An implementation of a virtual CAMAC processor

A. LANGSFORD
AERE Harwell, England

1. Introduction

The impetus to design and build a 'virtual CAMAC processor' came from experience in using CAMAC in a multi-user, multi-programming environment, gathering data for nuclear physics experiments [1]. Two factors were major influences in the design:

1. Programs tend to have a short life-time in this environment. As the needs of each experiment changed, it was found necessary to recode programs. This meant that not only high-level (FORTRAN) source code changed but also the individual CAMAC operations. Changing the high-level code is relatively easy, but manipulating bit patterns representing CAMAC commands was, by contrast, difficult and more prone to programmer error.
2. To speed data gathering, autonomous transfer of data between CAMAC modules and computer memory was implemented, the program controlling these transfers being hardware encoded within CAMAC.

Recognising that CAMAC had considerable processing power independent of the Central Processor Unit, it was a reasonable step to suppose that the CAMAC controller and the main processor could form a dual processor system, each accessing a common memory for their instructions (see Fig. 1).

Rather than build such a processor in hardware at the outset, it was decided to build a 'virtual processor' [2]. A simple hardware interface between computer and CAMAC dataway was supplemented by a software driver which interprets sequences of instructions placed in memory as CAMAC sub-routines. This approach has the advantage that, if the instruction set and addressing technique chosen for the virtual processor is found unsuitable, it can be readily altered.

2. Processor structure

The processor is designed to obtain its instructions from a read-only program segment. Data is held in a read-write permitted data segment. The

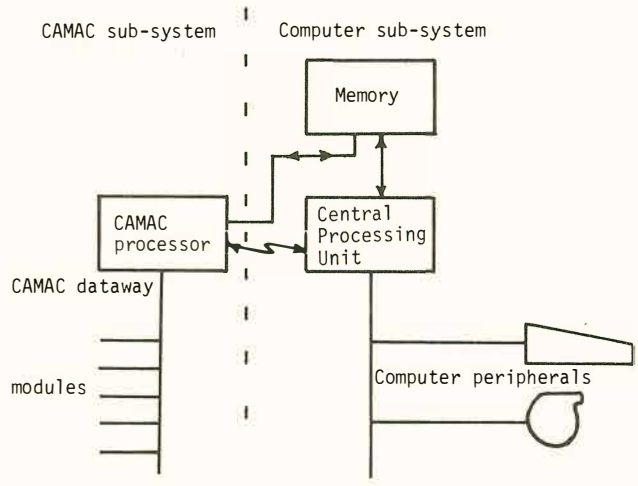


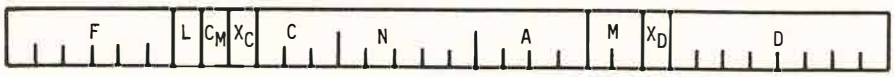
Fig. 1 Schematic diagram of CAMAC and CPU as a dual processor system

base addresses of these two segments are passed to the CAMAC processor whenever a sequence of CAMAC operations is called, the addresses being held in a Program Segment Register and Data Segment Register, respectively. In the present implementation each segment has a directly addressable length of 256 words.

The processor has, in addition, three active registers:

P	the program counter	(8-bit)
Q	the CAMAC condition register	(1-bit)
X	an index register	(8-bits)

A two-address structure has been chosen, the form of the instruction word being shown in Fig. 2. Because the CAMAC processor has been implemented on a 16-bit word length central processor, it was found convenient for the CAMAC instruction to occupy two central processor words, i.e. 32-bits. The simplest type of instruction provides a direct transfer of data between a CAMAC module (addressed by crate, C, Module, N, and Sub-address, A) and a location in the data segment, given by the displacement address D. All 32 CAMAC functions are implemented in this way, though half of them do not require a data address-



F	function code	N	module
L	length bit	A	sub-address
C_M	CAMAC address mode	M	displacement address mode
X_C	CAMAC index bit	X_D	displacement index bit
C	crate	D	displacement field

Fig. 2 The CAMAC processor instruction word

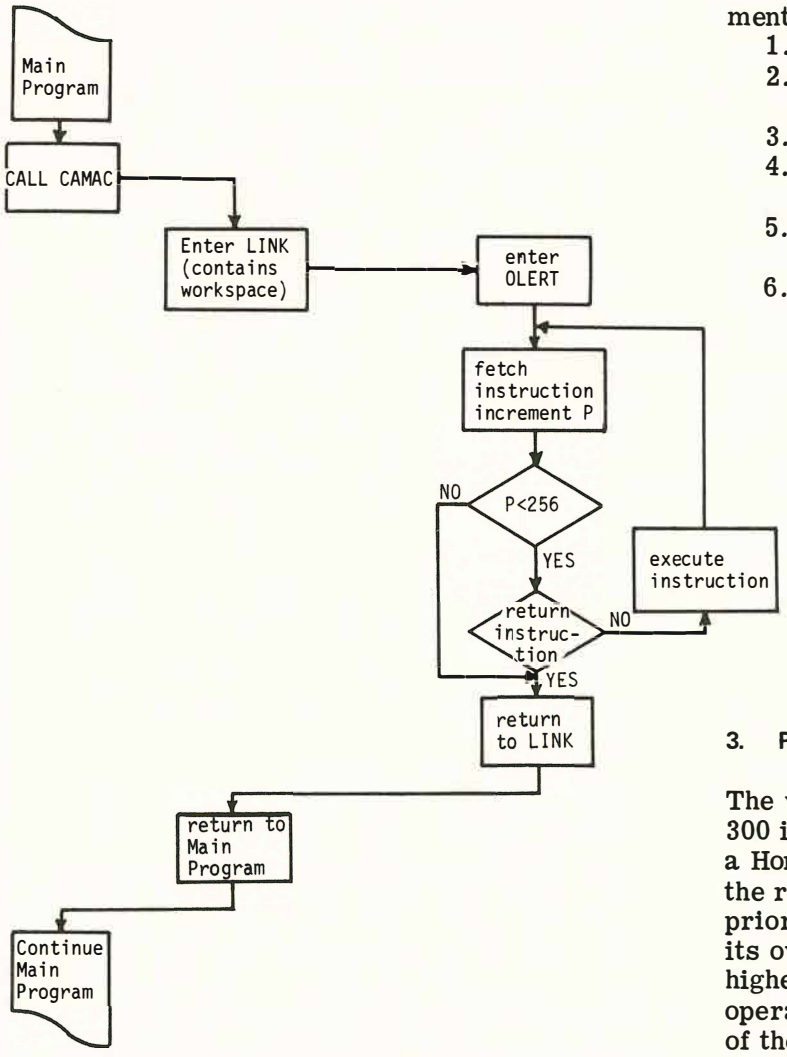


Fig. 3 Flow chart showing access to CAMAC interpreter

part being data-less transfers. However, CAMAC data-less functions 8, test-look-at-me, and 27, test-status, which affect the condition register, Q, have an address part. The contents of the specified address are set logical 'true' or 'false' depending upon whether Q is 'set' or 'cleared'. There are in addition 'non-CAMAC' operations which permit operations on the X and P registers. The latter provide branch unconditional and branch conditional on the value of Q. There is a return instruction which provides an exit from the CAMAC segment. Any instruction which cannot be decoded is treated as a return.

A number of address modes have been implemented. These provide for:

1. Direct CAMAC addressing.
2. Indirect CAMAC addressing, the direct address being found in the data segment.
3. Direct data segment addressing.
4. Indirect data segment addressing with pre-indexing.
5. Indirect data segment addressing with post-indexing.
6. Direct program segment addressing.

3. Processor implementation

The virtual processor has been written in about 300 instructions of re-entrant code to operate on a Honeywell DDP-516 computer working under the real-time executive OLERT. It runs at the priority level of the calling program which holds its own set of CAMAC registers. In this way higher priority processes may interrupt the operation of the CAMAC processor, the context of the interrupt process being automatically preserved. It is necessary to inhibit processor interrupts only for short periods. These are essentially during the sequence of operations to transfer data over the CAMAC data-way, when the operations –

set up command word: transmit data: test Q

– must be treated as an indivisible instruction.

The user program accesses the virtual processor through the FORTRAN call statement:

CALL CAMAC (data segment, program segment name)

or its assembly language equivalent.

The data segment is an integer array declared in a DIMENSION statement. The program segment can either be an internally named array or an external subroutine name. The sequence of operations carried out in accessing, running and return from the virtual processor is shown in Fig. 3. It will be seen that entry to the executive is through a link which holds workspace and the registers P, X and Q peculiar to the calling program.

4. Possible processor extensions

The author is aware that there are alternative approaches to the design of a CAMAC processor, real or virtual, and that many of the features of the present design result from the environment in which it has been implemented. With increasing experience and the freedom to manipulate the design of the processor, it is hoped that improvements will follow and omissions be remedied. Already certain desirable operations can be seen. It would be useful to have a conditional branch instruction, where the condition was set by the pattern of data transferred between store and CAMAC data-way. A far more significant omission, and one which the author intends to remedy soon, is the lack of real-time features. Because CAMAC provides for autonomous data transfer and interrupts, the virtual processor needs additional instructions so that parallel processing can be accomplished. To do this satisfactorily requires information from the user program to identify a point in his program whose execution can be scheduled when both the current CPU process and the parallel CAMAC process have terminated. While the author knows, in principle, how to achieve this, for demonstration purposes the present implementation has to be made compatible with the structure of OLERT, the environment in which the code will be executed.

It is the author's belief that, by making it more easy to program CAMAC operations, especially in a real-time environment, a better understanding of the structure of CAMAC programs will emerge. This will, in turn, be reflected in the improved definition of CAMAC statements in high-level languages. Moreover, defining a range of processors, real or virtual, on which CAMAC statements are to be executed provides ready implementations for the high-level language statements.

1. LANGSFORD, A., JARVIS, O.N., and WHITEHEAD, C., 'DAMUSC – a direct access, multi-user, synchrocyclotron computer', AERE R. 6832.
2. LANGSFORD, A., 'An implementation of a virtual CAMAC processor and its assembly language', AERE R. 6914.

Discussion

Q. What is the speed of the hardware in your system?

A. The computer itself is a microsecond machine, but the interpretation is appreciably slower than this: not fast enough for high speed CAMAC operations.

Q. What is the size of your interpreter?

A. About 300 instructions.