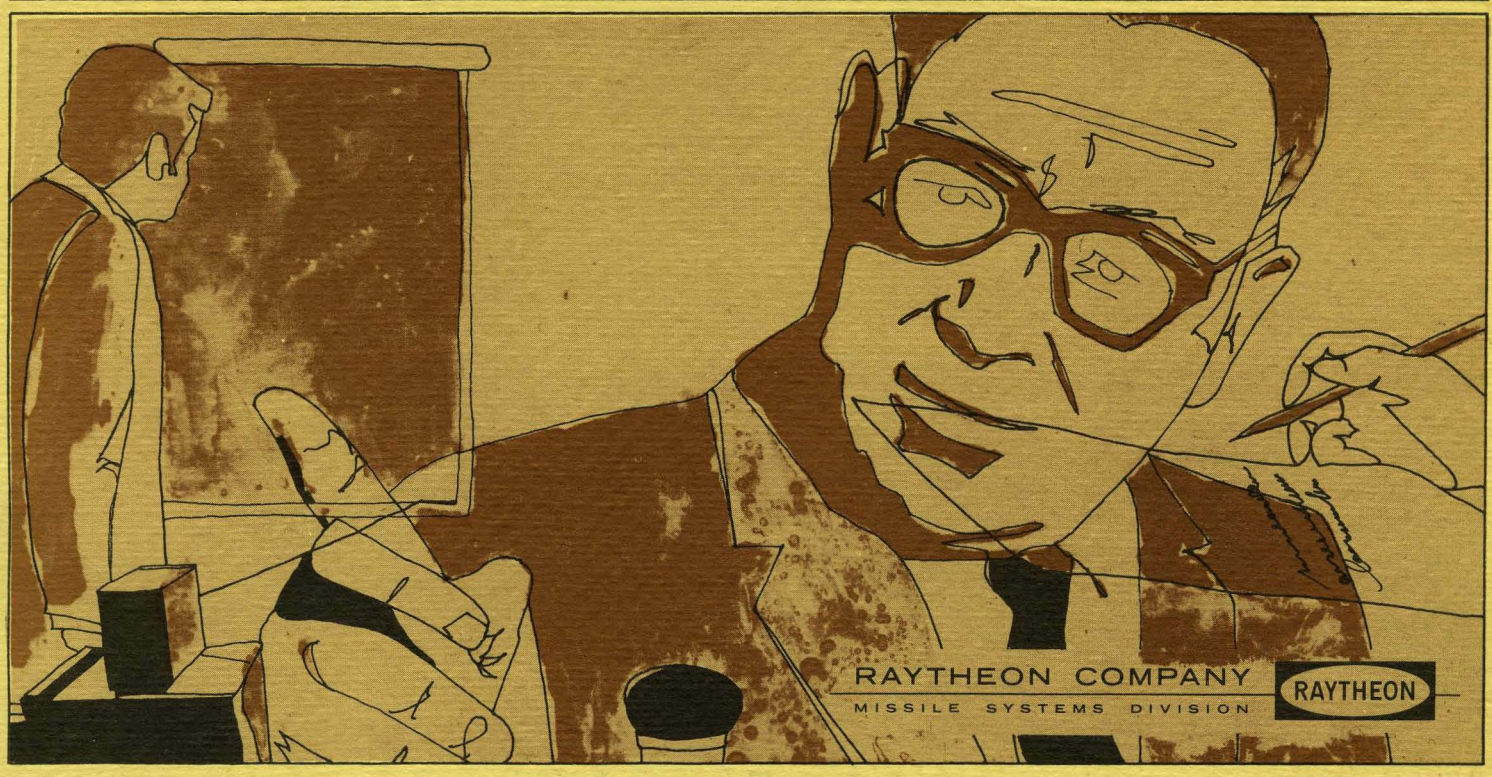
THE ALL APPLICATIONS DIGITAL COMPUTER

Stanley M. Nissen & Steven J. Wallach

SYMPOSIUM ON HIGH LEVEL LANGUAGE
COMPUTER ARCHITECTURE
College Park, Maryland
NOV. 7 & 8, 1973

P897



THE ALL APPLICATIONS DIGITAL COMPUTER*

by

Stanley M. Nissen

and

Steven J. Wallach

Raytheon Company
Crosby Drive
Bedford, Mass. 01730

1.0 Introduction

This paper describes the AADC Data Processing Element (DPE) developed and being built by Raytheon for the Navy to be used for multiplatform (air, land, seas, and underseas) applications in the 1978-1990 time frame. It was the intent of the designers of this computer to: achieve a Syntax-oriented internal architecture (Ref. 1) so that application programs written in a higher level language (HLL) would run efficiently, and to make use of a memory heiracy system to minimize the number of page faults generated by the virtual addressing mechanisms.

The HLL language used as a standard to model the internal computational structures was APL (Ref. 5). Studies (Ref. 2,3,4) convinced the designers that the most efficient machine code and most efficient program execution is achieved by the most compact representation of a problem in a HLL. This led to the choice of APL. Thus, the majority of APL operators are part of the instruction set of the DPE and the capability exists to directly execute operations on vectors and matrices without resorting to compiler generated loops.

1.1 System Overview

The AADC is a modular computer system. As the application warrants, system resources (DPE's, Channels, Signal Processor's (SPE), Random Access Main Memory (RAMM), etc.), are attached to the main data buses to achieve the desired configuration.

Figure 1 depicts a block diagram of a configuration of the AADC system. Here, the Data Processing Element normally executes all programs. The RAMM contains storage for variables and the BORAM (Block Oriented Random Access Memory) contains storage for procedure and constants. The typical addressing structure results in variables being accessed in RAMM. Consequently, no replacement of data pages in TM (Task Memory) results. Procedure and constants address BORAM. If the addressed BORAM structure is resident in TM, the virtual addressing logic supplies the base address of the resident page and no BORAM request is made. If the addressed BORAM structure is not resident, a read page request to BORAM is automatically generated. BORAM, typically, has a 2 microsecond access time and a 150 nano-second/word transfer rate. While waiting for the BORAM data, the location of the page to be replaced is derived by a micro-routine (see Section 5.0). Partitioning the data and procedure is this manner reduces the number of page faults detected in the course of a program's execution.

The DPE (Figure 2) is composed of two computers, the PMU (Program Management Unit) and AP (Arithmetic Processor). The PMU is a highly specialized 16 bit minicomputer. Its main functions are to fetch instructions, perform effective address calculations, support the virtual memory addressing mechanisms in micro-code, and take control of the PE when array operands are accessed.

The AP performs all full word arithmetic operations in floating point. During non-array operations the AP executes instructions and operands from the APQ (the fetch and execute portions of the PE are pipelined). When array operators require arithmetic processing the PMU takes control of the AP and issues commands to the AP to execute various control sequences (Section 4.0 discussed this process in more detail).

The remainder of this paper will discuss the salient features of the DPE. The features discussed are the:

- Data Insentive Arithmetic Structure
- Stack operated accumulator structure to directly execute expressions in infix notation (Parenthetical Control)
- Implementation of APL primitives

*This work was performed under contract number N62269-72-C-0023 for the Naval Air Development Center, Warminster, Pa.

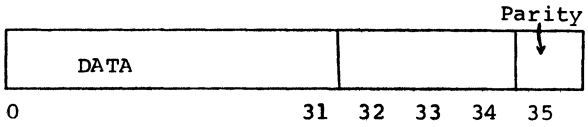
- Virtual Addressing Support
- Debugging & Performance Monitoring
- Pipeline Architecture

2.0 Data Insensitive Arithmetic Structure

A "Data Insensitive Arithmetic Structure" is achieved by appending a data tag (3 binary bits) to every word in memory and by providing a bit in every instruction word which controls the precision of the accumulator upon completion of an operation. The initial description of this structure discusses the use of the data tag.

Data Tag Description

Every word within memory has the following format:



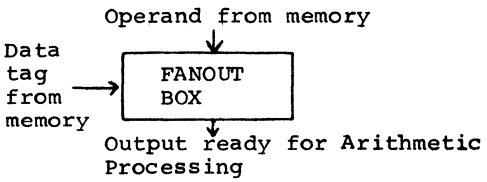
Memory Format

In the particular implementation, the 8 possible data tag meanings (2^3=8) are:

000 - LOGICAL
001 - DIMENSION WORD
010 - INTEGER
011 - LOGICAL
100 - SINGLE PRECISION
101 - DOUBLE PRECISION
110 - COMPLEX (REAL & IMAGINARY)
111 - SECOND WORD OF TWO

When a data word is fetched from memory, the data part of the word (BITS 0 through 31)are inputted to a logic structure referred to as a "FANOUT BOX".

The FANOUT BOX is depicted as follows:



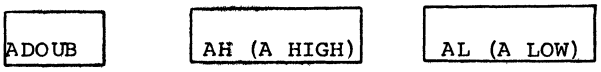
The FANOUT BOX converts memory formats into one common internal format. This internal format is "FLOATING POINT" (40 bits - 7 bit exponent, 1 bit sign, 32 bit mantissa. In the particular application the output of the FANOUT BOX is stored in the APQ (Arithmetic Processor Queue) along with appropriate control information to tell the AP how to manipulate the memory operand.

Controlling Precision of Result

Another part of the data insensitive structure is the insensitivity to the precision of the data (i.e. single precision or double precision). When an entry is made in the APQ, the data (as already described) has a common format. When the AP (arithmetic processor) fetches the operand from the APQ, the Data Tag is examined to ascertain the precision of the operand. Thus in the AP, all operands are the same type (floating point) and all are tagged as to their precision. Completing the data insensitive structure is the function of handling conversion from single to double and vice versa.

Within each AADC instruction bit 8 (Figure 3), the precision bit, is used to determine the precision of the result after the operation specified by the opcode is completed. This feature eliminates explicit instructions which would otherwise be needed. The conversion process functions as follows.

The AP accumulator is depicted as follows:



ACCUMULATOR PRECISION 1-Double 0-Single

When the accumulator is double precision and the result is to be single precision, the ADOUB flip flop is cleared to 0 and AL is cleared to all 0's. This results in the accumulator assuming the AP single precision format.

When the accumulator is single precision and the result is to become double precision, the ADOUB flop is set to 1 and AL is left unchanged. (Since the accumulator was single precision before the operation AL was all 0's to begin with. 0's in AL is exactly the value AL should be when converting single to double.) If a double precision result in generated from two single precision numbers, all precision can be preserved.

3.0 Parenthetical Control

Analysis of Naval programs indicated that 50% of the instructions executed were loads and stores. This instruction mix resulted from the storage and subsequent retrieval of intermediate results that were generated in the process of executing an arithmetic expression. To decrease program execution time and reduce memory requirements by eliminating unnecessary loads and stores, a multi-accumulator arithmetic structure was postulated.

Typical multi-accumulator approaches employ explicit addressing (e.g. IBM 360/370) or the addressing of the top two levels of a logically infinite stack (e.g. Burroughs B-6700). It was decided that a stack oriented accumulator approach was the most desirable. This decision coupled with some of the other architectural considerations (memory size and width, pipelining techniques,

indexing requirements) led to the development of a technique which executes expressions in an infix notation.

This technique, parenthetical control (Ref. 6), uses a 3 bit field (Bits 9-11, Fig. 3) of every arithmetic processor instruction to determine the sequence of execution of the instruction and consequently the control of the stack. The meanings of these three bits are as follows:

1. If all bits are 0, no parenthetical action takes place. This simply means that the instruction operates as it would in any conventional computer.

2. If all bits are 1, a begin parenthesis is specified. Analysis shows that it is never necessary to call for more than one beginning parentheses. When a begin appears, the present value of the accumulator (that register which holds arithmetic results already computed) is stored, with the operation code of the present instruction and certain control information. The operand associated with the present instruction becomes the new accumulator, and is treated as such by subsequent instructions.

3. If a number between 1 and 6 is specified, that many parentheses are closed. First, the AP performs the operation specified by the instruction operation code. However, the answer is now treated as an operand for the most recently deferred operation code and accumulator. This process constitutes the closing of one parentheses, and continues for each close specified (up to six).

The operation of parenthetical control will be illustrated via the following APL expression.

$$A \leftarrow (B+C) \times D$$

The AADC code to execute this expression is:

```
LOAD      D
x     )   C   DEFER THE MULTIPLY
+     (   B   EXECUTE THE ADD + POP
STORE     A   STORE THE ANSWER
```

Execution of the four instructions proceed as follows: the LOAD D loads the accumulator into the hardware stack, with the previous contents of the accumulator (D) and replaces the accumulator with C, AP control circuitry automatically detects a begin parenthesis on instructions fetched from the APQ. If the accumulator is double precision (as indicated by a flip flop), two pushes are necessary to store the accumulator. The +(B instruction results in two operations. First, the operand B is added to the contents of the accumulator (C), producing B+C, then one pop is executed. The AP detects the presence of a pending pop before fetching new instructions from the APQ. Honoring the pending pop results in the action of placing the accumulator B+C in the M register and bringing the top of the deferral stack into the accumulator with the opcode register getting the deferred opcode (x). Then the held opcode is executed producing (B+C) x D. The next instruction is now fetched from the APQ and the accumulator stored in A. Note that honoring a pending pop results in a register holding the pop count being decremented by one. A pop is pending if this register is not 0 or 7.

The above example illustrated the basic characteristics of parenthetical control. In this case all operators were arithmetic the following illustrates the ability to delay binding of operations to operands:

$$Z \leftarrow ((E=D) \wedge C>B) \times A$$

The AADC code is

```
LOAD   A
x    ) B
>      C
∧    ) D
=    (( E
STORE <Z>
```

Assume all operands are integers. The >C instruction produces a boolean value (integer 0 or 1). The ∧) D will eventually produce a boolean result. But, due to the begin parenthesis 1, the operand D is loaded into the accumulator with the deferred opcode being ∧. The = instruction results in a boolean result being produced for the ∧ instruction. The two pops result in the execution of the ∧ instruction and the X instruction.

Parenthetical control allows for direct execution of statements without the necessity to translation to Polish. In the AADC the translator must eliminate redundant pushes, insert a NOP instruction with closes if more than 6 parenthesis closes are necessary, and insert pushes where the HLL possesses operators which exhibit operator precedence.

4.0 Array Handling Features

One of the major features of the AADC is its array handling procedures, implemented entirely in hardware (Ref. 7).

When an AADC arithmetic instruction addresses an operand, it will encounter any one of 8 different types of data. Six of these describe the format of the data, which could span two words. A seventh type, complex data will be considered later. The eighth type is a dimension word (Figure 4). The dimension word acts as a descriptor for the array which has been accessed. The

instruction, having accessed the dimension word as an operand, will operate on the whole array.

Within the dimension word is a data type describing the form of the array which must be homogeneous with regard to number of words per operand. Additionally, either one or two dimensions can be specified, with one dimension being a vector and two a matrix. Each dimension can range up to 255, though the total physical space used by the array must be 256 or less including the dimension word.

In order to further support array operations, a set of APL primitives is implemented. These include inner product, outer product, reduction, take, drop, compression, expansion, lamination, decode, index generation (iota), ravel, reshape, reverse, rotate, monadic transpose, as well as non-APL operation, outer product reduction which includes as a subset membership and within limits search (Ref. 9).

Actual operation of these instructions occurs in a mode in which the arithmetic processor dedicates itself to servicing the array operands. The PMU also turns itself over to the array controller. For the duration of array operation, sequencing is performed within a ROM, which controls both PMU and AP operation. As main array storage is in TM, this Controller causes the PMU to provide the AP with appropriate operands at the proper time, initiating an arithmetic operation within the AP when these operands have been placed into the AP active registers.

Complex arithmetic is treated as a case of array arithmetic with an additional implied dimension of two. Thus, if two complex scalars operate on each other, array sequencing will be called upon. Complex matrixes can be processed with this arrangement.

Many operations involve rearrangement of array elements. These are performed exclusively within the PMU, though the AP remains halted pending instructions from the array controller or a return to normal operating mode.

Array mode is entered upon encountering either a dimensioned operand, a complex operand (can be a complex scalar), or an array operation code. At this time the array microprogram is entered, beginning with a fixed, uniform initialization sequence in which a page of Task Memory is set aside for the result of the operation with the scalar accumulator. This answer area becomes the accumulator of the end of the operation. During array mode, an extension to the index register scratch pad containing 16 registers is used to hold pointers to the various elements involved as well as dimensions of the accumulator and operand arrays.

Once the scratchpad pointers have been initialized, a particular sequence is entered, according to the operation code, in which length and domain checking is performed as in APL.

Having verified the computer's ability to operate on the accumulator with the operand, the operation itself is performed. Upon completion of the operation, a uniform sequence is used to return the machine to a common state in preparation for the next instruction.

A final facet of array operation is packing control. This function allows arrays to be stored with a homogeneous packing factor as either binary (32 1-bit numbers) Quaternary (16-2 bit numbers) hexidecimal, byte (4 8 bit numbers), half-word or full word (logical). These arrays are unpacked when they are operated upon as full word operands and optionally stored back as packed or fullword.

It should be noted, that as a result of the method used to bind operator to operand, procedures can be written that are data insensitive and structure insensitive.

5.0 Virtual Addressing Support

The AADC has provisions for a virtual addressing mechanism. Besides the "Normal" amount of hardware support for virtual addressing, 15 replacement algorithms are directly supported by hardware in the PMU.

When a page fault occurs, a microprogram sequence is entered. This sequence accesses hardware which determines the page in Task Memory to be replaced. Upon determining the replacement page, the kernel location of the page to be overlayed is accessed and the residency bit (bit 3, Figure 5) is reset to indicate that the page is no longer resident. The replacement algorithm chosen is under programmer control and is established by the STP (Set Task Parameters) instruction.

Prompting the inclusion of 15 replacement algorithms under program control was the desire to minimize the number of page faults. While it is not the intent of this paper to analyze the advantages of one replacement algorithm versus another, it is documented that no one algorithm can obtain the operational efficiencies of the optimal replacement algorithm, min, as postulated by Belady (Ref. 8). Thus, the programmer, knowing the sequence of execution of his program, can take advantage of the various beneficial characteristics of different algorithms in different program flows.

6.0 Debugging & Performance Monitoring

The AADC hardware provides two facilities which can be used for program debugging and performance monitoring. These features are referred to as the Instruction Trace Mechanism and Kernel Trap Mechanism.

Bit 34 of every instruction word (Figure 3) is referred to as the Trace Bit. When an instruction is fetched, the Trace Bit is examined. If the bit is 1, the instruction just fetched is executed and a trap to the Trace Bit Trap location is executed. The contents of the location trapped to are interpreted as an instruction, and subsequently executed. Normally, the instruction executed invokes a subroutine which can keep a running count on the number of times a particular type of instruction is dynamically being executed or output the value of the accumulator. This last feature directly parallels the Tracing mechanism of APL.

Bit 1 of every Kernel location (Figure 5) is the Kernel Trap bit. During the virtual address translation process this bit is examined. If it is 1, the instruction in the process of being executed is allowed to finish and then a trap to the Kernel Trap location is execution. The instruction of this location is then executed. This Kernel Trap is enabled for both reads and writes. This facility complements the instruction trace mechanism, and would be used to gather statistics and perform traces on operand accesses.

7.0 Pipeline Architecture

The DPE in order to meet the operational requirement of 2 MIPS (million instruction per second, based on a mix of seven adds to 3 multiply instructions), overlaps data requests to RAMM with instruction fetching, and provides up to 16 levels of instruction look ahead. The facilities which provide these capabilities are the APQ and the AADC channel.

The APQ is a FIFO stack which holds operands and instructions awaiting execution by the AP. The APQ is configured in a manner that allows the queue to act as two independent stacks, for the control part (bits 0-11 of an instruction), and the operand addressed. The AP executes all instructions from the APQ. The PMU loads the control part of the APQ every 450 nanoseconds.

The AADC channel is the means by which all internal inter resource communications occurs. The channel utilizes a non-dedicated time slotted bus. A distributed equal priority bus controller is used. In order to allow communications to operate efficiently, resources are always available to receive information. Queues in the input and output parts of the channel facilitate this "always available" attribute.

The pipeline functions as follows. The PMU fetches an instruction, accesses the kernel associated with the addressed page, performs a security check, and then, loads the control part of the APQ and issues a read word request to RAMM. This sequence is repeated every 450 nanoseconds. In parallel to these actions, the channel is transmitting the read request to RAMM. The RAMM reads the addressed word and transmits the data back to the originating resource. The data once it is available to be placed into its proper space in the operand part of the APQ, is stored simultaneous with the fetch of new instructions.

There are some other interesting facets of this pipelined structure. All index registers are located in the PMU. The PMU has its own instruction set, including adds, subtracts, logicals, multiply, divide, etc) that manipulate 16 bit operands. This physical and computational distinction between index registers and accumulators allows for programs to be written which run concurrently in the PMU and AP, thereby achieving a large degree of parallelism. Another advantage achieved by this structure is the elimination of the ambiguous condition of the use of an index register before the index register is loaded. In instruction look ahead machines with accumulators that function as index registers or where index registers are located in the arithmetic section of the processor, instructions which use an index register must first determine if any instruction queued and not executed modifies the index register to be used (e.g. the Reservation and Control Scoreboard actions of the CDC 6600). The AADC approach eliminates this control and results in a greater degree of parallelism.

Acknowledgement

The authors wish to thank: Ronald Enter of Navair, Carl Mattes of NADC, and Warren Loper of NELC for their encouragement and assistance in the design and development phase of this project. Also, we wish to thank Alan Deerfield of Raytheon who provided the authors with countless suggestions for improvement during the course of the project.

References

1. Chu, Y., "Introducing the High-Level-Language Computer Architecture", Technical Report TR-227, Computer Science Center, University of Maryland, February 1973.

2. Wallach, S., et al, "Second Interim Report - 2 MIPS Processing Elements", BR-6399, Raytheon Corporation, Bedford Laboratories, April 20, 1971.

3. "Analysis of the CMS-2 Programming Language", BR6704, Raytheon Corporation, Bedford Laboratories, December 1, 1971.

4. "Proposal for Aerospace Higher Order Language Study", BR-6296, Raytheon Corporation, Bedford Laboratories, February 26, 1971.

5. "APL/360 Primer", GH20-0689, IBM Corp., January 1970.

6. Deerfield, A., "Instruction Deferral Sequencing Mechanism", Proceeding of the Symposium on Programming and Machine Organization, IEEE Computer Society, Mid Eastern & New Jersey Coast Chapters, April 27, 1971.

7. Nissen, S. M., and Wallach, S. J., "An APL Microprogramming Structure", Sixth Workshop on Microprogramming, College Park Maryland, September 1973.

8. Belady, L. A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," IBM Systems Journal, Vol. 5, No. 2, 1966.

9. Outer Product Reduction performs the operation OP2/B o.OP1, A in one control sequence where A is the accumulator and B is the operand fetched from memory. If OP_2 is specified as $\vee$ (or) and OP_1 as = (equal), the membership operation results. By specifying the proper compare opcodes, a high low search is performed.

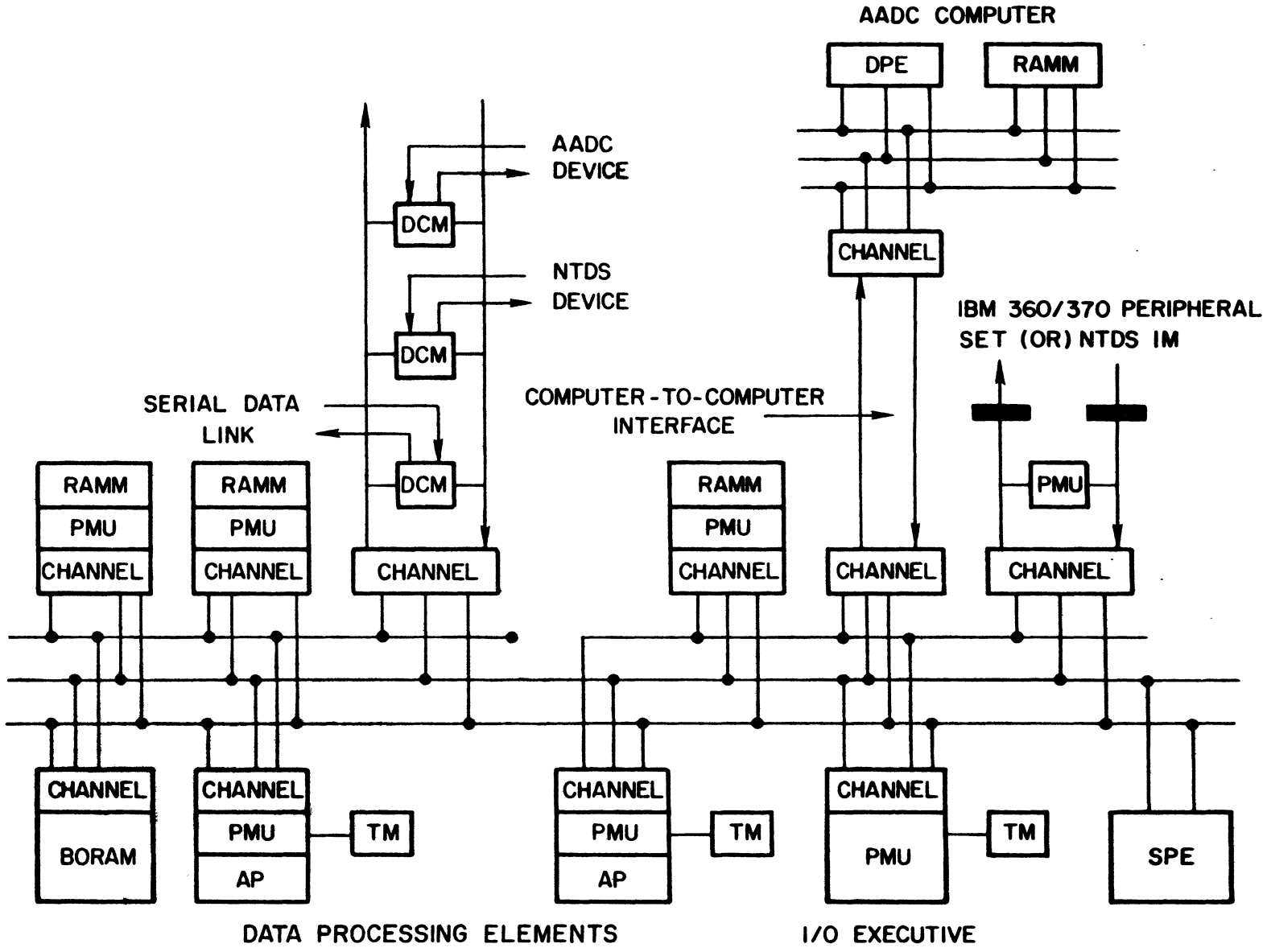


Figure 1 AADC System Configuration

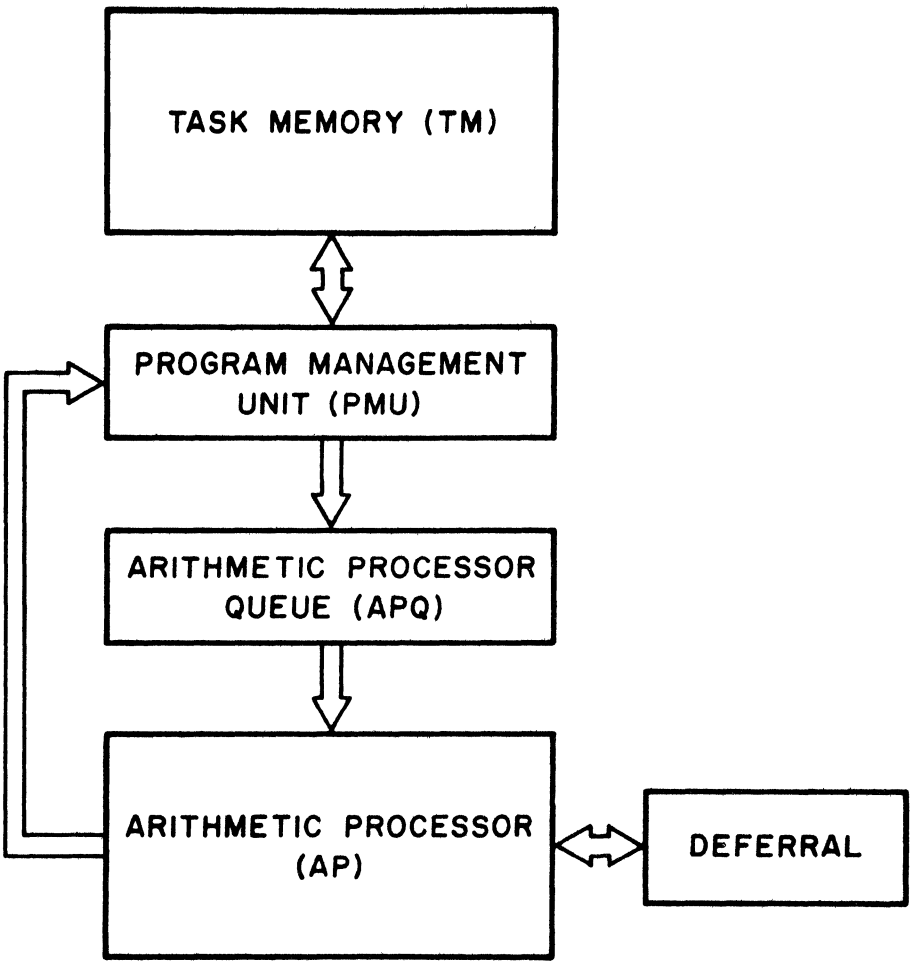


Figure 2 AADC Data Processing Element

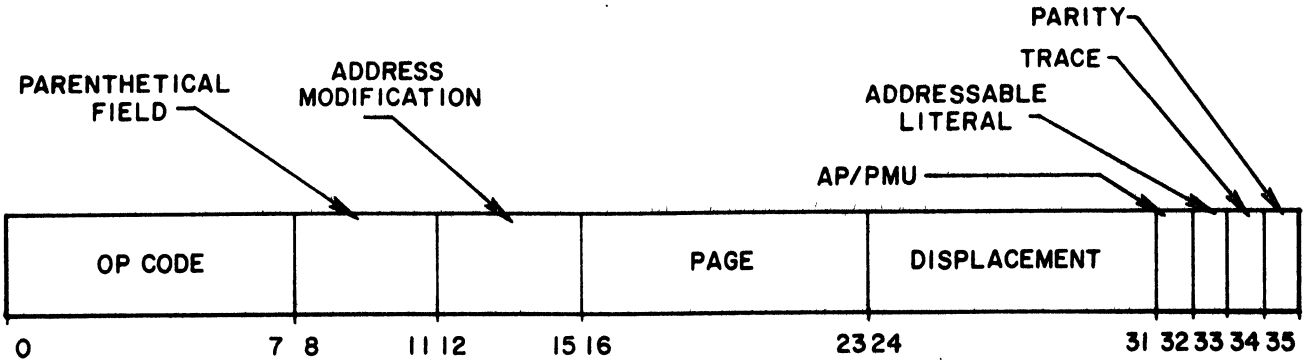


Figure 3 Instruction Format

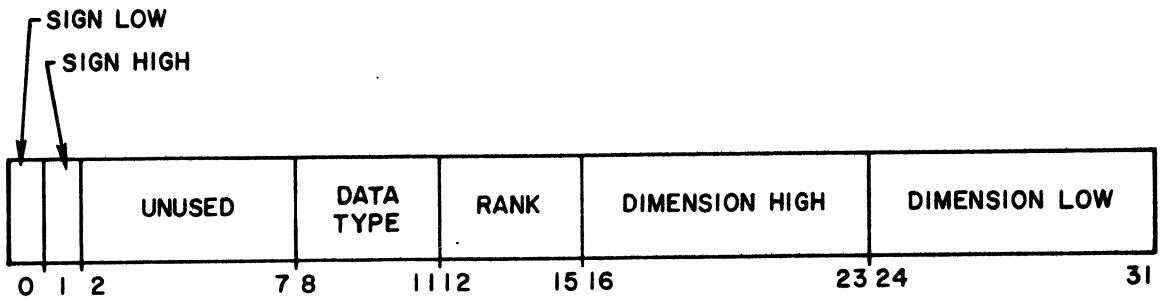


Figure 4 Dimension Word Format

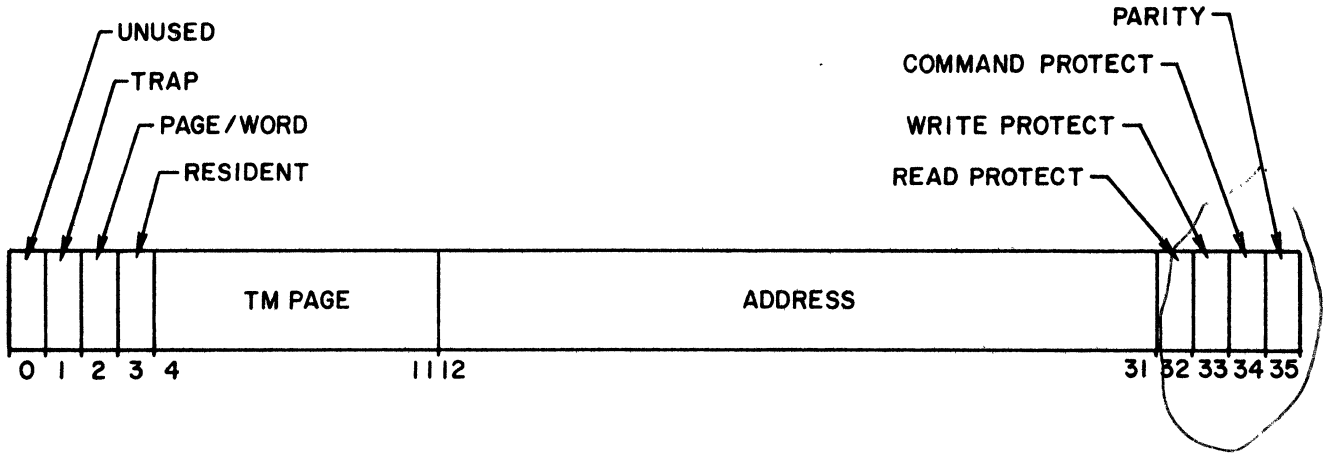


Figure 5 Kernel Word Format

REPRINTS AVAILABLE

S. HUMPHREY
RAYTHEON COMPANY
MISSILE SYSTEMS DIVISION
Bedford, Massachusetts