NAVAL ELECTRONICS LABORATORY CENTER
271 CATALINA BOULEVARD
SAN DIEGO, CALIFORNIA 92152
714-225-6011
AUTOVON 952-1011

IN REPLY REFER TO:

10462
Ser 5000-39
2 APR 1971

From: Commander, Naval Electronics Laboratory Center, San Diego, California 92152
To: Commander, Naval Air Systems Command, Washington, D. C. 20360 (Mr. R. Entner, AIR-5333F4)

Subj: Advanced Avionics Digital Computer (AADC) Arithmetic and Control Unit Design Study by Raytheon; comments on

Encl: Comments on the Raytheon Arithmetic and Control Unit Design for AADC

1. Enclosure (1) is forwarded for consideration.

M. D. VAN ORDEN

A.E. BEUTEL
By direction

Copy to:
NAVELEX, ELEX-0544 w/encl
NADC, Paul Brady, AEDC, w/encl
NRL, Dr. Bruce Wald, Code 5030, w/encl
NRL, J. Kallander, Code 5034, w/encl
FCPCP, W. D. Blair, Code 63, w/encl

29 March 1971

COMMENTS ON THE RAYTHEON ARITHMETIC AND CONTROL UNIT DESIGN FOR AADC

1. At the AADC program review at NADC on 18 February 1971, Mr. Ronald S. Entner (AIR-5333F4) requested NELC to review and report on the Raytheon design of the Arithmetic and Control Unit for AADC (Deerfield, Dapkey, Nissen, and Tannenbaum, 1970). This review was to be performed in the light of available literature, especially that on High-Order Language-Inspired computing machine designs. Articles reviewed included those referenced in the bibliography. Mr. Entner specifically requested that we answer two questions:

(1) What are some additional instructions, within the general architecture of the machine proposed in the Raytheon report, which would benefit compilation?

(2) What is NELC's evaluation of the impact of Raytheon's parenthetical control and general deferral mechanism on the compilation process?

2. In response to the first question, an example of the kind of hardware needed to substantially improve compiler throughput would be a hardware scanner. "A scanner, in programmer's nomenclature, is a means for recognizing the character boundaries of the next message to be fed to the compiler. This next message is called a token, and a token is an identifier, a reserved word, a special character, a number, or a character string. The means programmers have used to find the next message, or token, is a programmed routine to scan the sequence of characters waiting to be compiled in order to identify and mark these boundaries. The characters are passed on as a group to the main part of the compiler. Historically, this scanner routine is the slowest part of compilers. Hardware assistance in this area, again based on the way the programming language is constructed, could provide significant reductions to the programming task" (McKeeman, 1967). However, if one considers the trade-offs relative to hardware and cost, it is our recommendation that there is much more need and benefit in increasing the AADC input/output channel control capability to at least the level found in present day computers such as IBM System/360 than there is in adding a hardware scanner in the arithmetic and control unit of a processing element of AADC.

3. Additional instructions within the general architecture of the Raytheon machine could include a store multiple from the combined accumulator and deferred operator stack (a total of 40 bits wide by 16 deep) and a complementary load multiple.

4. One requirement to be expected in some environments in which a market for the multi-platform AADC exists is hardware design for security. Some recent published work in this area includes Lampson (1969) and Skatrud (1969). For security reasons, there needs to be a Supervisor Call instruction which transfers control from program modules to Master Executive Control modules by interrupt rather than by branch. Distinction of program state from supervisor state prevents program modules from exceeding their authority. Even for debugged programs in an avionics environment, this would provide one means of

detecting some types of hardware failures. Implementation of the distinction requires machine state manipulation instructions which are themselves "privileged" to be executed only in Master Executive Control (MEC) or "supervisor" mode which is automatically entered on Supervisor Call Interrupt. Further, block and even single word memory protection is useful, especially in RAMM, which is shared, but also even in Task Memory (TM). For diagnostic purposes, it is useful, when debugging a program to be able to request to be interrupted to one's own subroutine, specified in some way, whenever a specified word (or one of a list of words) suffers surreptitious alteration because of the bug. Such hardware facility, together with certain other such interrupts would provide the same diagnostics at full speed which currently are available only at 1/20 (or so) speed in TRACE modes which always seem to get implemented in software by those who recognize the need. Software TRACE packages always suffer an inability to handle time-dependent bugs. This hardware TRACE would not be so handicapped.

5 Interpretive execution of program modules can implement a language incapable of directly violating security, since the program only makes requests which are granted (only if appropriate) by the interpreter. Since software interpreters are slow, hence expensive, hardware or firmware (Opler, 1967) implementation of the interpreter should be studied to evaluate cost competitiveness. The same goes for the run-time libraries associated with the usual compile and execute languages (such as the built-in functions of PL/I).

6 Implementation of a "management by exception" philisophy should extend, for example, to input/output and clock supervision. Polling and the attendant processing element over-involvement can be replaced by contention programming responding to interrupts which occur only when actions must be enqueued or may be dequeued. To this end, input/output should be designed based on channel command lists which can be located in the same block as the data area, hence separable from the program module area which need not be resident for the duration of the I/O. Furthermore, chained I/O commands and chained data reduce the need for constant involvement by the processing element in the program module being serviced. Execution of the channel command lists can be by the same processing element in MEC mode, but under program module protection key, or by any other (say, dedicated) P.E. or preferably by hardware, but the effect should be retained.

7 Raytheon claims that, in general, their deferral mechanism allows programs to be executed as written. Raytheon's general deferral mechanism is an approach to the design of stack-oriented computing machines such as the Burroughs B5000. In the case of the B5000, "while it would be quite feasible to construct a machine to directly interpret ALGOL expressions having suitably restricted identifiers, it was decided, in view of the simplicity of the transformation, to use the Polish form" (Barton, 1961). Anderson's machine concept, on the other hand, would have provided for the direct execution of ALGOL. "Without the A and O stacks (for identifiers and operators respectively), some kind of pre-execution translation would be necessary" (Anderson, 1961).

In contrast with the direct execution of ALGOL, as proposed by Anderson, or the direct execution of Polish postfixed strings, as in the B5000, the Raytheon machine provides direct execution of full word instructions with explicit "parenthetical" stack control. The compiler must generate this parenthesis control by an algorithm which (1) only may be as trivial as the translation from classical algebra to Polish Postfix, and (2) is not yet so classical and well understood. While the B5000 stack could be up to 1024 words, the Raytheon stack is only 16 deep. Even if Raytheon's claim of execution of programs as written were valid, this would imply no, or only local, optimization. Iverson (1964) has suggested methods by which compilers should also perform global optimization on programs written in a suitable language. This in itself suggests a goal for CMS-3.

8 Even if such global optimization were not implemented, translation to Polish string would not be a significant portion of the compiler.

9 The process of compilation for interpretive systems has been one of translating a program model from a human oriented description to a control string, executed interpretively by a run-time emulation of someone's idea of an "ideal machine." With AADC, the capability to micro-program the "ideal machine" assumed for a language would exploit this approach to compilation with a recognition and representation of the program model and generation of the control strings for the interpretive machine associated with the source language, be it problem-oriented, such as POSE, STRESS, and MIDAS, or procedure-oriented such as FORTRAN (Bashkow, Sasson and Kronfeld, 1967; Melbourne and Pugmire, 1965), PL/1 (Sugimoto, 1969; Wortman, 1970), ALGOL (Randall and Russell, 1964), EULER (Weber, 1967) and APL (Abrams, 1970, Bingham, 1970). Economical use of a microprogram control store will include microprogrammed implementation of those functions most frequently performed in completing some collection of tasks. Not only generic functions which call no other functions, but also higher level, but heavily used functions should be so implemented. Therefore, the microprogram control should include some kind of closed subroutine capability allowing a particular control sequence to exist only once in the control store, but be invoked in execution of more than one instruction.

10 To exploit truly human-oriented and problem-oriented programming, it would even be desirable to be able to command in some way "make yourself an AEW machine" at one moment and "make yourself a missile release machine" at another. The appellative "stored logic computers" has been applied to a class of such computers. Of course the service of changing machine architecture would be a privileged one performable only in supervisor mode.

11 McKeeman (1967) commented on the impact of the multiplicity of arithmetic formats requiring different registers or in the Raytheon case, different modes of operation. The stack-oriented machine providing explicit program control of arithmetic format required three times as much code for this part of the compiler as did the machine with the clean, simple and consistent format accessible only through its almost pure stack structure.

- Diff data types require more code in compiler
- If machine provides many diff ways of doing same thing. Confusing compile e.g. DIVIDE DIVIDE & STORE

12 We affirm that different "types" of data must be supported: not only real (floating point and at least the effect of integer arithmetic), complex, vector and matrix, but also queue (both First In-First Out and Last In-First Out), list, tree, string, graph, ring, and plex data structures. The concept of plex programming is a generalization of the more common "list processing" (whose many forms are particular sub-cases) in which the atomic units of problem modeling are elements with any number of component properties. A principle advantage of plex programming is that the inclusion of a free storage or available space system enables elements to be created dynamically as needed from a pool of unused storage, so that it is not necessary to allocate fixed amounts of storage for various purposes beforehand. Ross (1967) describes over fifty procedures supporting free storage handling which could be regarded as candidates for implementation either in hardware or firmware.

13 We recommend the implementation of "Typed Data"; that is, data whose type identification is stored along with it and controls operation by the processor on it. Apparently, one would sometimes need to command format conversions (explicitly or implicitly) to control the format of a result to be stored. But references could be totally controlled by the stored type. The distinction between executable instructions and data would be a natural outgrowth. Identification (by a bit) of the "type byte" could provide hardware protection against attempting to execute data since any instruction would have to begin with an instruction type-byte. "Typed Data" is a logical fulfillment at the hardware level of a long needed unity only recently appearing even in software as "Uniform Referents" (Ross, 1971).

14 The virtual memory scheme proposed by NRL may impact the addressing scheme to such an extent that variable length instructions are required. Furthermore, hardware or firmware implementation of a large instruction repertoire representing heavily used functions of a run-time library or of an interpreter could run to more than 255 operations leading to a variable length operator.

15 To use the deferral mechanism to load the address of 'A' with a deferred operation to store result from next accumulator in the stack into 'A' seems a waste easily avoided at compile time. There is another consideration brought about by the suggestion by the Raytheon designers that, in general, expressions can be evaluated in the order in which they are written (from left to right) in the High Order Language. If this means "as a general, i.e., universal rule with diverse particular interpretations" it is false. If it means "most of the time, but there are exceptions" the burden is on the compiler and on the run-time environment to handle the exceptions. Actually, one must not ignore the possibility that an "accumulator stack full" condition may arise (Carlson, 1963). When it does, it must be treated (preferably as an interrupt) by using core storage as an extension of the stack. For example, the present stack can be dumped to storage; flag set, and a new start from the top of the empty stack made. When an attempt is made to retrieve from an empty stack (another kind of interrupt), the flag indicates a required

restoration from core to the stack. It is possible that an external function invoked within an expression will, temporarily in the course of its execution, lengthen the stack by an amount not visible during compilation of the module containing the expression but not the function. This is because the function may be a pre-compiled one invoked from a library or may not be compiled until some later time (hours or days). This is to be expected especially in the building of large systems which necessarily are modular in concept, design and implementation. For this reason, it will probably remain necessary to evaluate in the following order: first, arguments of functions; second, those functions; and finally the expression containing those functions. All this should be done with a standardized and general (universal) subroutine call and return interface, hopefully strongly supported by machine design.

16 Husson (1970) discusses some advantages and disadvantages of microprogramming. Even the Raytheon report mentions "programmable" instructions, "to be defined as required." One important advantage is that the order code need not be finalized until late in the design cycle. More importantly, one can make a processor a slave machine to perform a given task in the most efficient way. In addition, during a period of transition from prior computer such as the AN/UYK-7, the prior computer can be emulated. This is a capability that is of particular concern to the Marine Corps. With a selectable or changeable instruction repertoire, one can select the system architecture best suited to the immediate task to be performed. For these reasons, not merely instruction execution statistics, as in the SCI report (SCI, 1970) but rather generic function or "primitive" execution statistics must be gathered and studied. Since "the past is prologue" run-time statistics-gathering facilities will be useful in continued tailoring of the machine design to support best those services most required from the system. The single most important benefit derived from the above approach would be the advantage of an extended useful life throughout the time frame 1975-1985 intended for AADC.

07 Among the conclusions of the Raytheon report, one heartily endorsed is that recommending a complete analysis of extended CMS-2 and the full range of its applications, and incorporation of primitives of the language as instructions of the machine.

BIBLIOGRAPHY

1. Abrams, Phillip S., "An APL Machine", Stanford Electronics Laboratories Report SN-SEL-70-017, February, 1970.

2. Anderson, James P., "A Computer for Direct Execution of Algorithmic Languages", Proceedings of the Eastern Joint Computer Conference, pp. 184-193, 1961.

3. Barton, Robert S., "Program Structures and the Organization of Computers", Presented at the University of Michigan Engineering Summer Conference on Advanced Automatic Programming, 1963.

4. Bashkow, Theodore R., Sasson, Azra, and Kronfeld, Arnold, "System Design of a Fortran Machine", IEEE Transactions on Electronic Computers, Vol. EC-16, No. 4, pp. 485-499, August, 1967.

5. Bingham, H. W., "The BD: MACHINE, An APL Model for Micro-Instruction Execution in Interpreter Based Systems", Burroughs Corporation Report TR-70-3, December, 1970. (Propriortary)

6. Carlson, C. B., "The Mechanization of a Push-Down Stack", AFIPS Conference Proceedings, Vol. 24, pp. 243-250, Fall Joint Computer Conference, 1963.

7. Deerfield, A., Dapkey, B., Nissen, S., and Tannenbaum, B., "AADC (Advanced Avionics Digital Computer) Arithmetic and Control Functional Block Diagram Design Analytical Study", Raytheon Report No. BR-6154, December, 1970.

8. Flynn, M. M., and MacLaren, M. D., "Microprogramming Revisited", Proceedings of the ACM 22nd National Conference, p 457, 1967.

9. Husson, Samir, S., Microprogramming, "Principles and Practices", Prentice-Hall, 1970.

10. Iverson, Kenneth E., "Formalism in Programming Languages", Communications of the ACM, Vol. 7, No. 3, pp. 80-88, February, 1964.

11. Lampson, B. W., "Dynamic Protection Structures", AFIPS Conference Proceedings, Vol. 35, pp. 27-38, Fall Joint Computer Conference, 1969.

12. McKeeman, W. M., "Language Directed Computer Design", AFIPS Conference Proceedings, Vol. 31, pp. 413-417, Fall Joint Computer Computer Conference, 1967.

13. Melbourne, A. J., and Pugmire, J. M., "A Small Computer for the Direct Processing of Fortran Statements", The Computer Journal, Vol. 8, pp. 24-28, April, 1965.

14. Opler, Asher, "Fourth-Generation Software", Datamation, January, 1967.

BIBLIOGRAPHY (Cont'd)

15. Randall, B., and Russell, L. J., ALGOL 60 Implementation, Academic Press, 1964.

16. Rosin, Robert F., "Contemporary Concepts of Microprogramming and Emulation", Computer Surveys, Vol. 1, No. 4., pp. 197-212, December, 1969.

17. Ross, D. T., "AED Free Storage Package", Communications of the ACM, Vol. 10, No. 8, pp. 481-492, August, 1967.

18. Ross, D. T., "Uniform Referents: An Essential Property of a Software Engineering Language", Software Engineering, Academic Press, pp. 91-101, 1971

19. SCI, "Report on the Determination and Specification of the Preliminary Instruction Repertoire for the Advanced Avionics Digital Computer", Systems Consultants, Inc., February 27, 1970.

20. Skatrud, R. O., "A Consideration of the Application of Cryptographic Techniques to Data Processing", AFIPS Conference Proceedings, Vol. 35, pp. 111-117, Fall Joint Computer Conference, 1969

21. Sugimoto, M., "PL/1 Reducer and Direct Processor", Proceedings of the 24th National Conference of the Association for Computing Machinery, 1969.

22. Weber, H., "A Microprogrammed Implementation of EULER on IBM System/360, Model 30:, Communications of the ACM, Vol. 10, No. 9, pp. 549-558, September 1967.

23. Wilkes, M. V., "The Growth of Interest in Microprogramming - A Literature Survey", Computing Surveys, Vol. 1, No. 3, pp. 139-145, September, 1969.