```
8.      SHOW Δ[X]G[X]                                    SHOW Δ[X]G[X]
9.       ~G[X]                          ASSUMP
                                                         X 6,UI
10.      F[X]                           6,UI
                                                         X 3,UI
11.      F[X]→G[X]                      3,UI
                                                          10,11,MP
12.      G[X]                           10,11,MP
       SUB-PROOF AT LINE 8 COMPLETED.
     SUB-PROOF AT LINE 5 COMPLETED.
PROOF COMPLETED, CHECKING TERMINATED.
```

Notation:

Δ denotes Λ or ∀

∇ denotes V or ∃

Kalish-Montague Formal Proof

T201

1. ~~Show~~ $\Lambda x(Fx \rightarrow Gx) \rightarrow (\Lambda xFx \rightarrow \Lambda xGx)$
2. $\Lambda x(Fx \rightarrow Gx)$
3. ~~Show~~ $\Lambda xFx \rightarrow \Lambda xGx$
4. ΛxFx
5. ~~Show~~ ΛxGx
6. Fx — 4, UI
7. $Fx \rightarrow Gx$ — 2, UI
8. Gx — 6, 7, MP

$$\frac{\Lambda\alpha\phi}{\psi}\,,$$ Universal instantiation (UI)

$$\frac{\phi \rightarrow \psi \quad \phi}{\psi}$$ *Modus ponens* (MP)

A LANGUAGE MACHINE

Rodnay Zaks
Center for Research in Management Science
University of California, Berkeley

THE META–APL TIME-SHARING SYSTEM

META–APL is a multiprocessor time-sharing system developed at the University of California, Berkeley, for interactive real-time APL service. Its conceptual design has deliberately been kept simple and an expeptionally high performance to cost ratio has been obtained. Most of the functions traditionally performed by software have been pushed into the hardware-firmware execution unit. The core memory, although fast (750 nsec) is viewed by the processor strictly as an IO device.

The system was developed in two steps: completion of an APL processor and development of an operating-system processor, both processors being micro-programmed and

communicating through shared core. We will describe here the APL-machine. For further details about the total system, its use and possible extensions, see (1).

The APL machine may rightfully be viewed by the end-user as a dedicated hardware APL processor. Its design, however, embodies the integration of hardware, firmware and software concepts. Technological advances in the state of the art have made available micro-processors with a high major/minor cycle ratio (9 to 10) which are equipped with micro-instruction sets comparable in power to the conventional instruction sets, thus placing a premium on the number of instructions performed between core accesses. In fact, if this number if large enough (8 suffices in META–APL), the processor may never have to wait for core and will run at the speed of its internal registers (90 nsec in META–APL). This technological advance allows an interpreter for the first time to reside directly in control-storage and still provide very efficient language processing. A simple example will make this point clear: during the time it takes the processor to perform a (microprogrammed) floating multiply (20 usec), it could have walked down 167 levels of a binary decision tree or performed 237 logical or shift operations.

An APL interpreter being obviously needed to provide the time-sharing facilities and elaborate diagnostics, it was decided to implement directly the interpreter in the ROM (Read-Only-Memory) control storage of a micro-programmed processor.

THE LANGUAGE PROCESSOR: HARDWARE

Digital Scientific's META-4, a microprogrammed 16 bit-processor, was selected as a suitable host for the interpreter. It executes 32-bit micro-instructions in 90 nsec average time. Micro-programs are stored in up to 2K of ROM and this pratical limitation constituted an important constraint on the design of the system: the whole interpreter had to fit within 2,000 instructions of control storage, accentuating the original philosophy to keep the overall design simple. Core cycle is 750 ns proper, or 840 ns, accounting for propagation and interface delays.

The processor is organized around a simple 3-bus structure: data are obtained from the A-bus and B-bus and gated to a logical unit, followed by a shift unit. The result is then gated via the D-bus to the specified destination register. A typical 3-address instruction is shown below in assembly language.

MEMORY–ADDRESS–REG = REGISTER 7 + REGISTER 20 L8 MR;

- Registers 7 and 20 represent the origin registers on the A and B buses.
- + is the specified arithmetic operation.
- L8 (left shift eight positions) is the specified shift-unit operation.
- MR is an independent modifier-bit which wakes up the memory interface for the specified memory read operation.
- MEMORY–ADDRESS–REG receives the resulting data from the D-bus.

The machine is equipped with 32 logical register positions which have been filled with 28 hardware registers and a 64-word 40 nsec scratch pad.

To facilitate dynamic memory allocation, the processor has been equipped with a hardware map: 128 words x 12 bits. Each process thus has access to a 65K virtual memory, physical storage being allocated in 512 word-blocks, the page size. Care being taken to start map accessing at the same time as the priority level is being decoded, the overhead due to map insertion is nil in most cases.

THE LANGUAGE PROCESSOR: FIRMWARE

An APL statement, edited at one of the CRT terminals, is transformed by the "Translator" into a

token-string or "internal APL ." The Translator, resident on the Operating System Processor, then discards the external APL string. The one-to-one correspondence between external and internal APL insures that either string may be simply generated from its translator image. All tokens consist of a descriptor field, identifying the syntactic type of the token followed by a semantic field identifying its content (variable index number, function index number) or actually holding it (operator code or floating point number). All tokens are one word long, with the exception of numbers which are represented in a 2-word floating-point notation and vectors which have a 3-word header followed by the vector elements.

The META–APL interpreter processes such internal APL strings dynamically, incrementally and decrementally. Decremental program execution is made possible to the extent of a one-statement-backup and of a one-function level backup by deferred assignment of values to operand calls, the intermediate result being pushed on the Stack. To avoid too severe a storage loss, arrays of large size may not be backed up as this turns off the future.

The concept of process is associated with the entity in control of a processor during a computation. A process thus has a number of attributes such as: map or virtual space, virtual processor defined by the process capabilities or segment descriptor table. The correspondence being isomorphic, all characterizations belong to a common equivalence class that we shall call the process equivalence class. In practice, we shall characterize a process by a distinguished element of its class, the process segment of its virtual space. A process is brought into existence by a fork instruction in a mother process or by deliberate coupling of a user with the system, resulting in the creation of a master-process. We shall now examine the virtual space organization of a process.

The 65K virtual storage allocated to each process is partitioned in four segments:

1. Program strings: contains all internal APL strings in execution for all processes. Area 1 is common to all users and allows simple dynamic sharing of user programs and library APL functions.
2. System tables: common to all users – do the obvious thing.
3. Management Table (MT): a stack of address tables (OAT) followed by the Operator Push Down Lists (OPDL) at each level. Within each variable token, the identifier field constitutes an index number to the appropriate (global/local) OAT. The OAT entry points to the actual operand location within the stack segment.
4. Stack: all temporary results, literal arguments and unevaluated variables ("Operand Calls") get dynamically pushed down on or popped from the operand stack.

Stack and Management Table constitute the process segment.

In a first phase, the Interpreter operates a left to right parse of the tokenised APL line, pushing Operand Calls, numbers and vectors on the Stack and pushing the operators onto the OPDL. A function call is placed at the end of the previous OPDL and starts a new, local, OAT segment, followed by a new OPDL, for the function. When an APL execution delimiter is encountered, such as end-of–line, the Interpreter enters its right to left execution phase, applying operators from the OPDL to operands appearing on top of the Stack.

Naturally, the two top elements of the Stack as well as the top of the OPDL and all appropriate pointers are kept in hardware registers. Further discussion of the Stack management and dynamic array handling is not within the scope of this paper.

THE LANGUAGE PROCESSOR: SOFTWARE

Since the phsical size of the ROM limited the interpreter size to 2,000 instructions, it was not possible to implement in firmware all of the numerous and sometimes very complex APL operators. A basic spanning set, consisting of the 17 most frequently used operators was there-

fore included in the interpreter and all remaining operators were implemented as library APL functions, i.e., APL code using this primitive spanning set (2). The Translator is in charge of substituting a system-function call descriptor for this class of APL operators.

FIRMWARE DEVELOPMENT

All system firmware is to be developed in three phases:

Phase 1: simulation on the XDS 940 time-sharing system, with all the facilities inherent to a general purpose time-sharing system: file system, editor, on-line debugger, and assembly language.

Phase 2: simulation on META–META, a self-emulator for the hardware processor, residing in the processor itself. META–META fetches its insturctions from core and simulates by means of interpretive execution the operation of the corresponding ROM micro-instructions, thus allowing core to be used as a simulated control storage, with all the flexibility of read-write storage, but at an expense of a significant reduction in ex-exution speed. This simulator also provides an on-line debugger and assembly language. In addition, it is equipped with all the routines necessary to control the IO devices particular to the META–APL system (because of the generality requirement, no IO routines are available on the 940 simulator). This microprogram was naturally developed on the 940. The use of this second simulator may appear to be superfluous in view of the availability of the first one on the 940. Let us therefore indicate the important functions that it performs:

- insuring a thorough hardware check-out by early and continuous use of all of the machine micro-operations.
- providing continuous and on-site access to a simulator with complete IO facilities.
- allowing all programs, even test routines (other than CPU-test) to run on the machine interpretively from core for as long as possible before being finalized.

Phase 3: finalization of the debugged micro-code into ROM patterns. These patterns are easily field-alterable and final touch-ups are done directly on the ROM boards.

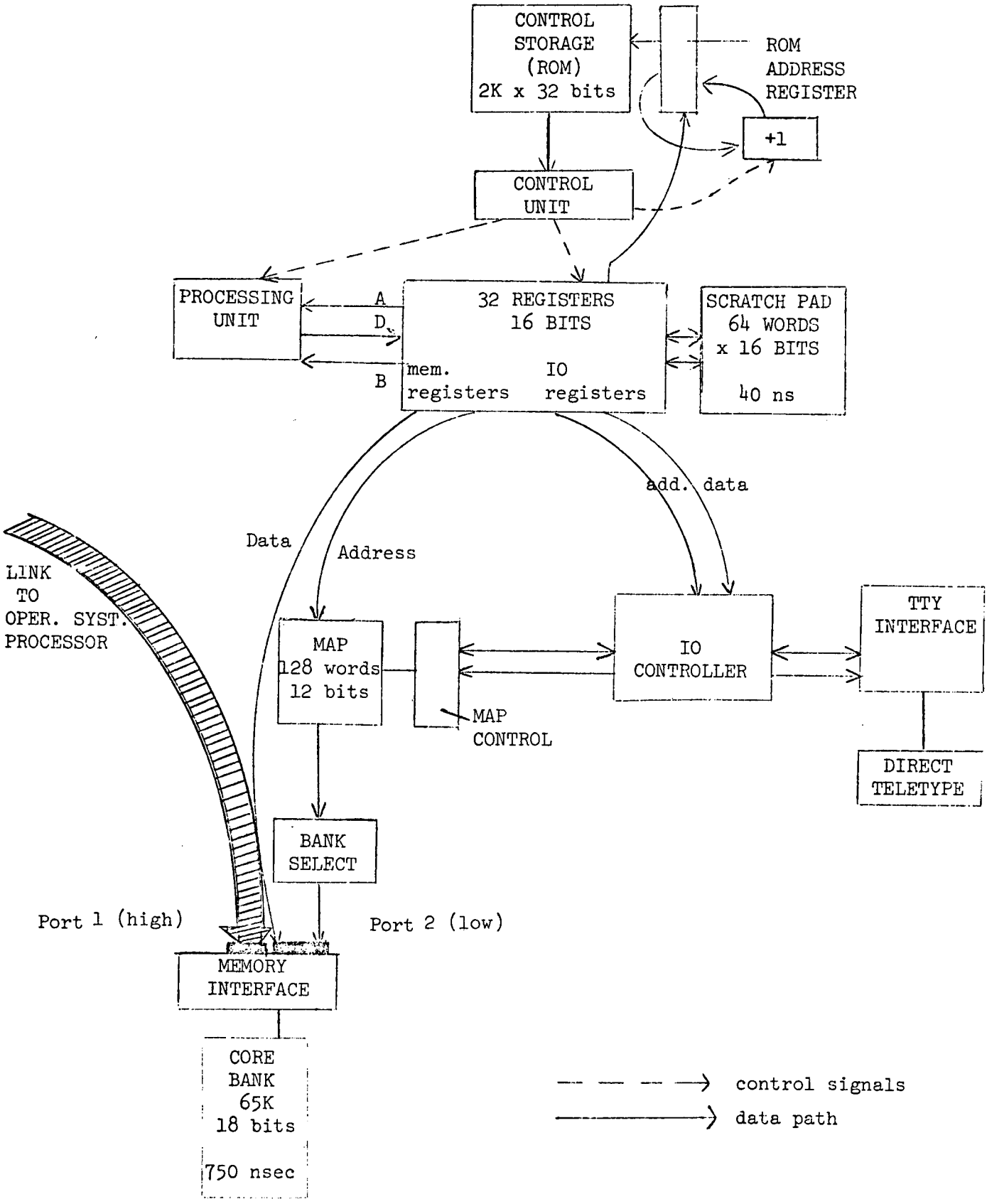
SYSTEM UTILIZATION

The META–APL system can service up to 64 simultaneous processes. APL has been extended to provide each user with the ability to create parallel processes running concurrently with the mother process and asynchronously. In a typical situation, one large program will activate 10 to 16 parallel controlling processes, each connected via a CRT terminal to the experimentee. These special processes have highest priority and run under a real-time constraint (response time less than 0.1 sec). Simultaneously, the system provides APL service at a lower priority for normal program development by other users.

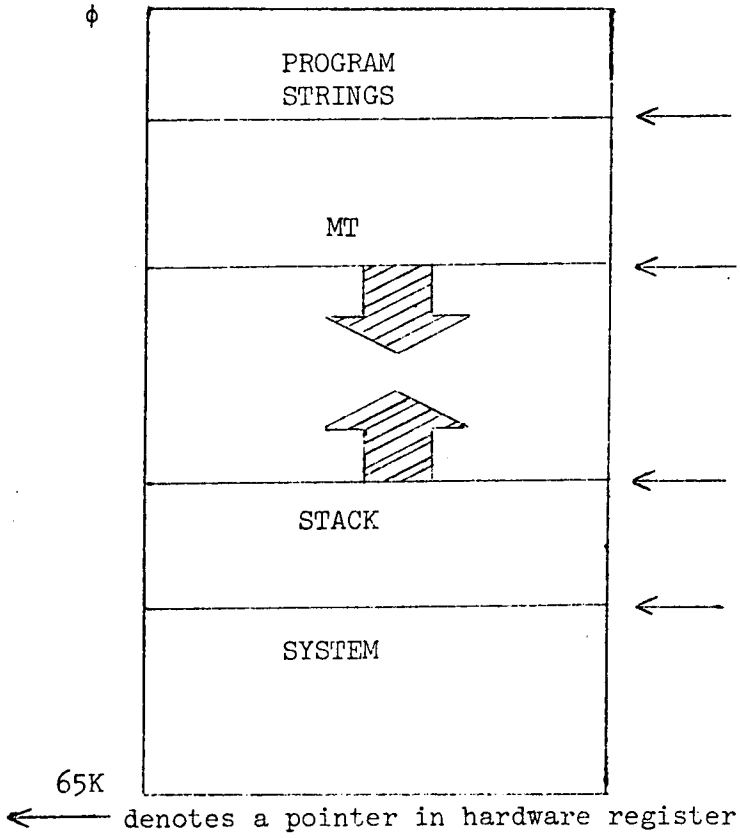
REFERENCES

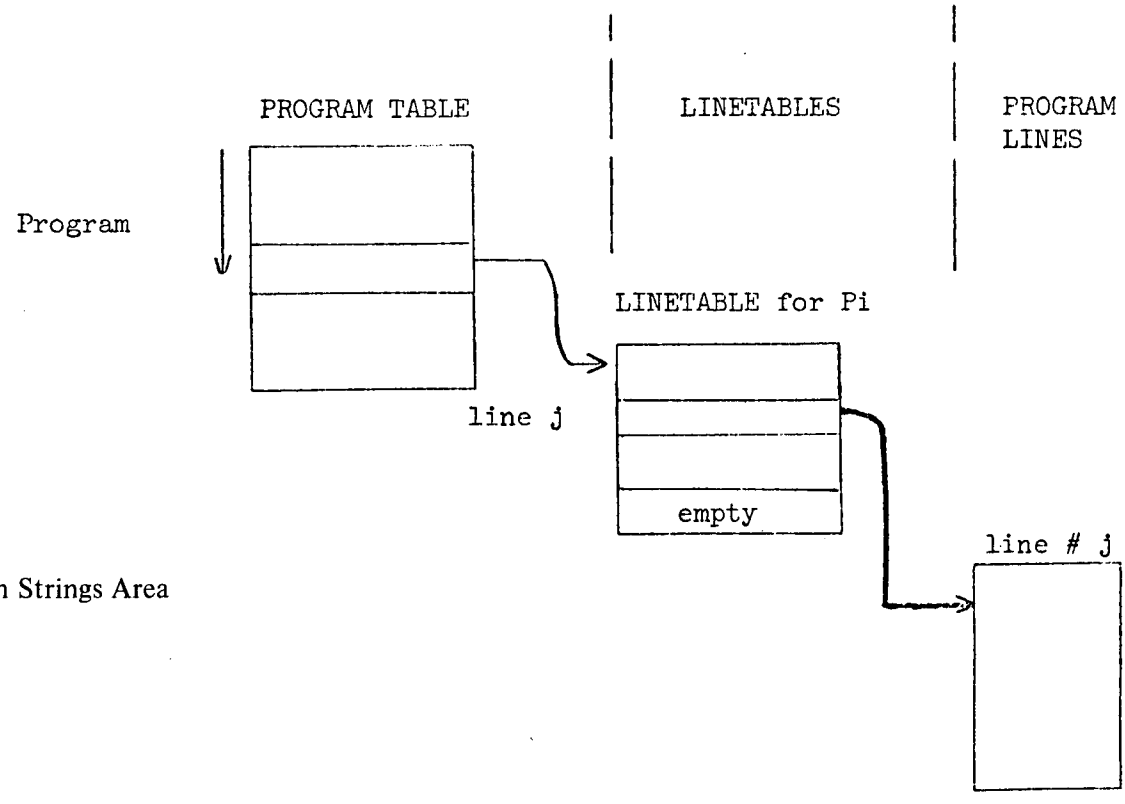
1. "A Firmware APL Time-Sharing System" Rodnay Zaks, Divid Steingart, Jeffrey Moore, AFIPS proceedings, SJCC 1971.
2. "Simulation of Some APL Operators" E.A. Stohr, CRMS Working Paper, February 1971, University of California, Berkeley.

The Language Processor



Process Segments





Program Strings Area