Comments on MIL-STD-1862, 28 May 1980.

I quote the document's opening paragraph:

"1.1. Scope. This standard defines the Instruction set architecture for the Military Computer Family (MCF) of the U.S. Army. The instruction set architecture includes all information required by a programmer in order to write any time independent program which will execute on computers conforming to this standard."

Very wisely, its authors have remained anonymous, for its English is atrocious. (For instance: delete in the first quoted sentence "architecture", and replace in the second quoted sentence "The instruction set architecture" by "It", "time independent" by "time-independent", and "which will execute on" by "for". The verb "to execute" is really transitive!)

Its lousy use of English doesn't only make the text ugly, but also makes it obscure. Sometimes a bit is 0 or 1, sometimes a bit is set or clear, and sometimes a bit is set or reset. With a word size of 32 bits the sentence "This 16 bit [should be: 16-bit] descriptor will be aligned on a word boundary." isn't very clear. I was duly puzzled by "The implementation Virtual Address space [should this be: Space ?] is the minimum of the distinct virtual addresses which can be generated and mapped using an addressing mode with memory mapping enabled." Also by "The im-

plementation virtual address space shall be a minimum of 2^{24} bytes or 8 times the maximum implemented physical address space, whichever is greater." (These sentences still puzzle me.) The obscurities quoted are only examples.

* * *

The document contains no formal syntax of the syntactic category <instruction>. The document contains no formal definition of the semantics; each time the informal semantics is "explained" by an example. The document is schizophrenic in the sense that, besides attempting to give the functional specification for a piece of hardware, it also gives what looks like an assembly code. The latter is not explained at all; yet we find it used in the "explanatory" examples. To be used as a standard, the document is, as a result, profoundly inadequate; we must judge its anonymous authors as insufficiently competent.

* * *

The authors pride themselves that the instruction set has been tightly coded. Strictly speaking that claim is empty without explicit assumption about the frequency distribution of the various instructions; I think that for most distributions encountered in practice the claim can, indeed, be justified. If I have interpreted the standard correctly, the tight coding is, however, not without anomalies, e.g.: if the register in question

is not R0 (= program counter), Indirect Register Mode is equivalent to Register Indexed Mode with the displacement = 0. In contrast to the Scaled Index Mode, which can address bytes, halfwords, words, and doublewords, Unscaled Index Mode can only address the first three, but no doublewords (like Indirect Register Mode and Register Index Mode). Scaled Index Mode gives the option of addressing a halfword either as a single halfword or as a doublebyte; similarly a word may be either a single word or a double halfword. Finally, indirect addressing of double word seems only possible by means of Scaled Index Mode with the index set to zero.

Note. Register Indexed Mode, Scaled Index Mode, and Unscaled Index Mode allow adressing with respect to the program counter (= R0), which, of course, has its value changed during the execution of the current instruction. The informal definition of their semantics fails to define with respect to <u>which</u> value of the program counter addressing takes place. With respect to the address of the leading byte of the instruction? With respect to the address of the leading byte of the operand specifier in question? Who knows! This is a telling illustration of the incompetence with which this "standard" has been written. (End of Note.)

Section 8.1.5. ("Alignment of Context Pointers") is too obscure to be understood by me. (I can correct the first sentence by replacing "The question....." by "The answer to the question....."; for the next sentences I cannot suggest a correction.)

Parameter Linkage (section 8.4) solves soundly the problem of translating the actual parameter -expressed in the context of the call- into a formal parameter to be interpreted in the context of the called procedure. The information contained in a single "parameter descriptor", however, is rather meagre, and I am afraid that it will only suffice for the simplest of programming languages. More demanding parameter mechanisms can possibly be implemented by allocating a few "parameter descriptors" to each parameter.

Reading section 9 ("Exceptions") I would have benefitted from a description how the designers propose to use the exception handling facilities. I am a little bit worried by the fact that a single UDLE bit controls whether <u>all</u> exceptions will be handled by the Supervisor or <u>all</u> exceptions will be handled by the procedure's own Exception Handler. The leading sentence of the section seems to equate exceptions with program errors; avoiding such exceptions seems preferable to using the elaborate handling facilities.

The intended rôle of bits 30:31 of the Map Pointer Registers (Section 12: "Memory Management System") is unclear to me. It controls "relocation" and "protection" per map; I would have expected the control per map entry. Reasonable performance of the translation from virtual to physical addresses requires requires some sort of associative store for the Segment Descriptors of the two current maps. Shared routines must have the same virtual address in all programs that use them; there are no "pages": the segments are the units of presence in/absence from primary store. During its complete lifetime a program had better occupy the same place in virtual store, which is a severe restriction for programs that never die. Clearly the assumption is made that a virtual store of 2^{32} bytes is large enough to avoid conflicts. It is unclear whether two independent concurrent programs are allowed to occupy overlapping areas in virtual store. (I think they are.) I think the system is OK, though I would feel more comfortable when I saw a virtual store allocation policy that would be feasible and free from conflicts.

I did not study the I/O Controllers (Section 13) and the instruction set for channel programs in detail. (I tried, but felt I was assumed to know too much.) The "Indirect Buffer List" (Figure 13-1) strongly suggests that a channel can be pre-

pared so as to generate in due time more than 1 interrupt. What happens when a second interrupt is generated before the first one has been honoured? I fear that one interrupt gets lost. I doubt that further "messages" can be safely given to an active channel, for I saw little trace of the usual precautions.

Masking of I/O interrupts takes place by means of "priorities", assigned to the interrupts, and to be compared with the current priority of the processor as specified in bits 4:8 of the PSW. (Section 11.1: Interrupt Priority). I thought that by now enough people knew that such arrangement should be considered a mistake.

Without further justification "the proposed IEEE standard for binary floating-point arithmetic" has been adopted. (It looks very messy.)

The instruction code is a bit baroque. On pages 85 and 86 we find suddenly a Push and Pop for which R1 acts as stack pointer; R1 is also copied at procedure call and return. I found no description of the special rôle that has been conceived for R1. A second stack to transmit the additional information for which the Parameter Descriptor is too meagre?

* *
*

"Compare and Swap" (p.88) is specified to be "an interlocked operation", i.e. "no other memory access" to one of its operands until a certain stage of progress of the instruction execution. Compare and Swap is one of the rare instructions where "interlocked" is prescribed and, by default, we must assume allmost all instructions to be non-interlocked. But that notion is only defined in terms of "atomic memory contacts," and about these the "standard" is remarkably silent. If a program fetches a one-word operand around the time that another activity may change that same word, is the program guaranteed to fetch either the old or the new value but never to fetch a mixture? In (usually unavoidable) circumstances such guarantees are crucial. (If the atomic memory contact transfers one byte, reading a 32-bit clock had better be defined and implemented as an "interlocked" operation!) Concerning the possible interleavings of which memory accesses the "standard" is extremely vague; in my experience even disastrously so.

On page 121 we have two move instructions – potentially long moves – for which is postulated that they "shall be interruptable". Can that be built? The non-interruptable version would have an anonymous variable keeping track of the progress, but that cannot be saved and restored. Starting the complete move over again isn't safe either: the operands may have come from the store

from locations the contents of which have been destroyed by the part of the move prior to the interrupt. (When that specification was written down, someone didn't use his or her brains.) On account of the existence of memory management traps, each instruction must be interruptable, and one had better see to it that each instruction can be restarted. (The necessary hardware by means of which "the danger of unproductive page flutter" can be exorcized efficiently doesn't seem to be required by the standard.)

* * *

The more I am reading the MCF standard, the more distasteful it becomes. At first sight the design struck me as rather conventional and conventionally elaborate, and the document struck me as the unconvincing result of a job done without either love nor care. A second reading reveals it as too rough a sketch.

Acknowledgement. C. Bron, H. Maaskant, and J. van de Snepscheut helped me to read the document; I owe them my gratitude. (End of Acknowledgement.)

Plataanstraat 5
5671 AL Nuenen
The Netherlands

11 September 1980

prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow