much more difficult to debug. FHP provides more support for the common case.

**Bit Strings**

The inner loops of this benchmark step through two arrays of bits, performing a logical operation at each step. Instead of sequencing through the arrays one entry at a time, we took the arrays as strings of bits and performed a single logical operation on the bit strings. Since the length of the strings varies, and since the base used in one array varies, this is almost impossible to do in most architectures. However, in SPL this variant is easier to code than the other variants, and on FHP it runs much faster.

**SUMMARY**

The PUZZLE benchmark was developed at Stanford to compare the fixed point performance of various architectures. It demonstrates that FHP is competitive with other super-minis in conventional fixed point performance, and that we optimize subscript calculation more effectively than conventional architectures. Also, by using bit-string operations, PUZZLE demonstrates that FHP can provide outstanding performance, far beyond its class, when its architectural strengths are exploited.
FEATURES

- Relational model
- Automatic enforcement of integrity constraints
- Traditional record-at-a-time processing
- Easy migration
- Automatic mediation of concurrent users
- High performance
- Powerful backup and recovery facilities
- Interactive query facility (IQF)

GENERAL DESCRIPTION

DBMS I is a full-scale data base management system for interactive and batch users who need solutions to storage management, concurrency control, and system performance problems.

Relational Model

DBMS I is a relational data base management system, giving programmers a natural way to view their data without being concerned over access paths to it. In addition to independence from access paths, DBMS I provides program and data independence by separating the description of the data base from the programs that process the data. This also allows data bases with the same view to have different internal storage formats geared to the needs of the data base users.

DBMS I provides powerful multirecord processing commands to manipulate sets of records in a single command. These commands allow you to extract columns out of a table, choose rows out of a table, and merge two tables based on matching field values.

Automatic Enforcement of Integrity Constraints

DBMS I provides several integrity constraints, which are automatically verified every time a record is stored, modified, or deleted. These integrity constraints help to ensure the validity of the user's data by putting restraints on what data is valid for some fields.

Record-At-A-Time Processing

In addition to the multirecord processing commands, DBMS I provides the usual record-at-a-time operations. Individual records may be stored, modified, deleted, and retrieved.

Easy Migration

Converting from a traditional file system to a data base management system has never been easy because of the necessary program rewriting. DBMS I makes this process much easier. Through the BIOS Logical Input/Output (LIO) interface, traditional files converted to DBMS I data bases can be accessed through programs as if they were either indexed-sequential files or sequential files. Other programs accessing the same data can be written to take advantage of full DBMSI functionality.

Automatic Mediation of Concurrent Users

DBMS I provides a comprehensive form of transaction processing that automatically mediates users' concurrent access to the data base. System throughput is increased since many users can share the data base at the same time.
time. Application programming is simplified since programs always see a consistent view of the database. Finally, DBMS I guarantees that any transaction will either complete successfully or have no effect on the database if the system crashes.

**High Performance**

The traditional drawback of relational database management systems is their poor performance. Thanks to the FHP architecture, DBMS I has overcome this problem (as the accompanying chart demonstrates). In fact, DBMS I performance exceeds that of traditional lower-level access methods.

**Backup and Recovery**

DBMS I features a powerful backup and recovery facility that assures database consistency and a minimum of downtime in case either the software or hardware should fail.

**Interactive Query Facility**

Users may access DBMS I databases interactively, without the need to write programs (see the IQF product brief).

**SUMMARY**

DBMS I is a full-scale, high performance, relational database product, automatically handling integrity and concurrency while providing a natural view of the database.
FEATURES

- Hardware Components:
  1) Desk-top microcomputer system (MPT-100)
  2) Diagnostic Processor Interface Card (DPI)
  3) Hardcopy device (optional)
  4) Modem (optional, for remote diagnostics)

- Software Components:
  1) Online Software:
     A) Operational Control Monitor (OCM)
     B) Firmware Design Verification tests (DVRs)
  2) Offline Software:
     A) Test Control Monitor (TCM)
     B) Fault Isolation Tests (FIs, logic tests)
     C) Modular Analyzers (functional tests)
     D) DTOS tests (IOP functional tests)
     E) IPL Hardcore tests

HARDWARE DESCRIPTION

The Diagnostic Processor (DP) is comprised of a stand-alone tabletop computer system (MPT-100) and the Diagnostic Processor Interface card with its hardware interface to SPRINT. The MPT-100 contains a power-up self test and is supported with a full set of DDOS diagnostics. The Diagnostic Processor Interface card (DPI) provides:

- A) scan-chain interface to all cards,
- B) margining control of power supply and system clocks,
- C) event monitoring logic, and
- D) DP to IOP/JP communication channels.

The DP is used both as the operator's console for online support and as the maintenance console for offline problem isolation.

SOFTWARE DESCRIPTION

Operational Control Monitor

The Operational Control Monitor (OCM) is the DP software which is responsible for Initial Program Load (IPL) for the operating system, and assisting with error handling and logging. OCM is a menu-driven monitor which does the following:

- Boots from a specified data channel device
- Handles operator console traffic (TTI/TTO)
- Halts the system (optional soft halt)
- Perform dumps of IOP/JP's memory, internal machine state, and DPI card register state
- Logs online errors
- Monitors for environmental system failures
- Executes Firmware Design Verification tests

COMPANY CONFIDENTIAL - DATA GENERAL CORPORATION
Supports a printer for all input/output communications

Provides remote console support

Design Verification Tests

The firmware Design Verification Tests (DVRs) are used to check the Job Processor’s (JP) product microcode. They are designed to test S-language implementation (FORTRAN, SPL, and COBOL S-OP’s) and elements of the FHP program architecture (e.g., Namespace). Each test executes upon operator initiative under OCM and replaces the operating system normally in control after the boot sequence.

Test Control Monitor

The Test Control Monitor (TCM) gives an operator access to the diagnostic system for maintenance and repair. This is used when the operating system is not active. TCM features are as follows:

- Hardware test execution and control
- Automatic test sequencing and isolation via scripts
- Error logging and display in full and limited formats
- Remote console support for Technical Assistance Center
- Machine access to hardware registers, main memory, power supplies, and system clocks

Optionally, the operator can run Fault Isolation Tests (FITs) individually or as part of a predetermined automatic test sequence, execute Modular Analyzers, and execute DTOS diagnostic programs.

Fault Isolation Tests

The Fault Isolation Tests (FITs) logically exercise the hardware, using the diagnostic scan-chain logic and test microcode to isolate a failure to a Field Replaceable Unit (FRU).

Modular Analyzers

The Modular Analyzers test each module (F-Unit, E-Unit, IOP, and Memory) by functionally exercising each logic net of the module. These tests use a signature analysis method (stimulus and response) to test for stuck-at conditions and verify hardware logic within each module. Stimuli (microcoded inputs) are scanned into the module under test, propagated (clocked) through the module, and selectively scanned out for comparison against a known “good” output database. Failures are isolated to the failing logic net or FRU where possible.

DTOS Diagnostic Tests

These tests verify the proper operation of the I/O Processor (IOP) as an ECLIPSE-compatible processor and all of its peripheral equipment.

IPL Hardcore Tests

These tests provide a quick check of the critical system components required for either IPLing the system or for running the diagnostic test system. These tests verify the proper operation of the DPI card, scan-chains, IOP microengine, and main memory. These tests are selected by the operator from the main menu as an option before running either TCM or OCM.

SUMMARY

The independent Diagnostic Processor increases system availability by reducing mean time to repair. ARM operational and test products are integrated on the Diagnostic Processor (DP) with DGC-supported MP/OS and Pascal. The DP plays a dual role. Online, it provides initialization, operating system initial program load, error monitoring/logging, execution of the firmware design verification tests, and an operator’s console. Offline, the Diagnostic Processor supports hardware logic and functional tests, with access to control and visibility through extensive hardware scan chains. The DP is responsible for margining voltage and system clocks to stress logic components.