Propriété littéraire et scientifique réservée pour
tous les pays du monde. Ce document ne peut
être reproduit ou traduit en tout ou en partie
sans l’autorisation écrite du Directeur général
du CERN, titulaire du droit d’auteur. Dans
les cas appropriés, et s’il s’agit d’utiliser le
document à des fins non commerciales, cette
autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des
inventions brevetables et dessins ou modèles
susceptibles de dépôt qui pourraient être décrits
dans le présent document; ceux-ci peuvent être
librement utilisés par les instituts de recherche,
les industriels et autres intéressés. Cependant,
le CERN se réserve le droit de s’opposer à
toute revendication qu’un usager pourrait faire
de la propriété scientifique ou industrielle de
toute invention et tout dessin ou modèle dé-
crits dans le présent document.

© Copyright CERN, Genève, 1974

Literary and scientific copyrights reserved in
all countries of the world. This report, or
any part of it, may not be reprinted or trans-
lated without written permission of the
copyright holder, the Director-General of
CERN. However, permission will be freely
granted for appropriate non-commercial use.
If any patentable invention or registrable
design is described in the report, CERN makes
no claim to property rights in it but offers it
for the free use of research institutions, manu-
facturers and others. CERN, however, may
oppose any attempt by a user to claim any
proprietary or patent rights in such inventions
or designs as may be described in the present
document.
PREFACE

The third CERN School of Computing was held at Godøyysund, near Bergen, from 11 - 24 August 1974. The majority of the 62 students who participated are engaged in various computing activities in high energy physics laboratories in the Member States of CERN. As at previous Schools, the lecture courses covered complementary topics in experimental physics data-processing and computer science, with the aim of increasing the awareness of the students of new developments related to their field of work.

On behalf of the Organizing Committee and all the participants, I would like to record our gratitude to our hosts, the Norwegian Research Council, the Bergen City Council, the Norwegian Foreign Ministry and A/S Norsk Data-Elektronikk for the help they gave us in arranging the School, and also to Dr. E. Lillestøl and Dr. E. Lil lethun of the University of Bergen, who took the responsibility for making the necessary local arrangements.

The efficient organization by the CERN Scientific Conference Secretariat and, in particular by Mrs. Ingrid Barnett, ensured a very smooth-running School.

The present Proceedings are a photo-offset reproduction of the texts which were kindly provided by the lecturers at the time of the School; the rapid publication of this volume is due to the efficient work of the CERN Document Reproduction Service.

It is a pleasure to thank all those who assisted with the School - the Organizing Committee, the Norwegian host organizations, the lecturers, students and CERN staff - for their contribution to making the two weeks at Godøyysund a very absorbing and enjoyable School.

G.R. Macleod
Chairman
Organizing Committee
CONTENTS

INTRODUCTION TO COMPUTER SYSTEMS ARCHITECTURE - HARDWARE

F.H. Sumner

THE M.U.5 COMPUTER SYSTEM

F.H. Sumner

INTRODUCTION TO COMPUTER SYSTEMS ARCHITECTURE - SOFTWARE

S.H. Gutboud-Ribaud

MULTIPROCESSOR SYSTEMS

G. Massaré

SPECIAL PURPOSE PROCESSORS

C. Verkerk

FUTURE DEVELOPMENTS IN COMPUTER ARCHITECTURE AS A RESULT OF IMPACT OF THE MICROCOMPUTER TECHNOLOGY

L. Monrad-Krohn

DATA ANALYSIS TECHNIQUES FOR HIGH ENERGY PARTICLE PHYSICS

J. Friedman

ERASME - AUTOMATIC PROCESSING OF BUBBLE CHAMBER PHOTOGRAPHS

W. Jank

SOFTWARE ENGINEERING

J.N. Buxton

INITIATION TO HYDRA

R.K. Bök

PROGRAMMING DISCIPLINE

O.-J. Dahl

ORGANIZING COMMITTEE

436

LIST OF PARTICIPANTS

436
1. INPUT/OUTPUT

The input and output of data to digital computers has always presented problems to the designer. For batch processing systems the input is generally either from paper tape or cards and the output is by means of a line printer. Input/output therefore uses electro mechanical devices which have always been slow relative to the speed at which computers can receive or transmit data. In the twenty-five years that computers have been in use the speed of input devices has increased by a factor of about ten and that of output devices by a factor of about a hundred. In this same period computing speeds have increased by a factor of ten thousand, so that the problems presented by slow input/output devices have got steadily worse.

1.1 Early system

In the early fifties systems were very simple, the computers had little or no operating system (in fact the term had not been invented) and they were used by one person at a time in a 'hands-on' manner, the user put in his input, watched the machine compute, and collected his output before passing the system to the next user. To a first approximation, all the input occurs at the beginning and all the output at the end, so that the job profile may be drawn

| INPUT | COMPUTE | OUTPUT |

During input and output the major part of the computer is idle for most of the time as the instructions to drive the input/output devices can be obeyed more quickly than the devices can react. In the very early systems this was tolerated but as the computing speeds increased the cost of having the computer idle during input/output could not be tolerated and new techniques
were developed.

1.2 The IBM 1401 approach

IBM used a small, relatively inexpensive, computer the 1401. This is essentially a card reader, a line printer and a tape deck. The users presented their jobs in the form of decks of cards to the 1401 and these jobs were transferred by the 1401 onto magnetic tape. When a suitable sized batch of jobs had been assembled the entire tape was taken off the deck on the 1401 and transferred to a deck on the main computer, say a 7090 or 7094. The jobs were then read one by one off the tape, computed on and the results produced were written to a second magnetic tape. The input/output rate with magnetic tape was approximately fifty times that of the card reader and line printer, so that the time in which the system was idle waiting for input/output was reduced virtually to zero. At the end of computing on the batch, the output tape was transferred to a 1401 and the results printed. Apart from the obvious advantages to the main computer a significant feature of this approach is that if the input/output load changes, the number of 1401 computers supporting the main computer can be easily varied. From the users point of view the disadvantage was that he had to wait whilst the whole batch was processed before he could get his results.

1.3 Input/output wells

By the early 1960's computers were being developed which ran under the control of operating systems. These computers could divide their time between several concurrent tasks and in particular they could obey the necessary orders to drive a slow peripheral device for a few milliseconds in each second and spend the remainder of the time computing on some other problem. The concept of input and output wells was developed as shown in Fig. 1. The store of the computer was divided into three areas, the input well, the computing area and the output well. Jobs were submitted by normal slow devices and the input data assembled
in a queue in the input well. When the system is ready to start computing on a new job the data for the job at the top of the input queue is copied from the input well into the computing area of the store. This transfer takes place at computing speeds so that the system is never idle and the transfer time is very small.

At the end of computing on a job all the results are collected together and transferred to a queue in the output well. This output queue is then emptied at peripheral speeds to the output devices.

For a job $j$ the input time is $I_j$ and the output time $O_j$ during these times some computing is required to drive the peripherals say $C^I_{j}$ and $C^O_{j}$ and the computing time for the job itself is $C_j$. Summing over a reasonable period of time, provided that

$$\sum C_j + \sum C^I_{j} + \sum C^O_{j} \geq \sum I_j$$

then the input well will never need to be empty and jobs can flow steadily through the system without the main computer being idle. Also provided that

$$\sum C_j + \sum C^I_{j} + \sum C^O_{j} \geq \sum O_j$$

then the output devices will be able to remove all the output.

The attraction of this scheme is its flexibility. If the work profile changes so that $\sum I_j$ or $\sum O_j$ increases the number of slow peripherals can also be increased. If the job profile is very variable the lengths of the input/output queues will fluctuate widely. As the criterion for efficient operation is that the input well should never be empty and the output well should never be full, these fluctuations can be accommodated by increasing the sizes of the two wells. Only a small part of each well needs to be in the core store and the expansion takes place onto disc or tape as shown in Fig. 2.

From the users point of view the system is not at first sight very different from the 1401 batch system, a job has to
wait in a long queue before being processed. However, there is no reason why there should only be one queue in the input/output well. There can be several queues of different priority and if a user is willing to pay for the privilege his job can very rapidly be pushed through the system.

1.4 Front end processors

In modern computers implementing the concept of input/output wells there is often a special front end processor to deal with all the input/output activities. In these cases the input well has one end in the store of the front end processor, the middle is on the file store (generally on disc) and the other end is in the store of the main processor; the output well is arranged in a similar fashion and the overall structure is shown in Fig. 3.

The similarity with the old 1401 approach is now very clear a human link has been replaced by a high speed hardware link and the fact that the queue, or queues, are on a randomly accessible memory means that movement of individual jobs through the system can be scheduled, either in response to user demand or in order to optimise the overall system efficiency.

1.5 The control of input/output

1.5.1 Interrupts

It was stated above that modern computers spend a small part of their time dealing with the control of peripherals and in between they compute on some other problem. The switching between these different activities is achieved by means of an interrupt mechanism. In its normal state the machine is computing on one of the jobs previously input and when a peripheral requires attention it sets an interrupt flag. This flag is checked periodically by the processor and when it is seen to be set the processor stops the current program and switches to an "interrupt routine" which will take any necessary action. The switch can be quite time consuming as the current program has to be stopped cleanly and all the registers have to be preserved so that the program can be restarted after the interrupt and
continue as if nothing had happened. In many computers special hardware is designed to make this switch very fast and the interrupt routines are frequently restricted in the facilities available to them so that the switch is less complicated than a complete program change.

1.5.2 Crisis time

Some peripheral devices must receive attention within a short time of requesting it. This time is known as the crisis time and the best way to proceed is to use a particular example to illustrate the way peripherals with crisis times are handled. Consider a paper tape reader with a single character buffer, also assume that, although it can be stopped between characters, that this is not normally done as it would slow down the operation. Once a tape reader has started running at say 500 characters a second then if the character in the buffer is not removed by the central processor within two milli-seconds then the character will be overwritten by the next one from the tape and information will be lost. Thus the crisis time for this reader is two milli-seconds. Normally there are several peripheral devices and other sources of interrupt and signals from all these sources are put together to produce a single interrupt flag. This is generally tested at the beginning of each instruction and if it is set the switch to the interrupt routine takes place. The first task of this routine is to determine which of the several sources of interrupt has occurred and then to transfer to the operating system routine specific to this device. In the case of the tape reader this routine will put off the interrupt signal, read the character from the buffer, possibly check its parity, and pack it away into a buffer within the store.

After processing the source of the interrupt a check is made to see that no other interrupts exist before returning to the interrupted main program. This is necessary for two reasons, firstly there may have been several interrupt signals present initially or secondly other interrupt signals may have occurred
whilst the first interrupt was being processed. Interrupt routines themselves are not interruptable so that any interrupts occurring whilst the computer is processing a previous interrupt have to wait.

It is obviously possible for several interrupt signals to be present at the same time and the computer must decide which to process first. The signals are arranged in order of priority according to the crisis time of the devices so that the device with the shortest crisis time is always dealt with first.

1.5.3 Interrupt tree
An example of an interrupt structure is shown in Fig. 4. Here there are 512 possible sources of interrupt and the one of highest priority is in position 511. These signals are put together in groups of 8 to form 64 possible interrupt signals at the next level and so on down to a single interrupt. A special instruction which gives the most significant of a group of 8 digits will then permit a very rapid scanning of this tree to find the interrupt of highest priority at the first level.

Such provision of special instructions is quite common in these circumstances as it is very important to keep the length of the interrupt routines as short as possible. The longest routine must be shorter than the shortest crisis time and all routines must be of such a length that any interrupt will be processed within its crisis time even if all devices of higher priority are also interrupting.

1.5.4 Multi-level interrupts
In some systems there are several levels of interruption and interrupt routines at a lower level can be interrupted by signals from devices at a higher level.

1.5.5 Increasing crisis time
In a large system with many peripherals crisis times can be very important. These can always be increased by providing
extra buffering at the interrupting device. For example if the tape reader could store 4 characters before interrupting, then the crisis time would be 4 times as long and the interrupt routine itself would have to pack away 4 characters into the input well. This could slightly increase the length of the routine or if 4 characters equalled one word the routine would, in fact, be shortened. The correct design of peripheral buffers can therefore be seen to be of considerable importance.

1.6 Autonomous transfers

In the system just described the information from the peripherals essentially flows through the CPU into the central memory; for some peripherals, for example, magnetic tape or disc the data rate is such that no interrupt system could cope no matter how big a buffer was used to reduce the frequency of interrupts.

The technique employed in these cases is to provide more than one entry point into the memory, for example, there could be three, see Fig. 5, one from the CPU, one from the magnetic tape system and one from the drum system. Suitable priority logic would have to be provided so that requests to the store from the multiple ports could be dealt with without either having addresses or information lost. This system of some cycles of the memory being given to the drum or tape system is often referred to as cycle stealing.

The method of transferring data from say the drum to the store would now be as follows. An interrupt would occur which meant that a transfer from the drum to the core was wanted. The interrupt routine dealing with this would send to the drum system the address of the first line of the block of information that was wanted and the first line of the block of store into which it was to be transferred and the length of the block. The interrupt routine would then end and the drum system would send information directly into the store as necessary. This information consisting of the address and the content of the word to be transferred.
When the complete block had been transferred a second interrupt would occur and this interrupt routine would check that no faults had occurred and generally tidy up after the transfer.

Autonomous transfers use just as many memory cycles of the main store for data accesses but there are far fewer instructions and the load of the CPU is very much less. Extra hardware has to be provided and the access time for the store is slightly increased by the priority logic but it is generally well worth it and in some systems even the slow peripherals transfer data in blocks by means of an autonomous channel directly into the core store.

1.7 Dedicated peripheral processors

A totally different approach to peripheral control is to have one or more small processors each with its own memory dedicated to looking after a group of peripherals. For example, consider again the control of a paper tape reader. This would again have a one character buffer and would set a flag when it required attention, the small computer would sit in a loop looking at the flag, as soon as it became set the correct routine would be entered to reset the flag and copy the contents of the paper tape reader buffer into a buffer area within the memory of the small computer. The computer would then return to the loop in which it looked at the flag waiting for it to be set again by the reader. With the relative speeds of computers and I/O devices it is obvious that one small computer could look after several devices by having the loop scan around the flags of all the devices and having a set of routines to deal with all of the different circumstances. The criterion now for a device not going overdue would be that the computer got around to looking at its flag within the crisis time irrespective of other activities. This is the technique used in the CDC 6600 in which there are ten peripheral processing units. The instruction time for these processors is approximately 1 micro-second so that one PPU can handle all the slow peripherals of a typical installation. The
PPU's are in fact so fast that the same approach can be used to control magnetic tapes or discs but here the transfer rate is such that one PPU has to be dedicated to each disc channel.

The information is transferred from the store of the PPU to the store of the main computer by the PPU sending a request for a main memory access via a priority circuit to a multiple memory port as was used for autonomous transfers. In fact a PPU may be regarded as a very flexible, programmable autonomous channel.

1.8 Control of individual peripherals

How does the computer control a peripheral? Continuing with the example of a paper tape reader, the control signals to the reader might be start, stop, remove the interrupt flag, whilst the input to the computer could be the bits of the character in the paper tape buffer together with the interrupt flag. In early computers with only a few peripherals there were specific functions to start, stop etc., and the read function caused the bits of the paper tape reader to be inserted into one of the machine's central registers. As the number of peripherals increased this was obviously wasteful of functions and the same function patterns were used to control different peripherals where the peripheral was defined by the address field of the instruction. A later development on the Atlas which is now quite widely used is to use only write and read orders to send the content of one central register to the addressed peripheral or to read from the peripheral to the central register. Certain digits positions in the word correspond to different functions and the presence or absence of a 1 in that position determines if the action is to occur. Other digits contain information. This is a very flexible system as all that is needed when a new peripheral is introduced are some addressable locations associated with the peripheral to and from which the correct information and control patterns can be transmitted using standard read and write orders. Provided, therefore, sufficient address space is available which
is very easy with a large virtual address system, then the number and type of peripherals is almost unlimited.

2. ADDRESSING AND STORE ORGANISATION

2.1 Instruction formats

2.1.1 One address formats

The vast majority of computer operations produce an output which is a simple function of two inputs. In several computers there is a single register sometimes called the accumulator register, associated with the arithmetic unit. The arithmetic unit hardware is organised so that the content of this register is one of the inputs and, after the completion of the operation, the output from the arithmetic unit goes back into this register. Instructions therefore only need to specify the nature of the operation and the address of the store location, the contents of which are to be used as the other input. These two fields of the instruction are referred to as F and S respectively.

The introduction of address modification requires a third field (B) to specify the modifier register whose content is added to the S field to give the operand address. The format for a single address instruction is therefore F,B,S.

The length of the complete instruction and of the individual fields are governed by several factors. The principle restriction on the length of the instruction is that this length should be the same as that of the computer word or should be a binary fraction of that length. If this is not the case then addition to control is rather difficult. A computer word generally holds one floating point number and the length of this word is therefore governed by the desired range and accuracy of these numbers. If the computer can handle characters then it is also convenient for the word length to be a binary multiple of the character length.
As an early example in the Ferranti Mercury the character length was 5-bits and a word contained 8 characters or 1 floating point number. There were 1,024 lines of store requiring 10-bits in the S field, and 8 index registers requiring 3-bits in the B field. Therefore, it was sensible to have an instruction of length 20-bits and pack two instructions into each word. The F field was left at 7-bits which is more than sufficient to identify all the available functions. If any redundancy exists in the instruction it is useful to have it in the F field because this permits quicker decoding of the function.

In the Mercury computer the addressable store was only 1,024 lines so that the required S field of 10-bits could be accommodated in an instruction length of 20-bits. As store sizes increased so also did the length of the S field required for direct addressing. If the normal F,B,S format was adhered to this would lead to instructions too long to permit packing of two per word. As the only choice after two instructions per word is one per word this leads to very long instructions. The Atlas computer which has a large virtual address space, and two modifiers, uses instructions of 48-bits, each of which occupy one word of the store. An alternative approach which was used by IBM in the 360 series and ICL in the 1900 series is to restrict the length of the S field to a length which can conveniently fit into a short instruction. This means that the whole store cannot be addressed directly and the only way of getting at the upper end of the store is by means of modification, the modifier registers being long enough to cover the whole store. This technique has some disadvantages in programming but these are more than outweighed by the efficiency of short instructions. For example, in the 1900 series the S field is 12-bits whilst the length of the modifiers is 15-bits, permitting a store size of 32K. The 1900 word length is 48-bits and to pack a 15-bit address into half words of 24-bits would have been impossible. On the other hand expanding the instruction length to 48-bits would have lead to a great degree of redundancy.
A further advantage of this approach is very clearly shown by what ICL had to do when they decided to increase the size of the addressable store in their larger computers. All that was necessary was to increase the length of the modifiers to 22-bits, the format of the instructions remained unchanged.

2.1.2 Zero address systems

In these systems the accumulator is replaced by a group of registers known as the stack whose position relative to the store and the arithmetic unit is shown below.

```
FROM STORE
    R1 ----> A.U. ----> TO R1
    R2 ---->
    R3 ---->
    R4    
```

Starting with all the registers of the stack empty the first read from the store loads an operand into register R1. On obeying the second read order the content of R1 is copied to R2 before the new operand is loaded into R1. On subsequent read orders the contents of the registers move down one level leaving R1 free to receive the operand from the store. This process is generally referred to as stacking.

As stated earlier most arithmetic functions require two inputs and have a single output. In this system the 2 inputs to the arithmetic unit are the contents of R1 and R2 and the output of the arithmetic unit after the operation is returned to R1. After the completion of this type of operation R3 is copied to R2, R4 to R3 etc., As both inputs and the output of the arithmetic unit are defined it is only necessary in arithmetic functions to indicate the nature of the function. Arithmetic
orders can therefore be very short. The orders for accessing the store are of more or less conventional format, they require an S field and a B field, the F field however, only needs to be 2-bits, one to indicate that this is a long order, and the other to indicate read or write. The action on obeying read orders has already been described, for write orders the content of R1 is written to the selected store address, and this is followed by copying R2 to R1, R3 to R2 etc. This process is called unstacking.

This type of system therefore, requires orders of two lengths, long orders for accessing the store and short orders for performing arithmetic.

The operation of a stacking computer matches exactly the reverse Polish representation of expressions which is a common intermediate stage in most compilers. An early implementation of a stacking machine was the English Electric KDF9. In this computer the stack was called the nesting store and the processes of stacking and unstacking were referred to as nesting down and nesting up. In the KDF9 the long read and store orders were 24-bits and the arithmetic orders were 8. The number of "bits" of code required to evaluate an expression can be significantly reduced in a zero address system.

One of the difficulties in stacking systems is determining the length of the stack. Also, when stacking what happens to the content of the bottom register of the stack? In the KDF9 the stack is 16 deep and the content of R16 is lost when nesting down. In a later computer, the Burroughs B5000, the stack is extended into the store, on stacking the content of the last register returns to the store and on unstacking it is loaded from the store.

2.1.3 Two address and three address systems

For the one address system an instruction may be written as
F( [Accumulator], [Store line] ) → [Accumulator] where the square brackets imply "contents of".

Alternatives to this are a two address system
F( [Store line 1], [Store line 2] ) → [Store line 1] or a three address system
F( [Store line 1], [Store line 2] ) → [Store line 3].

A direct extension of the one address format of F, B, S to the two address system would give a format of F, B₁, S₁, B₂, S₂. Such a format has been implemented in the MANIAC III Computer, a research machine at the University of Chicago. This computer has an addressable store of 16K words and the lengths of the different fields of the instructions are

\[
F = 10 \\
B₁ = B₂ = 5 \\
S₁ = S₂ = 14
\]
giving an overall length of 48-bits which corresponds to one instruction per word.

The direct extension of the normal one address format to a two address system therefore yields rather long and cumbersome instructions. A similar extension to a three address system would require even longer instructions and I am unaware of any computer in which such a format has been implemented.

2.1.4 Register-register systems

Two and three address systems have been implemented in modern computers but this has been practicable by having a small number of addressable registers outside the store and performing the arithmetic operations between the contents of these registers e.g.

F( [Register 1], [Register 2] ) → [Register 1]

An example of such a system is the IBM 360 series of computers. Here there are 16 registers and the F field is 8-bits which gives an instruction length of 16-bits (or 2 bytes) for register to
register operations.

Zero address systems required special long orders for accessing the store, the same is true of register-register systems. For a two address code these would be either

\[ F( [\text{Register 1}] , [\text{Store 1}] ) \rightarrow [\text{Register 1}] \]

to obtain new operands from the main store or to write results back into a store.

In the IBM 360 these register/store orders have the format

\[ F, \text{Register 1, Modifier 1, Base 1, Register 1} \]
\[ 8 \quad 4 \quad 4 \quad 4 \quad 12 \]

to give an overall instruction length of 32-bits or 4 bytes. The 12 address bits only cover 4K lines directly but the modifiers and base registers are each 32 bits long.

The most well-known example of a three address register-register system is the Control Data 6600. Here the word length is 60-bits and there are 24 central registers, divided into 8 arithmetic registers X, 8 modifier registers B, and 8 address registers A. The short instructions have the format

\[ F, i, j, k \]
\[ 6 \quad 3 \quad 3 \quad 3 \]

the F field defines the function and the types of register involved and the i, j and k fields define the particular register. These 15-bit orders can then be packed 4 to a word. Operands are transferred between the store and the X registers by setting the address of the operand required in the corresponding A register. Whilst this may be done by short instructions it is more generally done by instructions with the format

\[ F, i, j, K \]
\[ 6 \quad 3 \quad 3 \quad 18 \]

Function Address Modifier Address
Register

The 18-bits of the K field cover the whole of the addressable
store and the modifier registers are also 18-bits long.

2.2 Addressing

The conventional way of organising a store is to arrange the bits in a matrix of \(2^N\) words by \(2^M\) digits. The words are numbered 0 to \(2^N-1\) and they are selected by presenting the \(N\) bits of the address of the required line to a decoder which determines the physical position of the line and then arranges for the \(M\) information bits to be read out.

An alternative way of addressing is by association. In this method each line of the store is divided into 2 parts. The associative field and the value field. In normal linear addressing each address is unique and selects one and only one word. For associative addressing to have the same property each of the associative fields must contain a distinct pattern. In this method of addressing a line is selected by presenting the content of the associative field of the required line to the associative memory and comparing this with the associative fields of all the store lines in order to find the one that matches exactly. When this match is found the value field of the selected line is read out.

The cost of this method is obviously much higher than a conventional store as the relatively simple conventional decoding tree has been replaced by a mechanism for organising the comparison, and each line of the store has been increased in length by at least \(M\) bits. Also unless the structure of the hardware is changed significantly the access time will be much longer as each associative field will have to be inspected in turn to find the match, the access time would therefore be approximately \(2^{N-1}\) times as long as for a conventional memory.

This method of addressing has frequently been implemented by software on top of a conventional memory. Generally with variable lengths of the field of association and with the ability to look for multiple matches.
In recent years technology developments have produced small associative memories in which the presented pattern can be compared in parallel with the associative fields of each line of the memory at the same time. The access time is therefore of the same order of magnitude as the conventional memory. The cost however, is much higher and the size of the memory is still very limited, tens of lines of associative memory, as compared with thousands of lines of conventional memory.

The two most common areas of use of associative memory in high speed systems are for the translation of virtual addresses to real addresses in paged virtual memory systems and in the implementation of small scratch pad memories. These uses are discussed in detail later in these notes.

2.3 Store protection

In early computers the system was used by one programmer at a time and he controlled all the activities of the system. The user had free use of all of the store, except possibly for a small amount of backing store containing basic library routines which could not be written to. With only one user there was no need to protect one user area from another and with no operating system there was no need to protect a public area of store from the user. The single user might have liked some protection of parts of his program from others but this was left entirely to him to organise by means of his own software.

2.3.1 Early systems

The first computers with rudimentary operating systems still only ran one program at once and in some of the earlier versions the user was told that the operating system was in lines 0 - X and would he please not write into this area. Even assuming no malicious actions a program could easily alter the operating system by accident so that the operating system had to be frequently reloaded. To be on the safe side, it was in fact reloaded before running each new program. The next development was to have two modes of operation, user mode and operating
system mode. In operating system mode all addresses could be
accessed whilst in user mode no accesses to addresses 0 - X were
permitted and any such attempt caused the program to be
terminated. Read accesses to the protected area were sometimes
allowed but writing was obviously not permitted. The system was
of course designed so that the users could not switch into
operating system mode. This was still rather inconvenient in
that the user had to use addresses X to N where N is the top of
the store and the next development was to protect the operating
system by arranging that whenever the system was in user mode all
addresses presented to the store had X added to them before the
access took place. Thus the user had an address space of
0 to N - X and the operating system was completely protected.
(In this system even read accesses to the operating system space
are not possible).

2.3.2 **Partitioning - datum and limit**

As computer systems developed it became desirable to have
more than one program in the system at the same time, for example,
if the programs were using the slow tape backing store then
whenever they asked for a transfer from tape they would have to
wait quite a long time before the transfer was complete. Either
the system can idle for this time, or if other programs are in
the system a switch could be made so that the system continued to
compute. In one of the very early implementations of such a
multi-programming system the users all had access to all the
address space and they trusted each other. This, needless to say,
was not very successful. One way of solving the problem
involved deciding on the number of programs that could live in
the addressable core at the same time and then dividing the store
into this number of, not necessarily equal, parts. For example,
if four partitions were wanted then the operating system would
exist in addresses 0 to X, Program 1 in addresses X + 1 to Y,
Program 2 in addresses Y + 1 to Z and Program 3 in addresses
Z + 1 to N. Each program would behave as though it had an
address space starting at line 0 and the operating system would arrange that when, for example, Program 2 was entered a central datum register had \( Y + 1 \) loaded into it and this datum would be added to all the addresses generated by Program 2. This protects the operating system and Program 1 but Program 2 is still vulnerable if Program 2 writes to an address greater than \( Z - Y \). This is prevented by the operating system setting a limit register to \( Z - Y - 1 \) and all addresses generated by Program 2 are checked to ensure that they are less than this limit.

The use of the datum and limit pair therefore ensures that Program 2 can only access the space allocated to it by the operating system and all other programs in the store are completely protected.

In all these protection schemes it is only necessary to protect the directly addressable store. All accesses to backing store are organised via the operating system and it is therefore easy to check that these accesses are to an area of backing store which is permitted to the program making the request. The direct addressable store has, of course, been protected by making accesses to it go through an indirect path and making the necessary checks along this path. This indirectness is done by fast hardware and does not delay the store access by more than one add time as can be seen from Fig. 6.

In this type of system programs run in one of the fixed partitions of the store using the partition of the most convenient size.

2.3.3 Variable datum and limit

The next development was the introduction of a variable datum and limit system. Whenever the operating system decided to start a new program it looked at the amount of core store requested by the user and then looked at its own map of the store to see if there was an empty hole big enough to accommodate the new program. If a hole was found the datum and limit were set
for the new program and it could then execute with complete protection. If none of the existing holes was big enough the operating system had two choices, it could decide not to run the program and run instead a smaller one that would fit, or it could move existing programs about the store altering their datum settings in an attempt to merge several holes together to form one big enough for the new program.

This system worked quite well and is still used in many computer systems. The difficulty is the fragmentation of the store into small unused areas which then have to be merged together. In this type of system as in all the others described above the user has access to a fixed area of directly addressable core, and any extra space he needs is provided on the backing store, the programmer then organises the transfer of data between his core area and backing store area to the best of his ability. This type of activity is often referred to as overlaying.

2.3.4 Paging

When the Atlas Computer was being designed between 1958 and 1960 there was an obvious need for multi-programming with all the attendant protection problems and there was also an awareness that the user did not like organising his own overlays. It was against this background that paging was invented. It is convenient and probably easier to understand if paging is described by considering its actual implementation on the Atlas.

The Atlas at the University of Manchester had 16K words of core store and 96K words of drum store with 20 bits available for addressing this space. The core and drum were each divided into blocks or pages of 512 words and the 20 address bits were divided into 9 line bits and 11 page bits. For each of the programs currently active there was a directory relating the programmers address of a page with its actual physical position on the core or drum. The pages of any program did not have to be in contiguous real locations they could be anywhere on the core or drum. For the 32 pages of core store there were 32 registers
known as Page Address Registers (PAR). These each have 12 bits, see Fig.7, of which 11 held the block number of the block currently in the corresponding page and the 12th was a lock-out digit which if set to 1 prevented any access to that page except by the operating system. For the program being obeyed the generated addresses were split into line and block digits, and the block digits were presented to the PAR. If the required block is in the core then one of the PAR would signal equivalence and a 5 bit number would be generated to indicate the real page address, this was then concatenated with the 9 line bits to give a 14 bit address to select a line from the core store. If pages of other programs were also in the core store their corresponding PAR's would have the lock-out digits set thus preventing any unauthorised access. If an address was generated which did not get equivalence on one of the PAR then an interrupt occurred and the operating system was entered. The address on the drum of the required block was found from the directory and this block was transferred into an empty page of the core store, the PAR of this page was set, the directory altered and the program re-entered. In parallel with subsequent execution of the program the operating system transferred one of the pages of the core to an empty area of the drum to ensure that there would be an empty page ready for use at the next not-equivalence. The selection of which page to move to the drum was very important and to aid the operating system to reach its decision each page had a use digit which was set whenever the page was accessed. By reading these use digits at regular intervals the operating system could develop a picture of the pattern of accesses of the program and this pattern was used to help make the best decision of the page to remove to the backing store.

On the very first reference to a new page this page would not be found in either the PAR or the directory as there is no need for a real page to exist for addresses which have not been used. If the program had not used its full allocation of pages
the empty page in the core was designated to be the required page and the PAR and directory were set accordingly before re-entering the program.

The users of the system therefore only define the pages they required and if that number of pages were free anywhere in the system then the program could be started. The users were completely protected from each other as all pages of other programs were locked out if they happened to be in the core store. Of great importance to the user was the fact that he did not need to bother where the pages of his address space were physically located, they could be on the core or the drum and the operating system would move them between the two levels. The user, in fact treated the combination of the core and drum as a pseudo one level store.

This was ideal from the users point of view, as he no longer needed to worry about overlaying. However, the system had to place the pages in current use in the core and if the user was frequently referring to more pages than could fit into the core, then the number of transfers between the two levels of store would greatly increase and the overall efficiency of the system would decrease. Extreme cases of this situation are referred to as thrashing and this technique of paging is known as demand paging. In Atlas a page only existed on either the core or the drum. In later systems with bigger backing stores a copy of all pages in the core is kept on the drum. Thus, if a page has not been altered it does not need to be copied back to the drum at the end of its period of use. This is detected by means of an altered bit associated with each Page Address Register.

2.3.4.1 Page size

One of the great difficulties in all page systems is the selection of page size, if the page is made larger then when a page is transferred to the core then there is less chance that it will all be used before it has to be moved out and therefore some of the transfer time and some of the valuable fast store will
have been wasted. Also with large pages parts of the pages are quite likely to be empty. With small pages these problems are removed but the directories increase in length and the frequency of paging demands will increase as it is less likely that a required piece of data will have been brought to the core together with a previously requested address. Also, when the backing store has a long latency, as with a drum, the access time per word is greater with smaller pages. Some systems have two sizes of page and the MVS has a dynamically variable page size.

2.3.4.2 Efficiency

The overheads of paging can be very great especially if the user disregards the actual size of the fast memory. On Atlas the system spent approximately 10% of its time organising the one level store so that this gives some idea of the cost of the advantages that paging provides.

2.3.5. Virtual Addressing

Whilst Atlas introduced the concept of paging it also introduced an equally, if not more important concept, virtual addressing. Because the two concepts were introduced together on the Atlas they are frequently confused and the implication is that one cannot exist without the other. Whilst it is generally true that most virtual address systems have paging it is not essential to the concept of virtual addressing, nor is it essential for a paged computer to have virtual addressing.

Virtual addressing is the name that has been given to the situation where the user has a virtual address space different from and often much larger than the real address space. The users virtual address space is translated into real addresses by some system such as paging.

On the Atlas there were 11 block digits giving each user 2,048 virtual pages. The system only had 224 real pages and individual users were generally restricted to less than 60 pages. One of the major advantages of virtual addressing is that the
user can spread his data across the virtual space and never have to worry about one area of his data expanding and overwriting another. For example, consider the problem of sorting 4,096 items into 4 lists where it is not known how many items will be in each list. With real addressing 16,384 real words would have to be allocated and the 4 lists started 4,096 words apart. On a virtual address system such as the Atlas the 4 lists were started at 4 virtual addresses, at least 4,096 words apart, but real space was only used as required. Initially, none of the addresses corresponding to the lists existed in real space and as demands occurred to reference particular virtual addresses so real pages were allocated to these virtual addresses. In this way, the real space occupied by the 4 lists was only slightly greater than 4,096 lines (if all the 4 lists had been multiples of 512 in length then the total space would have been exactly 4,096 but this is very unlikely).

2.3.6 Segmentation

The advantages to the user of having a large amount of virtual address space are very great and the concept introduced in Atlas was extended and formalised into segmentation in the MULTICS system on the GE 645.

In this system the address space for each user was enormous, 36 bits, of which the top 18 gave the segment name and the bottom 18 covered the possible space of the segment, these latter being divided into page and line sections. The user placed distinct types of objects into different segments and these segments then occupied as many real pages as necessary. The implementation of segmentation on the GE 645 was quite complex as the machine was not initially designed for this type of addressing, it is therefore more sensible to describe an idealised segmentation system.

A block diagram of the system is given in Fig. 8. Each address presented to the hardware is as follows
The process field is concatenated to the users address by the hardware in order to protect the virtual space of one process from that of others (process is a relatively new word which means the same as program used to mean). The process field selects the segment directory for that process and the segment field then selects the page table for that segment which then gives the starting address of the real page to which the line digits are concatenated to make access to the store.

This is a very cumbersome method and in most systems there is a small associative memory as shown in Fig. 9. The Process, Segment, Page fields of the virtual address are presented to this memory and for the pages currently in use the real page address is produced. This memory also controls the type of access to the data which can generally be obey, read or write, or any combination of these achieved by means of 3 access permission bits with each page.

Whenever a page is not in the associative memory a complete scan of the tables has to be made and if the page is in the core store then a line of the associative memory is set accordingly. If the page is on the backing store a swap similar to that described for the Atlas takes place and again the correct entry is made in the associative memory (the tables are also updated).

Segmentation permits several other important developments. Segments can be either private, shared or common. Private segments only exist in the directories of one process and can only be accessed by that process. Shared segments can exist in the directories of more than one process and common segments exist in the directories of all processes. A convenient way of providing common segments is to lose half of the virtual address space and agree that all segments with the most significant bit
equal to a 1 are in the common set and addressing of these then takes place via the common segment table. The addressing of shared and common segments is shown in Fig. 10 and Fig. 11.

For common segments the access control is the same for all users but for shared segments the access control can vary from user to user under the control of the operating system. Code is generally placed in obey only segments and such code is often referred to as consisting of pure procedures. In a multi-access environment all the code of the editors or the compilers will be in pure procedures in common segments and therefore only one real copy of the code needs to exist and this can then be included in the virtual space of any or all of the active processes. The saving of real space by this technique is enormous.

Another very important feature of virtual addressing, as far as operating system overheads are concerned, is to do with the moving of information. These moves are generally done in blocks, the move will therefore be from one virtual address to a different one. The physical pages are not moved, all that the operating system needs to do is to make the correct changes to the appropriate page and segment tables. The unit moved is normally a segment so that neither the pages nor the page table are affected, only the segment table entries are altered.

Thus in the I/O system described earlier a segment or page can be moved from the virtual space of the Input well into the computing area merely by changing a few address pointers. This is obviously much faster than copying the actual data. The different wells will be completely mixed up in real space whilst they are distinct and inter-protected in virtual space.

2.3.7 Paging in a real address environment

It was stated above that paging did not necessarily go with virtual addressing and vice versa. A good example of a system with paging and without virtual addressing is the IBM 360/85. In this system the core store has up to 4096K bytes and the
real address field is 22 bits to match the size of the maximum available real store.

The speed of the system could be increased by placing a fast memory between the main and the CPU. This memory would have to be completely invisible to all the software if the model 85 is to be compatible with the other machines in the 360 range.

The buffer memory used was called the cache and is illustrated in Fig. 12. It is made of integrated circuits and is in two parts, the associative part of 16 lines, each line 12 bits wide and the value part of 16 areas, each of 16 blocks, each block in turn having 64 bytes. The total size of the cache is therefore only 16K bytes. The associative field can do a parallel association in 80 nsecs. and the value field can be read in 80 nsecs.

Both the cache and the main memory are divided into sectors, each of 1,024 bytes so that there are 4,096 sectors in the main memory needing 12 bits to identify them. Starting with the cache empty the first address presented is divided as shown and the 12 sector bits are compared with the 16 lines of associative memory. No equivalence signal occurs so the required line is read from the main memory and computing continues. Also in parallel with this the first line of the associative memory is set to the 12 sector bits of the presented address and the 64 byte block specified in the address is read from the store and placed in the corresponding block of the first sector of the value side of the cache. The block is marked present, all other 15 blocks of this sector are, of course, marked absent.

This continues, using for each new 12 bit sector address, the next line of the associative memory. If an address is requested that gives equivalence on the sector field then the correct block is examined and if it is present the correct line or byte is read out. If the block is not present it is read from the store and is placed in the correct part of the cache.
Whenever a write to store order occurs both the cache and the main memory are written to so that the contents of the main memory are always correct. If a write to store order does not find its address in the cache then the write to main memory occurs but no other action takes place.

When all 16 sector fields are full and a new sector field is presented one of the 16 is selected and the corresponding value area is declared empty and the new sector address is inserted in that line of the associative memory. This selection is done by a simple piece of hardware that keeps the 16 sectors of the cache in an activity list, and whenever a sector is referred to it is moved to the top of the activity list. The sector chosen to be deleted is the one at the bottom of this list.

The estimated performance of this system, (Liptey IBM Systems Journal Vol. 17 1968) is very impressive. The average hit rate is 96.8% and this means that the system of cache and main memory performs at 81% of the speed of a system with all the 4096K bytes of memory operating at the speed of the cache.

This system shows very clearly that paging, can be and should be, completely invisible to the user of the system.

Segmentation on the other hand is visible to the user and is, in fact, an extremely powerful tool in the organisation of his address space.

3. **HIGH PERFORMANCE SYSTEMS**

This section will be restricted to computers of more or less conventional design. The subject of multi-computers is covered by Dr. Mazare.

3.1 **Overlapping - pipelining**

Consider a hypothetical computer with a single address instruction format with address modification (Format f, b, s). The store is fairly fast core with an access time of 400 nsecs. The indices are in fast registers and can be read in 50 nsecs. The logic performs operations such as fixed point add in
50 nsecs. For a simple add instruction the timing will therefore be

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read instruction</td>
<td>400</td>
</tr>
<tr>
<td>Decode instruction</td>
<td>50</td>
</tr>
<tr>
<td>Read index register</td>
<td>50</td>
</tr>
<tr>
<td>Modify address</td>
<td>50</td>
</tr>
<tr>
<td>Read operand</td>
<td>400</td>
</tr>
<tr>
<td>Perform add</td>
<td>50</td>
</tr>
</tbody>
</table>

Total 1,000 nsecs.

The total time for this instruction is 1,000 nsecs. and in a very simple organisation in which each instruction goes to completion before the next one is started, the rate of obeying a continuous sequence of instructions of this type would be one every microsecond.

The first technique for speeding up the rate of execution was that of early calling of the instruction. For the above example instruction 2 may be requested 600 nsecs. after instruction 1 and the timing diagram changes as shown in Fig. 13. The rate of obeying instructions is now one every 600 nsecs. and the accessing of instructions has been completely overlapped with the other activities necessary to the completion of the instruction. This very simple bit of overlapping can be extended into a system which is now known as pipelining. For the above example a fairly basic unit is the 50 nsecs. required for several of the operations. A pipeline of 20 stages, each of 50 nsecs. would therefore permit instructions to be started every 50 nsecs. and the rate of obeying instructions would be one every 50 nsecs. as shown in Fig. 13. This assumes that the store can be accessed twice in every 50 nsecs., once for an instruction and once for an operand, this may be possible in an integrated circuit memory, but for a core memory it would imply that the memory was arranged with a large number of independent stacks as in the CDC 6500, successive accesses are therefore unlikely to be to the same stack and can therefore proceed without waiting for the store to complete its
read/write cycle. Each stage of the pipeline must have sufficient
registers to hold all the necessary bits of the instruction being
processed in that stage of the pipeline, these bits pass down the
pipeline as the instruction is obeyed. The pipeline organisation
is therefore expensive but it has a potential factor of 20
improvement on the very simple system and a factor of 12 on the
second proposal.

In a real program the probability of maintaining this rate of
execution is virtually nil. Firstly the store might not always be
able to respond immediately so that requests are held up and a gap
develops in the pipeline. Secondly, not all operations are fixed
point adds, there are floating point adds and other longer orders.
With the proposed organisation the next operand cannot be operated
on until the previous arithmetic has finished so that the pipeline
will be held up and there will be a period of several beats (the
usual name for the length of the basic step) during which no
instructions will be completed. This latter difficulty can be
partially solved by pipelining the arithmetic unit or by having
multiple arithmetic units, these approaches are considered later.
The principal difficulty with the pipeline approach are the
discontinuities caused by conditional branch instructions.
For example, if in a sequence of orders doing arithmetic in the
fixed point unit there was an order that branched if the unit was
negative, then the order after this branch may be either the next
one in sequence or from some totally different address. In the
latter case it will not be requested until the test order is
completed and there will therefore be a large gap as shown in
Fig. 14. For the case of not branching the next order is avail-
able but must not be allowed to proceed too far before the
condition of the test is known in case it does something which
cannot be undone if the test succeeds. If the time for an
instruction is divided into two parts, A the access time, and P
the execution pipeline time, then for a test that fails to branch
the gap can be reduced to zero, and for a test that branches the
gap is $A + P$. If $A$ is less than $P$ then it is possible to prefetch the branched to instruction before the result of the test is known, thus reducing the gap to $P$. This requires an extra instruction buffer and this has been done in the 360/195 (see later).

Monitoring of obeyed programs in several systems suggests that the occurrence of branch orders is between 1 in 5 and 1 in 10 of all orders. The magnitude of their effect is therefore very large and much of the effort in modern computer designs has been directed to reducing this effect.

The first method is to make the pipeline shorter without increasing the length of individual stages i.e. reduce the length of the pipeline and therefore reduce the effect of discontinuities. A very good example of this approach is the IBM 360/85 cache store described above. This has an access time of 160 nsecs, in two beats of 80 nsecs, and the system operates on a pipeline with a basic beat of 80 nsecs. For branch orders the gap will be $P + A$, but $A$ is small. A difficulty with this approach is that the pipelining is designed to be correct for a hit on the cache memory which is alright for 98.6% of the accesses (see above) for the other accesses a full store access has to be inserted and during this time nothing else passes into the pipeline so that a big gap develops and the system is degraded by far more than the 1.4% of misses. The figure suggested by Liptay is 19% degradation. This is a suitable point to stress that when a hit rate for a fast memory is quoted the question to ask is "What is the degradation caused by the misses?"

Very high speed access to a limited number of operands has been achieved in many systems by providing several explicitly addressed registers into which the programmer puts the frequently used operands. This cuts down accesses to the store and permits more efficient pipelining. The difficulty is to decide which operand to put in these registers, especially in the case of compiler generated code.
3.2 Instruction buffers

There has been a steady development of the methods of getting over the access time for instructions. The early methods were very simple early call procedures, where if the system wanted an instruction rate of one every 1 nsecs and the access time was A nsecs. when the instructions were requested A nsecs. before they were required, requests were made every 1 nsecs. and a buffer of length A/I was provided to hold all the requested instructions if the pipeline stopped for any reason. The address of the requested instructions is therefore A/I ahead of the instruction entering the execution pipeline. A very early development to make the organisation easier was to transfer several instructions from the store at each access. For example, with an instruction length of 16 bits a 64 bit word would be read on each access and 4 consecutive instructions would be read. Requests to store therefore only occur on every fourth increment of the early call instruction address register. This reduces the traffic from the store and is one of the advantages of short register-register instructions. However, when branch order occur several wrong instructions will have been prefetched.

3.2.1 CDC 6600

Programs spend quite a lot of the time operating in small loops and if the instructions of the loop could be trapped in a buffer then there would not need to be any accesses to the store whilst the loop was being obeyed. This makes it easier to keep the pipeline supplied at the correct rate and also reduces the load on the store, thus increasing the probability that operands can be accessed rapidly. The CDC 6600 had a nice example of such a buffer, see Fig. 15, instructions flowed from the store through the early call buffer, then as well as going to the CPU execution they went into the instruction stack. Each new entry pushed the existing entries up by one place and the top word in the stack was lost. If a backwards branch occurred to an instruction still in the stack, then it could be read from the stack and if this in
fact then became a loop there would be no further accesses to the store until the program exited from the loop. The length of the buffer was such, that, eight 60 bit words could be trapped and this corresponded to approximately 20 instructions.

3.2.2 IBM 360/195

A similar, although more complex, buffer exists in the IBM 360/195. This is shown in Fig. 16. Under normal operation the buffer is kept full with eight 64 bit words ahead of the word containing the next instruction just entering the execution pipeline. When a conditional branch order is detected at the beginning of the pipeline further accesses to the store are stopped except to bring the two words at the branched to address into the special buffers. Execution proceeds along the non-branch path being careful not to do anything which cannot be undone, if necessary there is then a wait until the branch condition can be applied, then execution either proceeds along the non-branch path and requests to the store re-start or the execution switches to the orders in the branch buffer which are loaded into the main buffer, which is then filled up. For unconditional branch orders the branch to address is calculated at the beginning of the pipeline and the new sequence of instructions is brought from the store, the sequence in the buffer being rejected. The gap for unconditional branches is therefore A, for conditionals that branch the gap is P (assuming P greater than A) and for non-branching conditionals there is no gap unless the condition setting is delayed by waiting for some slow arithmetic, this delay X would of course also be added to the P for the orders which branch.

When a jump back of less than 8 double words occurs the buffer prefetches the orders of the loop and traps these in the buffer. No further accesses then occur until the loop terminates.

3.2.3 Trapping of loops

Both the 6600 and 360/195 will trap loops of the types shown in Fig. 17. i.e. nice simple loops. A "fractured" loop
such as in Fig. 17 will not be trapped even though the number of instructions in the loop are less than the length of the buffer. This is because all instructions in the buffer must be sequential. In the CDC 7600 the buffer also contains the addresses of the instructions, these addresses can be read associatively so that the instruction of a loop can be read from the buffer irrespective of its address. A loop such as in Fig. 17 will therefore be trapped. Note that although the loop is trapped there would still be gaps of $a$ or $P$ in the execution pipeline for each of the branches. ($a$ is the access time of the buffer)

3.2.4 MU5 - Predicted early call

From the point of view of removing gaps from pipelines the simple early call system would be ideal if it could be arranged that after requesting a branch order the next order was the one branched to and not the next in sequence. This at first sight appears to be impossible as at the time of the early call all that is know is the instruction address, the actual instruction will not come back from the store for $A$ nsecs and will not be obeyed for a further $P$ nsecs. However, if we can consider the block diagram in Fig. 18 and the instruction sequence in Fig. 19 then a solution will be clearly seen.

The associative memory is initially empty so that all instructions are brought in sequence and after $B$ comes $B + 1$, $B + 2$ etc. When $B$ is obeyed and found to jump to $C$ a gap of $A + P$ occurs and the orders from $C$, $C + 1$ etc., are brought. In addition to switching the early call counter to $C$ an entry $B + 1$, $C$ is placed in the associative memory. Similar actions take place at all the other jumps, so that by the time the program gets back to $A$ the contents of the buffer are:

- $B + 1$, $C$
- $D + 1$, $E$
- $F + 1$, $G$
- $H + 1$, $A$

and therefore the next time the early call tries to go for $B + 1$ the associative memory will give equivalence and the early call register will be reset to $C$ and a request for $C$ will be made.
Thus, only the correct orders are requested and these go down the pipeline with no gaps at all (except for possible X type gaps, see above).

If a branch order is not in the buffer store there will be a gap of A + P. If it is in the buffer there will be no gap. However, if a conditional branch which is in the buffer does not branch then the wrong instructions will have been prefetched and there will be a gap of A + P whilst the correct instructions are selected and enter the pipeline.

This attempt to predict branch orders has been implemented on the MU5 and it is hoped that approximately 60% of all obeyed branch orders will be correctly predicted.

This technique can be used in conjunction with the more conventional buffer techniques in an attempt to reduce the gaps when the prediction is incorrect.

3.3 **High speed arithmetic**

For scientific computing the operation to be performed on the operand when it finally reaches the arithmetic unit at the end of the pipeline is often in floating point and there are frequent occurrences of long operations such as multiply and divide. The time for these arithmetic operations can be many times the length of the basic pipeline beat so that if there was only a single arithmetic unit at the end of the pipeline there would be long hold-ups whenever a long arithemtic order occurred. Most reasonable sized systems have two arithmetic units in the pipeline, one for fixed point and one for floating point. The fixed point unit can keep up with the pipeline for most of its operation, so that if the floating point orders occur, as is generally the case, separated by some fixed point orders then the pipeline can keep going at full speed provided that the floating point unit has always completed its previous operation before it is asked to do the next. Whenever this is not the case there will be holdups in the pipeline. For very heavy computation problems the pipeline will often be held up with the simple double arithmetic unit
system described and in computers designed for scientific computation more extravagant techniques have been used.

In the CDC 6600 there are ten processing units, four being floating point arithmetic: one adder, two multipliers and one divider. The individual units are very fast and their designs will be discussed briefly.

3.3.1 Addition

The basic steps are:

1. load operands
2. compare exponents
3. shift one operand
4. add

The first, second and fourth parts are relatively easy to do in one beat (100 nsecs.) but the time for the third stage normally varies according to the length of the shift. By building a fast cascade shifter as shown in Fig. 20 this can be done in one beat whatever the shift so that the total add time is four beats. A block diagram of the full adder is given in Fig. 21.

3.3.2 Multiplication

This operation can be speeded up by three different methods:

1. splitting the multiplier
2. treating the multiplier digits in groups
3. using carry save adders

Splitting the multiplier is very simple and quite expensive, for example with a 48 bit multiplier this could be split into two 24 bit parts and using two multiply units, the two multiplications would be done at the same time and the results then added together. Further splitting into more parts will obviously further increase the speed but such extra splitting gets progressively less and less cost effective.

In simple binary multiplication each bit of the multiplicand \( d \) is looked at in turn and the multiplicand \( D \) is added
to the correctly shifted partial answer if the multiplier digit is a one. If before starting 3D is formed in a separate register then, 0D, 1D, 2D and 3D are readily available. The digits of d can then be decoded in pairs and the correct multiple of D selected. Treatment in threes or fours requires preforming and storage of more multiples of D.

The time consuming part of parallel addition is the propagation of the carry. In multiplication there are many additions done in sequence during the building up of the partial answer. Only at the end is the answer required in the normal form and for the intermediate stages carry save adders can be used. A carry save adder has three inputs W, X, Y and two outputs S, C.

$$S = (W \oplus X) \otimes Y$$
$$C = (W \oplus X) Y$$

Both S and C can be formed in two levels of logic, for all digit positions of the word at the same time. C is the carry which in a normal adder would have to propagate along the adder chain. For repeated additions, as in a multiplier, the S and C from the first add are two of the inputs for the next add together with the next product term. (D, 2D etc.). Thus all the addition times are very short (about 4 times less than a normal add) and only at the end must the final S and C be combined in a full adder to give the correct sum.

In the 6600 the 48 bit multiplier is split into two parts. The multiplier digits are treated in pairs and the additions are done in a bank of carry save adders as shown in Fig. 22.

A simplified diagram of the full multiplier is shown in Fig. 23. With this relatively extravagant structure the multiply time is ten beats.

3.3.3 Division

Division is a very difficult process to speed up. It is normally done by repeated subtractions from the remainder, and in the 6600 three subtractors are used so that \(-d, -2d, -3d\)
(d is the divider) can be tried in parallel. By decoding the subtractor outputs two digits of the quotient can be produced in one subtract time. Division is still very slow taking 29 beats. A block diagram of the divider is given in Fig. 24.

The 6600 has a register-register instruction format and there are 8 X registers which contain floating point numbers. The multiple arithmetic units are controlled by a piece of logic at the end of the pipeline which arranges for the correct operands to be sent from the X registers to the correct unit.

This logic also arranges for suitable holdups if the result of an unfinished operation is required as an input. With the multiple arithmetic units the 6600 is approximately twice as fast as a single arithmetic unit system using similar logic.

3.4 Pipelined arithmetic units

The stages of floating point addition were stated above, obviously these could be done in a four stage pipeline so that although each add time would be the same additions could start and finish four times as often.

This approach of pipelined arithmetic units has been used to some extent in the multiple arithmetic units of the 7600 and to a much greater extent in the arithmetic units of the CDC STAR system.

(Note - in the CDC 6600 adder, Fig. 21. EAB stands for end around borrow, it is the signal which indicates the larger exponent and it therefore controls which coefficient has to be shifted)
REFERENCES

Books on Computer Architecture


2) H. Lorin, Parallelism in hardware and software: real and apparent concurrency, Prentice-Hall Inc. New Jersey.


Papers on Computer Architecture


Fig. 1. Input/Output Wells

- Input Devices
- Input Well
- Computing Area
- Output Well
- Output Devices
Fig. 3. I/O WELLS WITH FRONT END PROCESSOR
Fig. 5. STORE ACCESS — PRIORITY AUTONOMOUS TRANSFERS
Fig 7 ATLAS PAGE ADDRESS REGISTERS
Fig. 8. ADDRESS DEVELOPMENT WITH SEGMENTATION
Fig 9 ASSOCIATIVE MEMORY FOR ADDRESS TRANSLATION
Fig 10 Shared Segments
Fig. II. COMMON SEGMENTS
Fig. 12. CACHE ORGANIZATION IBM 360/85
Fig. 13. DEGREES OF OVERLAP
Fig 14 GAP CAUSED BY BRANCH
Fig 15 CDC 6600 BUFFER
Fig. 16. IBM 360/195 BUFFER
Fig. 17: SIMPLE AND "FRACTURED" LOOPS
Fig. 18. MUS5 Early Call System
Fig 19 EXAMPLE OF COMPLEX LOOP
Fig 22 LOWER BANK OF CARRY SAVE ADDERS
Fig 23 CDC 6600 MULTIPLIER
(MANTISSA ARITH)
Fig 24 CDC 6600 DIVIDER (MANTISSA ARITH)
THE M.U.5 COMPUTER SYSTEM

F. H. Summer

The University
Manchester, U.K.

1. INTRODUCTION

In the design of the MU5 research computer, the aim has been to produce a high performance machine whose structure is well suited to the needs of modern high level languages. It is hoped that a computing speed improvement of about 20 over the 2-3 μS instruction rate of ATLAS will be obtained. In the ten years which have elapsed between the ATLAS and MU5 projects, the speed of logic gates and main storage has increased by a factor of 8 : 1, and this will result in a commensurate increase in system performance. In order to approach the 20 : 1 performance target, however, it will be necessary to adopt extensive parallel processing techniques, and to incorporate data buffering systems to compensate for the disparity between processor and storage speeds.

1.1 Philosophy of design

The order code of any computer reflects its internal structure, and since this is usually quite different from the structure of high level programming languages, it is difficult to compile efficient object code. For example, in order to approximate to efficient use of hardware features such as central registers, it is necessary for the compiler to analyse the control flow through large sections of program. Even then, compiled code, which is within a factor of two of optimum hand coding, is considered good, and much worse figures than this are considered acceptable.

Algol and Fortran are typical examples of high level programming languages, whose structure must be emulated in the machine design. These languages have the following features in common:—

(a) Each routine has local working space for storing named
variables of integer and real types, and for structured data sets (e.g. arrays) which are also named.

(b) A routine may refer to non-local names of either an enclosing routine or in a work space common to all routines.

(c) Statements involve an arbitrary number of operands, which may be real names, integer names, array elements, function calls or constants.

Algol is distinguished by the recursive use of routines and dynamic storage allocation, while Fortran uses non-recursive routines with compile-time static storage allocation.

Data processing languages, such as Cobol, differ from the scientific languages Algol and Fortran in their data structures and the kind of elements these contain. In Cobol, the variability of elements is such, that it is usually necessary for the compiler to treat them as structures.

Each of the high level languages mentioned demands from the programmer information about the types of operand he intends to use. In the MU5 design, this information is made available to the computer hardware to increase the efficiency of its own store organisation.

The types of operand commonly in use in high level languages may be listed as follows:-

(a) Literals consisting of fixed-point and floating-point constants.

(b) Primary operands consisting of:-
   (i) integer and real variables
   (ii) stacked operands
   (iii) data descriptors
   (iv) names of functions and routines

(c) Secondary operands consisting of:-
   (i) elements of arrays and vectors
   (ii) strings of elements of structured data sets.
Literal operands are specified explicitly by the instruction, and may be of length 6, 16, 32 or 64 bits. Primary operands with the exception of stacked operands exist in the name space of the current procedure, the start of which is defined by the value set in the Name Base Register (NB). The Virtual address of a primary operand is obtained by adding its name 'n' within the procedure, as specified by the instruction, to the value of NB. Stacked quantities are normally read from the front of the Stack specified by the Stack-Front Register (SF). It is, however, possible to access quantities within the stack by specifying a displacement behind SF. Reading a stacked quantity causes SF to be decremented, while each time a quantity is stacked it is incremented. This makes the stacking action similar to that of a hardware push-down stack, although the stacked quantities are held in normal storage. In the MU5, this stacking mechanism is used both for evaluating expressions and for communicating parameters between procedures. When a Secondary operand is to be accessed, the Primary operand specified by the instruction is nominated a Descriptor. The Descriptor specifies the 'origin' or base address of a data structure, the type of element it contains, and the upper limit of the structure. The displacement of the required data element from origin is given by the contents of an index register, which may be added to the origin to give the Virtual address of the required element. The method of addressing both primary and secondary operands is shown schematically in Fig. 1.

In the case of static (compile time) storage allocation, a compiler would allocate space to the names of each procedure and set up a directory giving their name bases, which would be used to set NB when a routine is called. With dynamic (run-time) storage allocation, the name base of a procedure is unknown until the procedure is called. The name space of a called procedure is set up beyond that of the calling procedure, starting at the current value of SF, which indicates the next available location in the name segment. Parameters for a procedure to be called, can be
stacked and included in the name space of the called procedure by setting NB to the appropriate position behind SF when the procedure is called. SF is then set to the end of the name space of the called procedure. On exit, SF is set to the value of NB and NB is restored to its original value, which was preserved in the link, which would have been stacked when the procedure was called. The action is shown diagrammatically in Fig. 2, where procedure M calls procedure N.

An analysis of a large number of programs run on the Manchester University ATLAS computer showed that 80% of operand accesses were for operands which would be classed as names in the proposed system. A further study showed that the number of names in use at any time is relatively small, and that an associatively addressed store, with as few as 32 lines, would trap over 99% of these operand accesses in the majority of programs. The system therefore derives considerable benefit from the concept of naming, since 4 out of 5 required operands can be obtained from a small associatively addressed slave store, which could cycle every 40-50 nS instead of requiring a Main Store access. The importance of this can be appreciated when the overall access time for the Main Store is considered. The Main Store of the MU5 Central Processor comprises four stacks of 4K words (72 bit), with a cycle time of 250 nS and an access time at the stack of 130 nS. However, an elaborate system of Virtual addressing and paging is included in the system and this, together with priority circuits and cable delays, results in an overall access time of about 600 nS. The benefit of trapping 80% of operands in the fast 'Name Store' is obviously great. The remaining 20% of operand accesses are for array quantities whose access patterns are essentially sequential. The system used to access this type of operand is considered at length in subsequent sections.

1.2 The instruction format

A single address format has been adopted for the MU5 computer. It is believed that this type of code, with no explicitly addressed
registers, but with a fast associatively addressed name store, will permit the writing of efficient compilers which produce efficient object code. The proposed code adopts some of the more valuable features of a stacking machine by including special load orders which first stack the Accumulator contents, and a means of addressing stacked partial results.

MU5 instructions are of variable length and may utilise 16, 32, 48 or 80 bits. The two longer types are needed to specify 32 and 64-bit literal operands. The Basic MU5 instruction is of 16 bits with the format F (10) n (6). F specifies the central register to be used, the function to be obeyed and the operand type. The n part defines the name of the operand in the current routine, or may itself be a short literal operand. The basic 16-bit instruction can specify only a limited range of names, or a short literal. An extended range of names can be addressed by specifying the next 16-bit instruction unit as the displacement from the Name Base. Long literal operands are obtained by specifying the next 1, 2 or 4 16-bit instruction units as the literal operand.

The first three bits of the F field (CR bits) define the central register used by the instruction; these include the fixed and floating point accumulators, the index arithmetic unit and the store-to-store operation unit. If these bits are zeros, however, then organisational action is specified and the instruction is interpreted differently. The two instruction formats determined by the state of the CR bits are shown below.

<table>
<thead>
<tr>
<th></th>
<th>CR</th>
<th>f</th>
<th>k</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Computational and store-to-store instructions</td>
<td></td>
<td></td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>b) Organisational Instructions</td>
<td>CR = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CR</th>
<th>f'</th>
<th>k</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CR</td>
<td>f'</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

MU5 Instruction Formats

(a) is the format for computational and store-to-store orders, and the f field describes the function to be obeyed. The k bits
determine the type of operand, which may be a short literal, a 32-bit variable (Integer), a 64-bit variable (Real), or a 64-bit Descriptor, which specifies an array. If $k = 7$, however, then additional 16-bit instruction units are required, and the $n$ fields of the instruction is decoded to specify the type of the 'extended operand'.

(b) is the format for the Organisational instructions, and a longer field ($t'$) is used to specify the range of instructions. These instructions include the various conditional jumps and the instructions which manipulate the base registers, such as NB. In this format, the $K$ field is one bit only; if zero, then the $n$ field is a short literal, if not, then an 'extended operand' is specified and the $n$ field is decoded to give the operand type. The use of the extended operand format allows access to stacked quantities by use of SF as base, and to variables which are not local to the current routine by use of the Extra Name Base register XNB.

1.3 The addressing structure

The MU5 computer system incorporates a segmented Virtual store which enables several large program or data areas to be kept, and these areas may grow independently. This facilitates the writing of efficient software, and reduced the storage demands of processes which may share segments of store containing read-only data or program. A formal segmentation system may apply selective lock-outs to segments so that a read-only data segment might have an obey lock-out and a write lock-out, and a program segment might have both read and write lock-outs. The processor makes its store requests in terms of the Virtual addresses generated by the program. The Virtual addresses are translated into real store addresses by the store access control of the system.

The system can accommodate up to 16 active processes at any time and the process currently under execution is specified by the number set in a Process Number register. Each process may address 16K segments and each segment addresses 64K words (32-bit).
Segments are divided into blocks or pages which are variable in size from 16 words to 64K words in powers of 2. Hence the format of a Virtual address produced by the processor is as shown below.

If the most significant segment bit as a 1, then the process number of the Virtual address is ignored and access is always made to the segments of process Zero. This is a special case of shared segments where these 'common' segments are available to all users. They contain constantly used data, such as compilers, library routines, etc.,

The software part of the store access control system consists of a table structure describing the address mapping, access lock-out status and real store utilisation. Also included are the procedures operating on this table structure, which organise the movement of pages and the loading of Current Page Registers (CPRs). The CPRs are associatively addressed hardware tables which contain a working sub-set of the software tables to allow rapid access to the real store of the system.

2. **THE MU5 MULTI-PROCESSOR SYSTEM**

2.1 **Introduction**

The MU5 computer system is a multi-processor complex with processors and store units linked by an interconnection system, the Exchange. The computational power of the system is provided by the MU5 central processor, while a modified ICL 1905E processor provides the front end to the system. Each processor has its own dedicated
fast store; a Mass core store and disc system provide backing storage. The interconnection system has been designed so that additional processing or storage units may be added without modification to the existing system hardware. The various units of the MU5 computer system, linked by the Exchange, are shown in Fig. 3.

2.2. The exchange

The Exchange system is basically a multi-port OR-gate, with an input from each of the system's units and an output buffer connected back to each unit. All data transfers between units are made via this OR-gate, which is time-shared between requesting units. This single channel interconnection is acceptable, provided that its data rate is sufficiently high to meet the demands of the interconnected units. The Exchange is designed to operate with a 100 nS time slot, and it is not anticipated that this should impose any serious restriction on the system. The Exchange system has been built with 10 ports, but the capability exists to expand to 12 or 16.

2.3 The ICL 1905E Processor

This utilises a 24-bit word length and the particular machine is equipped with a 24D core store, cycle time 650 nS, capacity 32K words. It is equipped with the usual range of input and output peripherals and with two exchangeable disc drives, each disc cartridge having a capacity of 2 million words. The standard machine has been modified to incorporate a paging unit, which allows the addressing structure of the 1905E to approximate to that of the MU5 system. This simplified the use of the machine for the development of the system software and facilitates the connection of the processor into the MU5 Computer complex via a standard Exchange port. Once part of the overall computer complex, the machine fulfils the role of a front-end-processor, which provides the normal input and output facilities for the system.

2.4 The mass store

The Mass Store provides the first level of backing store to the fast stores of the system processors. The Mass Store currently
in use is a 2\(\frac{1}{2}\) uS cycle time, 2\(\frac{1}{2}\)D core store, with a capacity of 128K 72-bit words, each word consisting of 64 data and 8 parity bits. The store is made up of two 128K 36-bit stacks, which normally operate simultaneously and provide 72-bit operation. A fail-soft facility is provided, however, whereby the system may function with only one stack; in this mode, two cycles of the stack in use are required to read or write a 72-bit word, and the speed is correspondingly degraded. Each stack contains a self-test facility which may be activated manually or via the \(V_x\) store of the Mass Store system to provide fault diagnosis.

2.5 The block transfer unit

The Block Transfer Unit (BTU) is a processor type device, in that it may initiate requests to the Exchange. It is connected to a single Exchange port, and it may organise the transfer of data blocks between stores via buffers within the BTU. Its normal use is for the transfer of data blocks between Mass Stores and Processor Local Stores but it can be used for Mass to Mass or Local to Local transfers, if required. It is not used for the block transfers of the MU5 Disc system which contains equipment to organise its own data transfers.

2.6 The MU5 disc systems

The second level of backing store of the MU5 computer system is provided by a Disc system incorporating head-per-track disc units. The system at present contains two disc units but up to four may be accommodated as required. Each disc unit contains four twelve inch discs with 64 data tracks per surface, giving a total of 512 tracks per disc. Data is recorded at a 450 nS bit interval which gives a capacity of \(3 \times 10^5\) words per disc unit where a word is 64 data + 8 parity bits. The discs rotate independently, at 3000 revolutions per minute, which gives a revolution time of 20 milliseconds and an average access time of 10 milliseconds.
Transfers to and from the Disc system are made via the Exchange, and the required data width is 64 data and 8 parity bits. The data rate from a disc unit must not exceed that of the slowest device to which it must transfer. This is the Mass Store in the MUS5 configuration and hence the disc rate must not exceed 2.5 $\mu S$ per word. The physical constraints of the head selection system, dictate that the 64 data tracks may only be divided on a binary basis. This results in an optimum arrangement when eight disc tracks are read in parallel, and the 72 data bits are assembled in 9 x the 450 nS bit-rate. This provides a 4.05 $\mu S$ word-rate which can be accepted by the Mass Store system.

3. THE MUS5 CENTRAL PROCESSOR

3.1 Introduction

The MUS5 Central Processing Unit (CPU) utilises a pipeline concurrency technique to obtain a fast instruction-processing rate. This requires the use of separate hardware units to implement the various operations involved in the execution of an instruction. Hence at any time a number of instructions may be in the 'pipeline', at varying stages of execution. The overall time for an individual instruction to be executed is not changed by this technique. However, the rate of execution of instructions may be greatly improved, since a number of instructions are executed concurrently.

The MUS5 Central Processor may be divided into a number of functional units which together make up the processor pipeline. This is illustrated in the block diagram of Fig. 4. Each unit is fully buffered and may itself consist of several pipelined stages.

Instruction data is accessed from the Main Store by the Instruction Buffer Unit (IBU). This transfer from Main Store is made via the Store Access Control System (SAC) which organises all transfers between the processor and the store system. Instructions pass from the IBU to the Primary Operand Unit (PROP), which performs the initial decoding and processing of the instruction. Instructions destined for the B-Arithmetic Unit (B-Unit) which
specify a primary operand, will be transferred directly from the PROP to the B-Unit. Other instructions will follow the Main Processor Pipeline and will be transferred to the DR Unit of the Secondary Operand system (SEOP). DR computes the address of the secondary operand, when an indirect access is specified by the instruction. The input to the Operand Buffer System (OBS) is the function part of the instruction together with an operand address or operand. The OBS system queues up functions while access is made to the Main Store for required store words. Functions and store words are transferred to the DOP unit of the SEOP system, which performs any selection and shifting needed to extract the required operand from the store word or words. The functions will leave the DOP unit, together with its operand. This will be transferred to the destination specified by the function, and may be the Accumulator unit (ACC) or other central register. Subsequent sections describe the operation of each of the units of the Processor Pipeline, which together attempt to maintain a flow of instructions to the Main Arithmetic Unit or Accumulator.

3.2 The instruction buffer unit
3.2.1 The problem of control transfers (Branch Orders)

At the time of its design ATLAS was one of the fastest computers in the world and there was a considerable amount of overlap of the various stages of instruction execution in order to achieve the speed. However, it was decided that all test codes and other orders which may or may not transfer control would be taken to completion, this takes about 7 sec., before the request for the next instruction was made to the store. This simplified the design and at the time it was not felt that the effects would be significant.

After the machine was completed various tests of its speed were made using long sequences of instructions and it was concluded that the average time per count was 2 µsec. Later observations of the number of orders obeyed in a day gave instead an average time of 3 µsec. The discrepancy was originally put down to store clashes
but when a later version of the system with a much larger store gave the same results, it was concluded that the overlap in practice was not as good as had been expected, but no particular reasons could be put forward.

In order to collect data to help in the design of MU5 modifications were made to the ATLAS to permit the collection of statistics on the number of orders of the different types which were obeyed. These showed that over a very large sample of programs, the proportion of orders which may transfer control was about 20%. This clearly explains speed of 3 μsec., 80% of the orders will be overlapped and started every 2 μsec. but for 20% of the orders there will be no overlap and the gap before starting the next will be 7 μsec.

\[ 0.8 \times 2 + 0.2 \times 7 = 3 \text{ μsec}. \]

The experiment also yielded the distribution shown in Fig. 5 for the number of orders between transfer type orders for quite a large sample (several minutes of computing).

The general reaction by programmers to this graph was one of disbelief. It is, however, interesting to analyse the orders executed in obeying a simple piece of Atlas Autocode, Fig. 6 which shows 23 test or jump orders in a total of 78 obeyed instructions.

Other statistics provided by the Atlas monitoring are summarised below.

Results from monitoring the Manchester University Atlas.

Percentage of obeyed orders which may transfer control \( 20\% \)

Percentage of these which actually cause a transfer \( 70\% \)

14% or 1 in 7 orders transfer control

Distribution of transfer orders

<table>
<thead>
<tr>
<th>Type</th>
<th>Obeyed</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional transfers</td>
<td>5.5%</td>
<td>5.5%</td>
</tr>
<tr>
<td>Test and count</td>
<td>5.0%</td>
<td>3.9%</td>
</tr>
<tr>
<td>Arithmetic tests</td>
<td>9.5%</td>
<td>4.6%</td>
</tr>
</tbody>
</table>
Distance of jumps

\[
\begin{align*}
34\% & < 4 \\
6\% & = 4-16 \\
60\% & > 16
\end{align*}
\]

3.2.2 The prediction of branch orders

This has been described briefly in the notes on computer architecture (section 3.2.4)

Only branch orders with literal operands are involved and as all the code is in pure procedures any such branch must always go to the same place. The extension of the system to include branch orders which read their destination from an alterable store location is quite straightforward. The main omissions of the present system are sub-routine calls and return links and also multiway jumps. It is unlikely that the latter could ever be correctly predicted and therefore they should always be excluded, however the inclusion of subroutine calls and return links could be very valuable.

The results of tracing several programs suggest that at least 60% of the branch orders should be correctly predicted.

3.2.3 The MU5 instruction buffer unit

The block diagram of the Instruction Buffer Unit (IBU) is shown in Fig. 7. The IBU takes advantage of the double word read facility to access instructions 8 at a time, \((8 \times 16\text{-bits})\). The word-pair accessed is deposited in an 8 instruction buffer in the IBU. The selection from the input buffer is controlled by information supplied by the Unpack-Stack. When a request is made for a word pair, information is entered into the Unpack-Stack, which specifies the positions of useful instructions within the word pair to be accessed. When the accessed data is deposited in the input buffer, the information from the Unpack-Stack is used to select the useful instructions. These are transferred to the Close-Packed-Buffer prior to being transferred to the Primary Operand Unit (PROP).

The jumped-from table comprises eight lines of associatively
addressed memory. This is used to store the jumped-from instruction addresses of the last eight control transfers which took place. The jumped-to table is constructed from integrated, random access memory, and is used to contain the eight corresponding jumped-to instruction addresses for the same control transfers. The entries in the jump-Trace are made using the value of Control in PROP before and after the control transfer takes place.

The instruction Buffer operates by making a request for instruction data as soon as it may be accommodated in the input buffer. The instruction counter will have counted through the addresses of the last eight instruction units requested, and it will have been compared with the jumped-from table at each increment. If no match has been found, the next instruction word-pair requested will follow the last in sequence. If, however, a match has been found, then the word-pair containing the out-of-sequence instruction must be accessed. The address of this word-pair will be obtained from the line in the jumped-to table, corresponding to the matching jumped-from line.

An in-sequence bit accompanies each instruction unit supplied to PROP; this indicates whether an instruction sequentially follows the last. This is required by PROP when a control transfer instruction has been encountered, to determine whether the next instruction supplied by the IBU is the correct one to execute next.

3.3 The primary Operand Unit

3.3.1 General

The Instruction Buffer Unit (IBU) is designed to provide a steady stream of instructions from the Main Store, with the minimum of time lost due to Main Store access delays. The Primary Operand Unit (PROP) performs the initial processing on instructions, and must be designed to take advantage of the enhanced flow of instruction data from the IBU. This requirement has led to the adoption of a 'pipeline' system in PROP and throughout the whole of the MU5 processor. Using this system, the various operations in the execution of an instruction are separated into distinct stages, each
of which is buffered. When an instruction passes from one stage to
the next, the former is free to accept the next instruction, and
hence each stage may contain an instruction in varying stages of
execution. If the stage delay is matched to the flow of instruction
data from the IBU, then the minimum time will be lost by one unit's
waiting for the other. This simple scheme is complicated by a
multiplicity of special actions which prevent full advantage being
taken of the instruction overlap. However, the philosophy through-
out the design of the machine has been to approximate to this
totally pipelined concept as closely as possible.

3.3.2 The design of the MU5 primary operand unit

The block diagram of the Primary Operand Unit is shown in
Fig. 8. It consists of five overlapped stages with instructions
supplied by the IBU in 16-bit units, entering the input buffers
DF/DN. Instructions are normally 16 or 32 bits long and are
buffered in DF, or DF and DN, respectively. Instructions specifying
literal operands may extend to 48 or 80 bits and these will be
assembled in registers L1, L2, L3 as the instruction proceeds down
the pipeline. The PROP incorporates a fast, associatively addressed
slave store, known as the PROP Name Store. This is used to buffer
certain classes of primary operands accessed by PROP for use by the
B-Unit or DR system. The associatively addressed field of the Name
Store occupies the third stage of the PROP pipeline and the value
field of the Name Store occupies the fourth. These are separated
by a line register (LR) which enables the associative field to be
interrogated by the contents of the Interrogate Register (IN), while
a line in the value field, selected by LR, may be read.

The action of the PROP pipeline in processing a simple
instruction is as follows. The first stage decodes the instruction
and selects the appropriate base register NB, XNB or SF. The
appropriate base is added to the displacement specified by the
instruction in the second stage, producing the primary operand
address which is placed in the Interrogate Register (IN). The
contents of IN are presented to the associative address field in Stage 3 and a line giving equivalence sets a bit in the Line Register (LR). This is used to extract the corresponding line from the value field in Stage 4. Stage 5 performs selection between the output of the value field in VF and the Literal Assembly Register VU, before performing any shifting necessary to align the operand and depositing it in HI. The instruction is then ready to leave PROP for its destination, and the instruction count or 'Control' is normally advanced once the instruction has left PROP.

3.3.3 The PROP name store

The PROP Name Store is used principally for Named operands accessed for the B-Unit or DR system. These will be the indices and counts manipulated by the B-Unit, and the Descriptors which are processed by the DR system to address structured data arrays. The PROP Name Store is not normally used for Named operands specified for Accumulator instructions. This is because a large number of pipelined stages divide the Accumulator and the PROP Name Store, so that the loss of instruction overlap for an Accumulator write to store would be unacceptable. Named operands specified by Accumulator instructions are normally stored in the Name part of the OBS Buffer Store, which is much closer to the Accumulator. The PROP Name Store and the OBS Buffer Store together form the lowest level of the MU5 store hierarchy, and transfers of data between these buffer stores and the Main Store are performed automatically when required.

The Name Store comprises the address field which is associatively addressed and the value field, which is constructed from integrated, random access memory circuits. The address field contains no segment bits, so that at any time the Name Store may buffer operands from a single segment only, this being specified by the entry in the Segment Number Register. The Value field entries are 64 bits, and correspond to a single word accessed from Main Store. The rate of operation of the Name Store is about 50 nS and
this sets the limit on the rate at which instructions may be processed by the PROP pipeline. The PROP pipeline may process instructions at its maximum rate, if operands which it is required to access are found within its Name Store. When a required operand is not found, then a transfer from Main Store must be made; this results in an additional delay of at least 800 nS. If 1% of instructions require a Main Store access, then the average processing rate would drop to 58 nS, whereas 5% would cause it to drop to 90 nS. Simulation studies were carried out on the Atlas computer, and these indicated that a 32 line Name Store in the PROP should result in accesses to Main Store for about 2% of instructions. This size of Name Store was adopted and hence a degrading of rate to 65 nS might be expected. In practice, however, multiple length instructions and instructions which destroy the overlap also have a degrading effect on the rate, and the effect of all these has been estimated to reduce it to 120 nS average.

3.3.4 The position of control

In a simple computer the instruction address register or 'Control' is the address of the instruction currently being processed; this would normally be incremented when it is required to access the next instruction to be executed. In a pipelined computer, however, several instructions may be concurrently under execution and it is not obvious which of these should be addressed by Control. When an interruption causes a program transfer, the value of Control provides a return link into the interrupted program. On interruption, instructions prior to the Control point are abandoned and will be re-accessed when the program is re-entered. Instructions which have passed the Control point, however, will not be re-accessed on program re-entry and hence their execution must be guaranteed.

Only a limited set of instructions obtain their operands within the PROP pipeline, others leave the end of PROP as a function with a Descriptor or an operand address. If the operand address from PROP or the address computed from the Descriptor causes non-
equivalence when presented to the current page address registers, then the operand will not be available to allow execution of the instruction. Operand availability could be guaranteed, by preventing the advance of an instruction beyond the Control point until its operand address had obtained CPR \( \equiv \). In the present system, this would cause a delay of about 350 nS, which could not be tolerated for Accumulator instruction, normally obtaining their operands outside the PROP system. If Accumulator instructions are to pass the control point, when there is a possibility of a CPR \( \not\equiv \) preventing their execution, then a facility must exist to preserve unexecuted instructions. This facility is provided by a function well later in the instruction pipeline. When a CPR \( \not\equiv \) occurs, any instructions which have passed the control point but may not be executed, are trapped in the well. From here, the non-executed functions, together with their operand addresses, may be preserved in Main Store when a program transfer is made. This system is not adopted for non-Accumulator instructions, however, since these normally obtain their operands within the PROP system. Instructions of this type which could cause a CPR \( \not\equiv \), are prevented from advancing beyond the Control point until a CPR \( \equiv \) indication is obtained.

Moving the Control point as near to the CPRe as possible would minimise the delay suffered by an instruction awaiting CPR \( \equiv \). The Control point is positioned at the last stage of the PROP pipeline. This position is chosen because the processor pipeline splits when it leaves PROP, to supply instructions to either the B Unit or DR system. If the Control point were moved beyond the end of PROP, then its management would be greatly complicated by the bifurcation.

3.4 The secondary operand system

3.4.1 General

The Secondary Operand system (SEOP) is concerned with the manipulation of data which has some well defined structure. A common example is data which is declared as an Array in a high level language such as Algol, but there are, of course, numerous other examples of data with a definite structure. Certain properties of
this type of data influence the design of an accessing system. Elements of a data structure, or Vectors, are usually accessed sequentially, but accesses to individual Vectors are infrequent. Hence the system adopted for the access of Named operands, which relies on the frequent use of a relatively small number of operands, would not be suitable for Vectors. The calculation of the address of a Vector is often complicated, and since it may have to be calculated for each Vector of a structure, it would be advantageous if at least some of the calculation could be performed automatically. Hence the design of an accessing system for Vectors should aim to automate the generation of addresses as far as possible, and should be geared to accessing sequential data from the Main Storage of the processor.

3.4.2 Data descriptors

The address of a Vector quantity is calculated from a Descriptor which specifies the format of the particular data structure. There are several types of Descriptor, and Vectors varying in size from 1 to 64 bits may be accessed without regard to store word boundaries. An instruction requiring a Vector operand must specify a Descriptor as its Primary operand. This Primary operand is accessed by PROP, and supplied to the DR Unit with the function part of the instruction. The DR Unit forms part of the SEOP system, and its action is to compute the address of the required Vector using the Descriptor, and a Modifier which is supplied from the B-Unit when required. The address of the store word containing the Vector, is normally generated by using the value of the Modifier as a displacement from the base address, specified by the origin field of the Descriptor. The store word address and the function are transferred to the Operand Buffer System (OBS), which accesses the required store word. In general, it is necessary to perform some selection on the store word to extract the specified Vector. This is performed by the DOP Unit which, together with the DR Unit, makes up the Secondary Operand System. The selection of the Vector from the requested store word is specified by a set of
control bits generated by DR, and known as the DOP-bits. These DOP-bits accompany each function through the OBS system to the DOP unit, where they are used to perform the selection on the store word accessed by the OBS. The selected operand and the function, will then be transferred to the appropriate destination for the execution of the instruction to be completed.

3.4.3 Descriptor types

A Descriptor word is split into several fields, the most significant two bit field specifying the Type of the Descriptor, which defines how the remaining fields are interpreted. The basic Descriptor types are shown in Fig. 9 and these will be discussed separately.

3.4.3.1 Type 0 Vector descriptor

A Type 0 Descriptor, as shown in Fig. 9(a), is used for the access of single Vectors of size 1 to 64 bits, from a data structure. The origin field of the Descriptor defines the base of the data structure, and the Bound field defines its extent. The origin field may be used to access a Vector directly but, in general, it will be modified by the addition of a 32-bit integer, supplied from the B-Unit. The Modifier is used as a displacement in bytes, or in data elements, determined by the setting of the scaled/not scaled bit in the Descriptor. Scaling is always required when accessing 1 or 4-bit Vectors because the origin field of the Descriptor addresses to the byte-level only. The bound field of the Descriptor is used as an upper bound for modification; the Bound field is shorter than the Modifier because of the limited space available within the Descriptor, but this is not considered to be a significant disadvantage. The lower bound is assumed to be Zero and the check: Bound > Modifier > 0 is applied, unless inhibited by the Bound-Check-Inhibit bit in the Descriptor.

All Vectors are assumed to be uniformly aligned with respect to a 32-bit word, e.g. a 16-bit Vector must start at a 16- or 32-bit boundary, and not on an intermediate 8-bit boundary. A 64-bit
Vector may span two 64-bit store words, since it may be aligned at any 32-bit word boundary. When this occurs, two Main Store words must be accessed in order to extract the 64-bit Vector.

3.4.3.2 Type 1 string descriptors

A Type 1 Descriptor, as shown in Fig. 9(b), defines a string of bytes. The origin field specifies the first byte, and the Bound field is interpreted as the number of bytes in the string. Scaling of the Modifier is not relevant for this Descriptor, since only bytes are involved. It is possible with this Descriptor to specify a string of bytes which cross a 64-bit word boundary, as in the case of a 64-bit Type 0 Vector.

3.4.3.3 Type 2 'descriptor' descriptor

A Type 2 Descriptor is identical in format to the Type 0 Descriptor. By convention, this Descriptor defines a data structure in which the Data elements are themselves Descriptors. Such Vectors may be used at the nodes of a tree structure. Each node contains pointers to lower level nodes but may have only one pointer to it from a higher level node.

3.4.3.4 Type 3 descriptor - various sub-types

The format of the Type 3 Descriptor is shown in Fig. 9(c) where it can be seen that the sub-type field defines the various Type 3 sub-types.

(a) Type 3.0 real-address descriptor

This Descriptor is treated similarly to the Type 0, with the element size assumed to be 64 bits. The Type 3.0 Descriptor is unique, however, in that the address computed is used to address the physical store directly, without reference to the Current Page Registers. This Descriptor is used, solely by the Operating System of the processor, and is used for tasks such as organising the Virtual Store Addressing system.

When a Type 3.0 Descriptor is used, the physical store itself is always accessed. No entry is made in the lower-level
buffer store of the Operand Buffer System (Section 5.2.8).

(b) **Type 3.1 Read/Store-Direct Descriptor**

This Descriptor again specifies a 64-bit Vector and produces a normal Virtual Address. This sub-type, however, requires that the Main Store system should be accessed directly, bypassing the lower level store of the Operand Buffer System. This Descriptor is again used solely by the Operating System for its organisational tasks.

(c) **Type 3.2 Read-and-Mark Descriptor**

This Descriptor specifies a 64-bit word which may be read from Main Store, bypassing the buffers of the Operand Buffer System. This sub-type causes a special action to be taken by the Main Store, however, the location read being left in all zeros state. This operation is required for the organisation of shared store in a multi-processor environment. A processor may check whether a section of store is available to it, by accessing a Key-word for this section, using a Read-and Mark Descriptor. If the word read is found to be zero, then the section of store is already in use by another processor and the Key-word must be read again after some suitable delay. When the value read from the Key-word is found to be non-zero, then the section of store is available to this processor and, having been set to zero by the action of the Descriptor, will now be locked out to other processors. When the processor has completed its operation, then the store section may be released to other processors by writing to the Key-word, using a Type 3.1 Descriptor.

(d) **Type 3.3 Indirect Descriptor**

The Indirect Descriptor provides a facility for chaining, in that it specifies a further Descriptor to access the Operand. This further Descriptor may itself be an Indirect type, so that the chaining could continue indefinitely. The replacement of the Descriptor is performed automatically by hardware; if modification is specified, this is applied only to the final Descriptor, which will not be of the Indirect type.
(e) **Type 3.4 Procedure-Call Descriptor**

This Descriptor is used when replacement of a formal parameter on entry to a procedure involves the evaluation of an expression. The Descriptor is used to access the first 32-bits of a Vector, which is the starting address of code to evaluate the expression. The code leaves the value in a suitable location and alters the Procedure Call Descriptor to Type I format, with an address pointing to the value. Control is returned to the instruction causing the procedure call, and this time the value of the formal parameter is accessed.

### 3.4.4 Description of the DR unit

The DR Unit forms part of the Main Processor pipeline between the instruction Buffer Unit and the Accumulator. When an instruction specifies a Vector operand, the Unit performs address calculation and Bound checking on the appropriate fields selected from the descriptor. If, however, some other operand is specified, then the DR Unit must act as a passive buffer to the instruction. Hence a function transferred from DR to the OBS system may be accompanied by a Vector address computed by DR, a Name address produced by PROP or a literal operand supplied by PROP.

### 3.4.5 The OBS unit

This unit is between DR and DOP and it either requests the word from store as specified by DR or it passes on the operand supplied from DR. In both cases the operand goes to DOP and then to the arithmetic unit. The function from DR passes via an eight stage queue or pipeline to DOP and then to the arithmetic unit. This permits up to 8 functions to be queued up waiting for operation in the arithmetic unit, and whilst the function is moving down the queue there will probably be sufficient time to obtain the required operand from the store. There is thus far less need to try and trap these operands in fast buffers. However, eight lines of buffer are required for each type of operand, array, name or literal, as there could be 8 functions in the queue each with the same type of operand, and there has to be somewhere for each of these to be stored as the arithmetic unit could be busy for
sufficient time for all 8 operands to be brought from the store. Any operands which can be trapped in buffers in OBS will be an advantage as this will reduce accesses to the store and will also decrease the chances of delays in operands being supplied to the arithmetic unit.

The configuration of the buffers is shown in Fig. 10. Any literal operand goes to one of the 8 lines of literal buffer and the tag generator forms a pointer which joins the function in the queue and is later used to select the operand for transmission to the arithmetic unit.

The virtual address of an array element is presented to an 8 line associative store and if equivalence occurs the tag goes to the queue. If not equivalence occurs one of the 8 lines of the 128 bit wide buffer is selected and emptied if necessary. The tag is sent to the queue and the tag and virtual address are sent to the store. The required operand together with the other half of the 128 bit word is brought from the store and placed in the buffer. This buffer is two words wide as the main store is also two words wide and it is economical to bring two words if there is a reasonable chance of the second being used. This is quite likely for array quantities which tend to be scanned sequentially. Traces of the array operand accesses for several programs indicate that this small buffer could have a hit rate of approximately 60 per cent.

For named real quantities the action is the same except that only single words are brought from the store, this buffer was made as big as possible, regard being paid to both economic and space considerations. Twenty-four lines were the most that it was possible to include and with this configuration the hit rates for the named real operands were between 75% and 99% for several traced programs. These hit rates are very satisfactory for such a small buffer and in this situation misses do not necessarily incur a penalty as the function is quite likely to be delayed in the queue, whilst the arithmetic unit is busy, thus permitting the operand to be read from the store and placed in the buffer in a similar manner.
to that used for array elements.

The restriction of array elements to a particular buffer should permit the using of a buffer which makes considerable use of the sequential mode of addressing of most arrays. A range of such buffers was studied and whilst it was not possible to include any of these designs in the current version of the MU5 it is of interest to note that hit rates in excess of 90% were obtained on a 16 line two word wide buffer in which a system of predictive loading was employed. This is a further proof of the value of permitting the structure of the hardware to match the structure of the data of the programs being obeyed.

Having the name store in two parts causes some difficulties. The names are all in the same segment and it is sometimes necessary to move named operands from one buffer to the other. This is time consuming but as it occurs very infrequently, it should have very little effect on the overall performance. A possible development would be to have two segments of the virtual address space for names, one for indices and descriptors which would be trapped in one buffer and a different segment for the real named quantities.

3.4.6 The DOP unit

The DOP Unit follows OBS and performs the required selection from the store word supplied by OBS. Any shift from 0 to 63 bits may be made. The shift mechanism, in conjunction with a masking facility, enables any data element to be extracted and aligned at the least significant end of the 64-bit field at the output of the shifter.

DOP then passes the operand and the function to the appropriate arithmetic unit. For write to store orders DOP performs the reverse operation of correctly placing the operand from the arithmetic unit into the 64-bit word before returning it to OBS.

3.5 The store access control system

The Store Access Control System (SAC) forms the interface between the MU5 processor and the Store complex of the MU5 computer
system. It connects directly to the dedicated Local Store and via
the Exchange, to the other units of the computer system. These
other units include the Mass and Disc Stores, as well as other
processors in the complex.

SAC contains the Current Page Registers (CPRs) which
convert the Virtual addresses used within the processor into Real
addresses, which are required to access the physical stores. The
CPRs contain a currently used sub-set of the Main Store tables,
which map the Virtual Store of the computer system into the avail-
able Real Store.

3.5.1 The current page registers

The Current Page Registers (CPRs) comprise a 32-line
hardware table containing Virtual and Real block (or Page)
addresses. The Virtual address table is constructed from associati-
ve storage modules which provide a content-addressed facility.
Integrated, random access memory modules are used for the Real
address table, which provide conventional read-write storage.

Translation from Virtual to Real addresses is performed
at the block level, the block size being variable from 8 words to
32K words (64-bit) depending on the particular application. The
entries in the CPR tables are maintained by the Operating System. A
new entry in the CPRs is made when a Virtual address supplied to the
SAC does not correspond with any of the current table entries. The
format of the Virtual and Real address entries is shown below.

a) VIRTUAL
ADDRESS
FORMAT

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>PROCESS</td>
<td>SEGMENT</td>
<td>BLOCK / LINE</td>
</tr>
</tbody>
</table>

b) REAL
ADDRESS
FORMAT

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>UNIT No.</td>
<td>ADDRESS WITHIN UNIT</td>
<td>BLOCK SIZE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACCESS PERMISSION</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
The Block/Line bits of the Virtual address field illustrated are specially constructed using two storage bits per Block/Line address bit. This enables the Line bits within the Block/Line field to be masked, so that they do not take part in the association process. The variable block size is thus specified within the line itself, so that a single block size need not be applied to all entries in the CPRs.

The Unit number of the Real address field specifies the particular physical store to be accessed. Any Unit, other than the Local store dedicated to the MU5 processor, must be accessed via the Exchange system. The Block size bits specify the Block boundary within the Real address field. The complete Real address is obtained by concatenating the block bits from the Real address field with the Line bits of the Virtual address supplied to SAC.

The Virtual address field and the Real address field are divided by a line pointer register. This permits the interrogation of the virtual address field, and the reading of the Real address field to be concurrent, so that the CPRs may form part of the address pipeline.

3.6 The MU5 local store

The dedicated store of the MU5 processor is known as the Local Store. It comprises four Plessey Series-250 memories, each holding 4096 x 72-bit words (64-bits + 8 parity-bits). These employ a plated wire technology, with a 130 nS access time, and a 250 nS cycle time.

The store is such that the 4K x 72-bits of store are actually arranged as 2K x 144-bits with output selection. This arrangement provides the double-word-read facility, whereby a pair of words from a Local Store stack may be accessed in rapid succession by changing over its output selection.

The four Local Store stacks are normally arranged in a fully interleaved condition, Words 0 and 1 exist in Stack 0, words 2 and 3 in Stack 1, and so on. This provides the most rapid
access of sequential data, required for instruction data and array accesses. The arrangement may be reconfigured, using a number of 'fail-soft' modes if one or more stacks becomes unserviceable.

3.7 The B-Arithmetic unit

The B-Arithmetic Unit (B-Unit) is primarily intended for the manipulation of indices and address modifiers, and it has a direct link with the DR unit for the supply of address modifiers. The manipulation of indices and modifiers usually involves only primary operands, which may be accessed by the Primary Operand Unit (PROP). In view of this, the B-Unit is situated on a spur of the main processor pipeline (Fig. 4.), where it may receive instructions specifying primary operands direct from the PROP unit. Thus, B-Unit instructions may be executed before the execution of a previous Accumulator instruction has been completed. This helps to keep the Main processor pipeline full, since it is possible to compute the required address modifiers early. Great care is necessary when instructions are executed out of program sequence, however, to ensure that there is no operand interaction between the out-of-sequence instructions. This system provides a degree of true parallel processing, since the index/modifier manipulation may occur simultaneously with computational action by an Accumulator.

The B-Unit contains a single 32-bit addressable register and a simple arithmetic unit. The instructions available for manipulation of this B-Register cover arithmetic and logic with a full set of test codes. Operation such as Loading or Addition take place in 40 to 50 ns, whereas multiplication can take up to 800 ns. The Unit may hold two instructions, one currently being processed and the other awaiting execution.

3.8 The accumulator unit

The computational power of the M65 processor is provided by the main Accumulator Unit (ACC), which performs fast arithmetic and logical operations in either fixed or floating-point formats. The
The ACC is situated at the end of the main processor pipeline, and it is the main purpose of the pipeline to provide an adequate supply of functions and operands to keep the ACC busy. The fast access of operands for ACC functions is achieved by use of a small fast slave store in the Operand Buffer System (OBS). Acc-write-to-store instructions, respond into this slave store, and the transfer of data between the slave store and the Main Store is performed automatically by the OBS system.

The ACC contains a 32-bit register X, a 64-bit register A, and a 64-bit ACC extension register AEX. Register X is used for signed fixed-point arithmetic, whereas A is used for 64-bit floating-point arithmetic, or 32-bit unsigned arithmetic. Register AEX is used to hold the least significant part of a double-length result, when applicable. The facility exists in the instruction code to include a Decimal accumulator. This has not, however, been implemented in the present machine.

The expected instruction times are 80-120 nS for the simpler instructions, including floating-point addition. Floating-point multiplication should take about 300 nS and division may take 2 - 3 μS.

ACKNOWLEDGEMENT

A great deal of the material of this article has been taken from the Ph.D. thesis of Dr. J.V. Woods. The author is grateful for his permission to use this material.
REFERENCES


FIG. 2 ACTION WHEN PROCEDURE 'M' CALLS PROCEDURE 'N'
FIG. 4 THE MUS PROCESSOR PIPELINE
FIG. 5 DISTRIBUTION OF NUMBER OF ORDERS OBEYED BETWEEN TRANSFER ORDERS
cycle $i = 1, l, n$

$a(i) = \sqrt{b(i)} + \cos(c(i))$

repeat

FIG. 6 ANALYSIS OF A SIMPLE PIECE OF CODE
FIG. 7 INSTRUCTION BUFFER UNIT (IBU)
a) TYPE 0 GENERAL VECTOR

<table>
<thead>
<tr>
<th>T</th>
<th>SIZE</th>
<th>USB</th>
<th>LOWER BOUND</th>
<th>ORIGIN IN BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

BOUND CHECK INHIBIT
SCALE/DO NOT SCALE ACCORDING TO SIZE
SPARE
SIZE 1, 2, 4, 8, 16, 32, OR 64 BITS
00 - TYPE 0

b) TYPE 1 GENERAL STRING

<table>
<thead>
<tr>
<th>T</th>
<th>SIZE</th>
<th>LOWER BOUND</th>
<th>ORIGIN IN BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>32</td>
</tr>
</tbody>
</table>

SPARE
SIZE 8 BITS ONLY
01 - TYPE 1

c) TYPE 3 MISCELLANEOUS SUB-TYPES

<table>
<thead>
<tr>
<th>T</th>
<th>SUB TYPE</th>
<th>UPPER BOUND</th>
<th>ORIGIN IN BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

USE DEPENDS ON SUB-TYPES
- 0 READ ADDRESS
- 1 READ/STORE DIRECT
- 2 READ AND MARK
- 3 INDIRECT
- 4 - 31 PROCEDURE CALLS
- 32 - 63 INTERRUPT

FIG. 9 MU5 DESCRIPTOR FORMATS
FIG 10 SECONDARY OPERAND BUFFER SYSTEM
INTRODUCTION TO COMPUTER SYSTEM ARCHITECTURE : SOFTWARE

Serge H. GUIBOUD-RIBAUD

CII Scientific Centre, Grenoble, France

1. INTRODUCTION

1.1 Historical evolution of computers

Figure 1 shows the principal steps in the computer evolution, it is over simplified but some remarks may be made about it:
- We distinguish three periods, the frontiers between them being approximate.
- The second generation, because of the technological developments, promoted the diffusion of computers. For the third generation, for the same reasons, we may say that many people are concerned by computers in our countries.
- The third generation promoted the development of huge operating systems and has seen the failure of many big systems (such as B8500, TSS 67), and on the contrary many small computers are now on the market. Figure 2 shows the evolution in price of some small computers.
- The development of technology does not, at present time, have an equivalent in the development of software; figure 3 shows the cost (in lines of code) of some computer operating systems.
- There is also a difference, in quality, between the evolution of languages and the evolution of systems.

1.2 Evaluation of costs

i. Cost of computers.

Let us define what are the main factors entering into the definition of the cost of a computer:

\[
use \ cost = \ \text{hardware cost} + \frac{1}{n}(\text{hardware + software}) \text{ development cost} + \text{maintenance cost}
\]

where

\[n = \text{number of developed products}\]
Looking at the previous figures (1 to 3) we see that factor (1) is going down, so we must put our efforts on factors (2) and (3).

ii. Development costs.

. General conception of the system: this include possible evolution (expandability), reliability, and definition of the domain of problems. There is a need at this level to have global description tools, to be sure that the overall definition is coherent and meets the primary objectives.

. Realization of the system:
  - hardware conception (circuitry): need for tools
  - programming\(^\text{\textsuperscript{4}}\)
    - usual programming time:
      - microprogram: around 3 micro-instructions a day
      - program: around 1 line of code per hour, almost independant of the language. This tends to favorize high level languages.
  - number of programmers:
    Figure 3 explains the number of lines of code, this shows easily the number of programmers required to be tremendous.
    . the degree of complexity (number of programmers, size of programs), leads to the problem of communication between people, and also provides an appreciation of the difficulties of debugging such systems.

iii. Maintenance costs.

  - possibility of integrating changes
  - reconfiguration, which is a preventive maintenance
  - reliability: the possibility of running a degraded system.

iv. Execution costs.

This cost was, for most computers, the primary goal; technology advancement removes it from the first place, because sometimes high speed of execution is not the problem to be treated; one takes into account good debugging tools, reliability and availability which are very often
primary goals. Speed does not seem to be the problem because we may be able to adapt the computer to the users' needs rather than adapt the user to the computer's constraints; so the goal is to raise the level of current computers, speed will then be solved. For example if we built a processor to accept a high level language then as will be seen in section 2, a better knowledge of program behaviour will achieve faster execution; also we will usually gain in code size (compaction). In both cases we probably will have a better use of resources.

In the following sections of this course we shall try to show why this view is very important.

1.3. The computer as a tool for the user

Let us take the user of a computer and suppose that he is writing programs in FORTRAN. For him the machine is:

- FORTRAN language
- the job command language

Both have been designed starting from existing machine and going upwards.

The system provides a set of facilities (hard, soft) which supervises the computer operations, and very often the user has to take into account some constraints of operating system. These constraints are, for example the knowledge of how much main memory the computer has.

As seen before, technology allows now to release the user from the burden of the operating system.

We will try to see what are the primary objectives to be reached when defining a computer system\(^{13,16}\), but we will see first what kinds of systems exist:

- a general system: to be able to support many different kinds of applications
- a dedicated system: to support only one application (airlines reservation, real time such as telephone switching, process control)
- a data base system
- a network of computers

Moreover the objectives may be:
- generality
- reliability
- efficiency
- simplicity
- compatibility.

Let us look in detail at each of these elements:

i. Generality.

The system must, first of all, be comprehensible; and be able to allow future, far as yet unknown, expansions.

ii. Reliability.

The reliability includes hardware problems which can be solved by techniques such as majority processors (more than one is doing the same problem and their results are checked constantly) or the possibility of running with less hardware (as in multiprocessors in which one processor is down, or when input-output devices may or not be on line, or when memory banks are out of orders,...).

But reliability is not only a hardware problem, and we want software to be also reliable; solutions impose that we define very neat structures and interfaces. Also we would like to be able to prove the correctness of the programs, otherwise the system must provide mechanisms for debugging.

iii. Efficiency.

This element depends of the details of the realization, but it is in great dependency with the design because at least poor design can only involve poor efficiency. As seen before efficiency is an element among others, it is not always the most important.

There is another kind of efficiency: the programmer's or designer's efficiency, which has nowadays to be taken into account more than it was previously.
iv. Complexity.

It is the principal enemy of reliability and, very often, of efficiency and generality. We shall try in this course to show that correct structuring, and specializing, will reduce this element. Tools are very much needed.

v. Compatibility.

This element will weight more and more as we go on, because of existing investments (which are sometimes huge as in the case of OS360) and of the need to avoid rewriting everything.

This element is, most of the time, a check to get better systems; and very often it is coupled with the wish to be more efficient than before and also to have new facilities. A puzzle the designer will face.

Whatever kind of system, and which ever of the previous elements are choosen, we can still determine general properties of the systems. The rest of the course will consist in looking to what they are.

1.4. Methodology

1.4.1. Problems to take into account

The definition of a machine is not a simple thing and one must start by defining which method to use, this method must apply from the global design to the specifications of parts of the machine.

i. Objectives.

One must start with the definition of the goals to be met and of the criteria for judging that the goals are reached. The following is an enumeration which must be:

- coherent, that means not to be contradictory
- complete, that means not to ignore some phenomena which will appears in future stages of the design or realization and which will oblige one to rethink the problem from the beginning.

Among objectives we see:

- easiness to use the machine (system and language), which
means that the outside view must be as close as possible to the problems to be treated by the user

- possibility of expansion of the machine, because one sees that it is difficult to define the right product at the first try.

ii. Design.

Having recognized the goals, and even defined a solution two problems arise:

- how to describe specifications, both internal and external. It is easy to see that the designer, while not, very often, be the implementor so the former has to give instructions to the latter and these instructions must be the most precise possible allowing no erroneous interpretation from the implementor, because this interpretation will, most of the time, be made from the implementor viewpoint and be contradictory with the initial goals.

This point shows the need for some language to write specification.

- how to be sure of the correctness of the design. Most of the time machines were designed and people building them discovered problems in the outside definition; for example problems of synchronization, or even logical "errors" meaning that, under certain circumstances, the machine does not behave as it was supposed to.

iii. Implementation.

The more we go into the realization the more people will be involved in the implementation and the more difficult the communication of instructions will be. At this level also another problem arises which is how to control the evolution of a project. Some tools are also needed at this level.

1.4.2. Methods

In theory there exists two different methods, one starting from the user domain of problems and trying to define the next most appropriate level, and the other one starting with a given level (for example existing machine) and trying to map users' problem onto this machine. It is
more natural to design hardware to match the software rather than the other way around.

In practice both methods are used, parts of the design are top down and part are bottom up. Let us explain in detail a top down methodology we used for the definition and realization of the system GEMAU.

The method is divided into levels where each stage is concerned only with one level to define the next one. For example to realize the hardware for a machine one needs only to know the machine instruction set, its addressing and control. starting from these logic design may commence.

Each level is defined using the following four steps:

- knowledge and definition of objectives
- observation and analysis of the phenomena
- building of a model or a prototype
- measurement and evaluation of this model or prototype.

Some back up may be necessary between steps.

Let us make some remarks on this method. It is usable only if time constraints are not to rigid because each level must be defined when the previous one is completely defined; there will exist problems in a short term design; but we think that this will provide an answer to the high costs of design, development and maintenance.

1.5. Distinction between language and system

In this section we try to show what are the main differences in the domain of languages and systems. But in fact the frontier is very hazy and the distinctions are made to facilitate the understanding of the problem.

i. Languages:

they are well known and a good amount of formalization has been done on the subject. We will say that languages deal with the semantics of objects: data types, operator on data, control structure (procedures, go toes, iteration, recursion,...). Another remark about languages is that user programs will very often have predictable "behaviour".
ii. Systems:

In contrary to languages they are not very well known and are more of the domain of research. They deal with the problem of allowing communication between objects, whose semantics are not known from the system point of view. The system provides a structure and must enforce relations. Its behaviour is unpredictable due to the very different, and even contradictory, nature of the treated problems. For example on the same system we find program written in different languages such as FORTRAN, COBOL, ALGOL, and so on... Some programs will do a lot of input/output operations (file manipulation), some others will do computation involving few communications with the outside world; their behaviors are not similar. The system will try to share the computer resources (processor, input-output devices, main memory) as most efficiently as possible; we will see later what criteria can be taken.

iii. Comparison between languages and systems

there are no reasons why the system must not reach the same quality a compiler has, this involves that the system designer understands the common principles of operating systems; section 3 will try to look at these principles. Moreover operating systems are no different from a user's program, they are just bigger.

To conclude we can make a parallel with a telephone system. The system enables people to join one another (establishment of communication, and transfer of information) but the information transferred on the lines are not interpretable by the system, it is the responsibility of the communicating people to understand each other: if one is talking English and the other one Japanese they will have to agree on a common language for communication: English, Japanese or a third language.

But this distinction between language and system is not absolute, if we take languages like ALGOL 68, SIMULA, PL/I they include some system aspects.

2. RELATIONS BETWEEN MACHINES AND LANGUAGES
Although every machine is defined to treat information using algorithms expressed in some language, there has always been an interest in incorporating some interpretative elements oriented towards a particular language (at the hardware level).

1960 : Burroughs B5000
1963 : Randell and Russell, "ALGOL 60 implementation"
1965 : CAB 1500
    Bashkov, "System design of a FORTRAN machine"
    Weber "A microprogrammed implementation of Euler on IBM 360/30" \(^{14}\)
1968 : Burroughs B6500
1970 : Abrahams, "an APL machine" \(^{1}\)
1972 : Burroughs B1700, a machine with dynamic microprogramming
    Wortman "A study of language directed computer design"
    Grebert, "Space Machine Language Study" \(^{25}\)
1973 : Hassit, "APL machine language study"
    Chevance, "Machine COBOL"
    Alvan

One instruction of high level language generates several machine level instructions, so high level language is more expensive than assembly language, but it has the advantage of being usable (very often) on other machine and for COBOL and FORTRAN on a wide range of different machines. Assembly language is usually restricted to a given machine range. An idea is to orient the hardware towards high level languages. There will be more efficiency in the treatment of recursion, iteration (APL like operators), in the code compaction of a program and in the time of execution.

The basic idea is still here to design the hardware to fit the software.

2.1. Notion of processor and program

i. Case of a 360.

The 360 is a computer with byte addressing, 16 general registers for addressing (base), arithmetic, indexing. There exist a set
of instructions each of which has an order-code and a length different from the length of other instructions. Programs are stored in memory, and instructions are fetched into registers of the central processing unit (CPU), where recognition of the instruction order and of the operands are made before execution proceeds. On some 360s the CPU is implemented purely by hardware, on others it is implemented by microprograms.

ii. Case of an interpreter (software)

Let us take the language ALVAN, which is a functional language. Functions calls like \( f(x, y) \) are written \([f, x, y]\) where \( x \) and \( y \) are parameter strings of characters and may also be function calls. For example \((1 + 2) \times 3\) may be written

\[
[ x, [+ , 1 , 2 , , 3 ] ]
\]

to execute sentences in this language one may write an interpreter with some internal variables to represent at any instant the state of the interpreter. In this case the interpreter will analyze the first characters and put it on a stack, then continuing... until it finds an end of function call (]), at which times it executes the \( + \) operation. To be able to realize stacks we need pointers and other variables, these variables are internal but the ALVAN program is considered as data by the ALVAN interpreter.

iii. Levels of abstraction

The two previous examples show the relativity of the notion of processor and program. The ALVAN interpreter may be written in a machine language, which is in turn interpreted by some other processor. This processor may be itself microprogrammed on some other machine, and so on... Figure 5 shows the relativity of the notions of processors and programs.

The user will only know his program and the ALVAN language, the designer of the interpreter, the ALVAN language (in fact an automaton to recognize sentences in this language) and the language of the micro-coded machine, and the micro-code designer the micro-code language and the hardware to realize it. Each of these people do not need to know more than the level to be realized and the level on which to realize, everything else is ir-
relevant to him.

2.2. User's standpoint

We shall try to see what is a language from the user's point of view; it is essentially a way to describe and solve his problems. He is concerned with several characteristics of the language, which we will present in a descriptive manner.

A language is essentially characterized by

- the types of treated information (data)
- the operations which may be performed on these data
- the syntax, which concerns how the user will write his statements.

i. Data types:

They concern the kind of information the user can treat, and they depend essentially on the domain of application of the language. In a scientific environment we have numbers (integer, real, complex ...), in a business environment string of characters, records (which are some kind of compound data types).... Types may be primitive or built, and also elementary or compound.

An elementary type is used for atomic objects:

- integer
- real
- boolean ...

A compound type is a composition of existing types, it refers to non-atomic data:

- array
- stack
- record ...

for example array of integer

- stack of array of integer
- record: integer, character string ...

Primitive or built types refers to the possibility of defining new types (ALGOL 68 modes), not only compound. In fact, a type defines the kind of operations which can be made on an object of
this type, built type will be the ability to define new type, i.e. new operators.

It is important for the reader to note that a procedure is also a type.

ii. Operators

The only way to manipulate data and to control execution of a given program is through the use of operators; they depend on the type of the data. For example addition on integer does not mean the same thing as addition on real.

iii. Syntax

The syntax is very important from the user's point of view because it represents the "sugar". But syntax, from the designer point of view, is far from important and there exists techniques to change a program written in one syntax to an equivalent program written in another syntax. Translators are example of this.

2.3. Abstract machines

An abstract machine is a set of concepts and specifications written to achieve some goals (in user's problem terms) in such a way that a user is able to interpret the operation of this machine in a non-ambiguous manner. The objects manipulated may or not be realized, for example the IBM 360 as described in the "principle of operations manual" as an abstract machine. We shall try, now, to look in more detail, into what kind of concepts usual abstract machines deal with.

2.3.1. Elementary operations

An object has a value; for example in ALGOL 60, 2 and 25 are two objects whose values are the integer numbers 2 and 25. But very often we want to have variables and give a name to these variables. A name is a symbol, and we need a way to create variables (names) and affect a value to a name. The first problem to solve when defining a language is how names are manipulated and what are the values which can be assigned to names; the latter is equivalent to the definition of the object types
2.3.2. Syntax

For the designer the first task is to determine the domain of problems which the machine must be capable of solving; then he has to describe this machine. He needs techniques of describing the semantics.

At present no "universal standard" exists. Historically, there exist a lot of "ad hoc" methods, each of them being more or less oriented toward ease of communication, or portability. And the description is more or less complete. We will just give some headling to the reader who will know more about abstract languages.

The following works have been done:

- PL/1 : VDL (Vienna definition language)
- Euler : syntax based method
- Pascal : axiomatic method
- SPLM : specialized machine described in APL

2.3.3. Interpretation versus compilation

i. Definitions:

A compiler transforms some program written in a given language into another program written in another language (cf. 2.1 example about processors).

An interpreter directly executes the program.

The question arises for a given language of knowing what is necessary: an interpreter or a compiler? A first answer is to say that it depends on the languages, some are partially compilable, some others like ALVAN are not.

Let us take ALCOL 60, where an array has a fixed but unknown (at compile time) size; the user is able to use subscripted variables and to do arithmetic on them. To check that when accessing an element of an array, the subscripts are not out of bounds can only be done at runtime. At this point the solution may be either to generate the instructions to do the check, or to have hardware array instructions or to do nothing as in many compilers. In problem of that kind it is important to know what services the user wants to be sure that it is worth while
doing the check. In the present case it seems very useful for debugging purposes.

A second answer to the interpreter/compiler problem resides in the difference of services. In 3.4.1, we will see that usually compiling is not reversible, names of identifiers used in the source programs are transformed into addresses at the moment of execution; one sees that if some problem arises during execution, (such as divide by zero operation, over or underflow in an arithmetic operation or subscripts out of limits), there is no way to tell the user what happens in terms of his source program. In fact, methods exist where the compiler keeps a correspondance table between run time addresses and variable names, and also with statement numbers in term of the source program. Let us take the following instruction:

\[ A := B / C / D; \]

it is very awkward to tell, if a divide by zero occurs which variable C or D equals zero. In this case it will be possible for the user if a post mortem dump (in terms of the source program) is provided with the current values of the different variables. But if the expression is more complex it becomes hardly practical because the user will have to play the computer game. For example

\[ A := X / ((A*N - B) \times (\exp(X) - C)); \]

where A, B, C, N, X are real. Who will compute to know if

\[ A*N - B = 0 \quad \text{or} \quad \exp(X) - C = 0? \]

If in this case we had used an interpreter then the answer would have been easier, see 2.1.

A third solution may use the fact that compaction of the user code is usually better with interpreters than with compilers (figure 8).

Interpretation does not mean that the processor will run source programs without any changes, but some translation may occur. This translation is in fact generating a one to one correspondance between the source program and the object program.
ii. Classification:

we may class machines in two different classes:

- multi-language processor
- specialized processor.

The figure 6 summarizes the differences between the two classes

\[ P_{l_i} \] : means a program written in language \( l_i \)

\[ T_{l_i} \]: means a translator for a program written in language \( l_i \) into a program written in language \( L_k \)

\[ I_{L_k} \] : means an interpreter for language \( L_k \)

The figure 7 gives comparative figures between compilers and interpreters, in terms of size.

Figure 8 gives the compaction expected by using specialized processors versus general processors.

iii. Remarks

Before deciding which way to go the designer must answer to the following questions:

a) is the time required from compilation to loading in the same order of magnitude as the execution time? This depends heavily on the application; in a software house, a development centre or a research centre this time is probably higher than the execution time.

b) what is the common denominator of the previous steps (edition, compilation, linkage, loading): usually, they all need character string operators to analyze text. With the evolution of technology, it becomes possible to have hardware (or micro-programs) especially adapted to each of these steps. We will return to this as system aspects are involved.

It is possible to reduce in great proportions the previous
steps because they are here only to simplify the real machine.

In section 3 we will see in detail:

a) dynamic linking (MULTICS, GEBAU)
b) loader: dynamic address translator (paging and other mechanisms)
c) translator: one specialized machine per step.

For batch type systems we will recommend interpretation during debugging, and optimized compilation for exploitation.

2.4. Techniques

2.4.1. Prefix notation, descriptors

Let us take the following program, written in a language like FORTRAN or ALGOL.

integer A; real B; real C;

C := A + B;

and let us also make the assumption that we are using a machine with general registers to do arithmetic; this machine also having direct addressing, i.e. an instruction could have the following format:

```
+---+-----------+
|   |          |
|   |    operand address    |
|   |          |
|   +------+
|          |
|    register number |
|          |
|          |
|    order-code |
```

i. Hypothesis 1:

order code in the instruction indicates the type of data to manipulate. The compiler generates:

Load, Ri   A ..... load A into Ri
Float, Ri   ........ transform integer in Ri into floating point, result in Ri
Fadd, Ri   B ..... add (floating point) B to Ri and
result in Ri

\[\text{Store, Ri} \quad C\]

A, B and C here are the address of memory cells.

ii. Hypothesis 2:

we have another machine, but with each word of memory we attach a prefix (tag field)

\[
\text{data} \quad \text{value} \quad \text{tag field}
\]

\[
\begin{cases}
001 & \text{integer} \\
010 & \text{real} \\
\vdots & \vdots \\
111 & \text{indirection}
\end{cases}
\]

Then we have for A a tag field equal to 001 and B, C a tag field 010. The generated code will be

\[
\text{Load, Ri} \quad A \\
\text{Add, Ri} \quad B \\
\text{Store, Ri} \quad C
\]

We remark that operation code is the same which ever type is the manipulated data. A big advantage of this method is with indirect addressing. Suppose that A instead of being an integer was a reference to integer D, i.e. the type of data is address and value is an address. Then we will have

\[
A \quad 111 \quad \rightarrow \quad 001
\]

but the code will still be the same. Load, Ri A is equivalent to "Load, Ri D". Problems with this kind of indirection are the possibility for infinite looping; solutions may utilize a watchdog (every instruction must be done in less than n milliseconds, or a trap will occur like in the B3500) or with a counter on a maximum number of indirections.

2.4.2. Stack machine
Let us return to the example of 2.4.1, but remove the hypothesis that the machine has general registers and replace it by a new one : no tag field on data.

i. Hypothesis 1 :

the machine has \( n \) registers: then we have exactly the same solution as in 2.4.1. i.

ii. Hypothesis 2 :

the machine has a stack, and instructions apply to the top of the stack. The program will be

Name C ..... put in the top of the stack the address of C
Load A ..... push the stack and load the top with the value of A
Float ........ operate on the top of the stack and transform it into floating point
Load B ..... 
Fadd ........ add the two elements in the top of the stack, push it and replace them by the value of the addition
Store ........ Store the top of the stack into the address given in the subtop, these two elements are removed from the stack.

2.4.3. Comparison of the previous techniques

i. Comparative figures. See figure 9.

In the case of stack machine, there need no address field in instructions working only with the top of the stack (such as Add, Mul, Store).

ii. very often the previous techniques are mixed the example in i. shows in the third case that this gives interesting results.

3. RELATIONS BETWEEN MACHINES AND SYSTEMS
Very soon the designer tried to orient machines towards the inclusion of some operating system features. A rapid look at the evolution of operating systems will be useful:

first generation: - single-job operation
   - operating system functions included in the job
   - one machine for each user
   - relied on operator's work

second generation: - execution of jobs with automatic transition
   (serial batch)
   - independency between the operations in the different pieces composing the hardware. (CPU, main memory, i/o devices)
   - introduction of interrupts
   - master/slave mode

third generation: - all known basic services: control of execution, scheduling of jobs, i/o handlers, ...
   - introduction of virtual memories and techniques to realise them (paging, base registers for relocation ...)
   - multiprocessing
   - time sharing
   - virtual machines\textsuperscript{22})
   - real time
   - networks
   - needs for high level of reliability

The third generation shows that software cannot be successfully signed if the hardware does not follow, and some features were even put in hardware for some special systems (see the GE-645 for MULTICS\textsuperscript{39}). Another remark which may be made is that this evolution of systems has been possible because of the tremendous progress in technology (speed, integration, reliability), and also because of the development of measurement tools.

Systems are more than the languages linked to the hardware (see for example: protection 3.4.2 and synchronization 3.4.3).
ATLAS was the first machine to have a virtual memory.

We will pay special attention to two different systems MULTICS and CP67.

MULTICS started in 1965 and was built to be the operating system for large scale systems, it provides the computer field with a tremendous amount of influences. It was aimed at giving a comprehensive philosophy of programming design and structures, and also has influences on its own hardware, where there must be built in features.

CP67 is the system (on IBM 360) which introduced the concept of virtual machine; the idea is very simple: why not give to all users a machine which is a 360 with some core memory of its own and some peripheral devices. All these machines will run on the same hardware, and their configurations will be "virtual". Subsystems of different kinds may be run on different virtual machines.

3.1. Definition of the aspects of a system

3.1.1. One user's system

By this we mean something equivalent to a monoprogrammed batch system, i.e. only one user at a time using whatever resource of the machine he wants.

i. The standpoint of the user:

He only wants to solve his problem in the simplest way, and does not want to worry about what he must do if the system is working in some crazy way. The first thing an user wants is a language to write his problem and solve it. The second is storage to keep information during execution of his program or when his program is finished and before another program will reuse the storage. The storage will be more adequate to the user if it is symbolically named (usual file system). The third thing an user may want is to have a coherent view of the system, i.e. he has not to know how a file is realize (punched card, magnetic tape or disk,...), and whether the file is part of memory or not. The user does not want to use things like overlay structures because
central memory is too small, the fact that a program has to be loaded into central memory to be executed is irrelevant for the user. A fourth need is that of being able to replace part of its program or data by other parts. A fifth need will be the possibility of managing traps and external events, this is particularly useful for debugging.

ii. The standpoint of the system's designer.

The system will take care of a lot of problems to make the user able of running his problems on a given machine. It will provides a virtual memory, i.e. mapping the user's view onto the real world, for example a program does not need to be entirely resident in central memory to be executed. The techniques to do this are referred to by the name of paging, segmentation, memory hierarchy, ...

On the same kind of ideas the use of multiple processors (central processor, peripheral processor) is indifferent to the user from a functional point of view. If the execution of a program waits until an i/o operation is completed or not has only an influence on the performance of the execution. This is not true if asynchronisms appear explicitly in the user's language, for example in SIMULA. Efficiency must be taken care of by the system as must reliability. This does not mean that a user has no idea how his program is executed but this is not essential. Also the system people must provide information to tell the user what is the cost of certain operations.

It is important that an user can do what he wants even if he does not know how the system is working, some knowledge will sometimes improve efficiency but this is not always necessary. Let us take an example: if the system has one or more CPUs is irrelevant to the user, he need have no knowledge of this fact.

3.1.2. Multiple user's systems

i. The standpoint of the user

The needs are the same as for single user, but three more requirements are necessary: the first is to be able to
communicate with other users on the same system, the second is to protect the shared information (control which user may access it and what he can do with it: read, modify, delete ...), and the third is to be able to synchronize with others.

ii. The standpoint of the system's designer.

The designer has several contradictory constraints: the first to satisfy the user on the functional side, the second to provide him with a tolerable response time, the third to use the machine in the most efficient way (from an overall exploitation point of view), and the fourth, to provide a way of ensuring the coherence of data (reliability). The reader sees easily that these constraints are contradictory; for example, between user's response time and efficient use (throughput, turn around,...) and also between response time and reliability.

To be able to built a good system the designer must make trade-offs. The best way seems to have a good knowledge of systems and to have some kind of functional model (from the user's point of view) which will tell the designer what is needed, and by studying this model and knowing the price and performances desired he may be able to make proposition. We will try, on the remainder of this course, to show that this is not only a possible way to go, but also a very promising way.

For example we may implement a system with specialized processors i.e. ALGOL, FORTRAN and COBOL for a given price/performance ratio, and in another case have only a classical processor with three interpreters written in software for another price/performance ratio. The user will not notice the difference in regard to the fact that his jobs may run on both machines he will only see a difference in efficiency, but the price he pays may be different. The reader sees from this example, that this is the same method as the one used in the relations between machine and language, and especially in the interpreter problem (hard, soft ...).

3.2. Naming and addressing

We try to see what is a system from an user's point of view. By this
way we tend to define more and more precisely what is the domain of a system, in some cases we take some options and illustrate them but their justifications will occur only later.

3.2.1. Objects manipulated by a system

We consider a system as a set of objects which may be primitive (processor, card reader, memory,...) or compound (procedure, files, ...), the last category is built using primitive objects with some construction rules. In fact the system is only concerned by the existence of these objects and the relations between them, but not by their internal semantics. For example, a file will be a compound object built with some memory and procedures, but how memory and procedures are realized (their internal algorithm) have no interest for the system.

The type of an object is either primitive or compound, the type is completely defined knowing the operations possible on the object, for example a file is defined by two operators GET and PUT, a memory is defined by FETCH (address) and STORE (address, value). If the type of the object is primitive that means that its operators are built into the machine (this is the current level of abstraction).

The value of an object is its internal contents and this value can only be accessed by one of the operators of the type. For example (figure 10), the central memory of a computer, the only way to obtain the content of the word of address 100 is by FETCH, even if memory is divided in several physical blocks, for example interleaving. Interleaving is a technique which separates even and odd addresses in two separate physical blocks of memory, so parallel access can be made by two different processors. The technique is even used with more than two blocks. But in every case (with or without interleaving) all the processors are only concerned with the type of memory and they ignore completely where the information is. If I store 20 in address 100, I want to get 20 when I fetch the content of 100 wherever 100 is.

Other examples of objects can be found in:

- file directories, libraries
- procedure
- systems operators (call to the operating system : SVC on
The reader sees that there are not many differences with what exists in the languages studied previously, especially in the examples of prefix and descriptor.

As seen the same object may be accessed simultaneously (example of central memory and multiple processors). The system must be able to choose, in case of conflict, which access can be done. The synchronization being done only at the object level, it is an intrinsic property. For example, if several users are accessing the same object, let us say a compiler, they will use the same procedure code but this code may or may not be written reentrantly. If it is reentrant then multiple users may use it concurrently, otherwise some will have to wait until one has accomplished his compilation. Let us go back to the example of central memory, suppose that are 3 processors P1, P2 and P3; P1 and P2 try to access the word 100 and at the "same" time, also that there is no interleaving in memory (figure 11). Then the memory access controller will serve one, let say P1, then another one (for example P3) and the last one (here P2). This example supposes that there is a way of defining the access priority. If we take the same example and suppose that there exists interleaving (figure 12), then, if the same priorities are taken as before, P1 and P3 will be served at the same time and P2 will wait until P1 has finished its transfer.

We see, on this example, that the synchronization is done at the level of the object and the choice of decision when conflict arises are also taken at this level.

To conclude, figure 13 recapitulates the different components of an object.

We shall call the set of objects known at a given level of abstraction a VIRTUAL MEMORY.

In the previous examples it is easy to see that objects have some intrinsic properties which are of interest for the user, and the technological properties, i.e. how to realize them, which are related to the
designer's problems and especially with the trade-offs to be made.

3.2.2. Computational objects (process)

As we have seen before (3.2.1) there may exist several independant and simultaneous use of an object through its operators. These different uses are done by processes, i.e. some independant executions. Before giving a more precise definition of what a process is let us take an example written in ALGOL 60 (see figure 14), and let us take a procedure, for example P1. This procedure uses several variables:

- local to the procedure X (3)
- global Y (2)
- formal parameter A

When activated the formal parameter will be replaced by an actual parameter and the execution proceeds. We call the computation record the set of information necessary for allowing execution to be performed, this includes, for example, the address of the current instruction. In ALGOL 60 procedure calls may be recursive, that means that P1 may call P1 inside itself, then a new computation record will be created with a new local to the procedure (X) which is different from the previous activation, and a new storage location will be assigned to X.

If we look at the previous example we discover that there exist several objects named X and the problem is now to determine which X is to be used in the different statements where it occurs. Let us take the call to P1, the object of name X is the integer numbered 4, while during the execution of P1 X is the real numbered 3. For Y at calling time it is integer numbered 5 and in the execution context it is integer numbered 2. This example shows the existence of contexts where names do not refer to the same object. The calling context (here objects 4 and 5) is used to solve actual parameters names, and the procedure execution context (2 and 3) is used to solve local and global variable names.

This example shows also that there exists clashes on names, i.e. the same name refers to different objects and to be able to specify which object we were talking about we used the terminology "object numbered i". For the system point of view it is the same problem. We call this number the universal name of the object, and to have no clash on this name it
must be unique. We can summarize in the following schemata the different kinds of names

![Diagram of set of all names mapping to universal names and objects]

and to be more explicit

![Diagram of global context, execution context of PL, and calling context of PL mapping to universal names and objects]

The first goal of the system is to realize the mapping from the usable names to the universal names.
The second goal is to associate objects to a process (these objects are very often called resources).

3.2.3 Names

As seen before there exist at least two different types of name; we will see that there may exist more, but before going deeper into the different types let us define what a name is.

A name is a symbol, this symbol is interpreted to give another name, and this until we reach a name which is primitive.

Let us take as an example a FORTRAN program, where I is an identifier of type integer; at compile time, this name (figure 15) will be transformed into a displacement relative to the beginning of the program, and at execution time, this displacement will be interpreted by the hardware (processor) to give an address in memory, and as seen before, the memory controller will find the location of the address.

The previous example is very simplified and we will talk more about name transformation in the section about binding (see 3.4.1).

Another example can be found with the file management in operating systems. In FORTRAN one may say:

\begin{verbatim}
WRITE (unit, format) variable
\end{verbatim}

where unit is an integer, which is the internal name of a file and before execution of the program a real file will be assigned to this internal name, this real file being either a predefined file or a file assigned by the user.

All these internal names are in fact called local names in the computational object which represents the execution of the program. This shows the existence of several set of names:

- object names: in the previous example these are file names

We call these sets the addressing space.

- local object names: in the previous example these are unit numbers.

We call them the execution space.
Let us now look in more detail at different kinds of naming spaces.

i. Example 1: independant user - System SIRIS 7.

The filing system (figure 16) provides each user with a set of owned files, these files are referenced through one directory per user, the users themselves are described by a system directory SIRIS7 allows one also to have super-files, so objects described in the user's file are either files or directories.

The execution context is given by the user's name provided in the job control card, and every name referenced by the user's program is automatically prefixed by its username, this is the system name which is unique for every file. To share one must be able to give the system name.

ii. Example 2: cooperating users - MULTICS.

At this point it is worth while noting that there exist different types of sharing:

- sharing may be explicit or implicit: explicit if the user himself specifies what he wants to use, and implicit for systems procedures such as a compiler

- sharing may be sequential or simultaneous: sequential for example in the batch operating system (one user uses a file, and another one uses the same file after the first) one has terminated, and simultaneous, for example a compiler.

MULTICS provides both type of sharing and defines a tree structure for files; the naming procedures are different from the one in SIRIS7 in the sense that a user may name any file in the tree. In future sections we will see what are the problems linked to sharing and also how they can be solved.

3.2.4. Job control language

Every operating system provides a job control language to allow users to control his executions, to specify how to bind objects into his computational space and also to control the evolution of this space.

Very often existing control languages are awkward, and they desi-
designer of a machine should carefully define them because it is one of the languages of the user's machine.

3.2.5. **Remarks on the system aspects**

At this point of the discussion it is easy to see that these system aspects are all related to one another and it is impossible to define protection without defining how a user addresses objects. We may have the following statement:

> Usual system problems may be solved easily, elegantly and efficiently by the system designer's by the definition of an adequate naming mechanism.

The next section describes in detail a prototype system built on a CII 10070, which tries to justify the preceding statement. In the section after the description we shall try to explain how this naming schema reaches or does not reach the primary goals and compare it with some other existing systems.

3.3. **The system GEMAU**

3.3.1. **Main objectives**

The primary goal is to provide an abstract machine (we call it the nucleus) on which one may easily write subsystems. This nucleus should allow modularity, sharing and synchronization to be done in the user's point of view. This is not an operating system but rather a provision of basic tools to enable someone to design the operating system of his choice.

i. Modularity, hierarchy.

The structure provided by the nucleus should allow to write subsystem, and these sub-systems will in turn support other subsystems, and so on ... There exist between subsystems (and objects composing them) a hierarchical relation, so it is not necessary to give to a procedure access to the entire addressing space, but only to provide access to the objects needed for computation. Figure 17 gives an example of such a
structure.

ii. Naming objectives.

Names are used as indicated in figure 18, this figure is for one computation \( S_{LN} \) and one subsystem \( S_{PN} \) only.

If there are more than one computation, each of them will be represented by a \( S_{LN} \) idem for several subsystems \( S_{PN} \)

iii. Sharing, protecting and synchronizing.

The objectives stated in i. impose that object be shared between subsystems. The sharing imposes some mechanisms for protecting and synchronizing accesses to these objects.

3.3.2. Definitions

i. Addressing space.

We use a tree structure, as in \(^{10}\) where nodes are directories which describe objects. We call segment an object whose type is either data or procedure or both, peripheral device an object which represents a peripheral i/o device, and link an object which refers to another object.

A directory allows one to symbolically name objects which may be of type "segment", "directory", "peripheral device" or "link". An object has a unique entry name in a directory, and the name of an object (in regard to a given directory) is the list of every entry name of objects in the path ; intermediate objects must be directories or links, its name is called the pathname.

A predefined object is called the root.

Each object is defined by a descriptor in a directory (with the exception of the root), and this descriptor holds the following characteristics:

- entry name
- protection
- type of the object
- universal name of the object
The control of the addressing space is realized by a special mechanism using the environment concept; an environment is a distinguished directory which allows one to address a subtree (and only this subtree) of which this directory is the root. An environment defines a visible space, it is the set of all objects having direct pathname from this directory (i.e. intermediate entry names are never of type link). The reader can see that visible space of different environments are either nested or disjoint, special properties of this fact will be seen later (3.4.3).

Figure 19 gives an example of an addressing space. The segment 3 has the names U1.PP and PP in regard respectively, to the environment 0 and 1.

Segment 9 has the names:

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>SP2</td>
<td>UT1</td>
<td>from 0</td>
</tr>
<tr>
<td>U1</td>
<td>library</td>
<td>UT1</td>
<td>&quot; 0</td>
</tr>
<tr>
<td>SP2</td>
<td>UT1</td>
<td></td>
<td>&quot; 1</td>
</tr>
<tr>
<td>library</td>
<td>UT1</td>
<td></td>
<td>&quot; 2</td>
</tr>
</tbody>
</table>

ii. Execution space, process space.

A procedure is well defined by its local names and its environment (i.e. the closest directory which is an environment on this procedure direct pathname from the root). As seen in the previous chapter, local names may refer to objects non visible in the procedure's environment if they refer to actual parameters.

A procedure is also defined by the set of the primitives GMAU which may be used (capabilities), these capabilities being bound to the environment of the procedure.

An execution space, which is the activation record of a procedure, contains the following information:

- set of local names
- environment of the procedure
- capabilities
- current instruction address.

A process is a stack of execution spaces, and the current state of a process is defined by the top of the stack, which means that calls are
procedural (the returns always go to the calling procedure).

Figure 20 gives an example of an execution space. We note that local name LN₂ refers to an object not visible in the current environment which is 6. This figure also shows that processes P₁ and P₂ share the same object though not in the same environment and not with the same local name. We shall see later on how one process changes its environment, we state now that it is only by use of procedure calls and returns.

The creation of a process needs an initial procedure, its eventual parameters and a reference environment (to solve parameters' names; this environment is the father's environment at the time of creation. Execution space of the son is built from the current environment of the father. This means that a process will never have more capabilities than its father has, but this does not mean that it will not have, from time to time, more capabilities than its father had at the same time (i.e. its father may have more if it wished.

The death of a process is effective only at the end of its initial procedure. For control reasons we privileged the father-son relationship: a father is responsible for its sons, all other relation between processes may only be done through the addressing space mechanism.

This hierarchical structure supposes the existence of process ancestor, whose initial environment is the root of the addressing space.

3.3.3. Environment, addressing space
i. Operators on objects

The nucleus provides some primitives (or verbs) for manipulating objects in the addressing space. These primitives are different depending on the type of the object to which they apply.

Modification of the addressing space

```
CREATE (< designation of directory > *,
        < entry name >,
        < designation of initial value > **,
        < description > ***)
```
Enter a new object in a directory and define the characteristics of this object.

```
DELETE ( < designation of directory > , < entry name > )
```

Delete an object from a directory. This object no longer has a pathname, it is only accessible through local names, if some are referring to it. Physical destruction will be seen in detail later.

```
* < designation of object > ::= pathname/local name
** initial value refers to an object, and defines the initial value of the object to be created as a copy of the referred object
*** see iv. for a more precise definition
```

. Modification of the execution space

```
[ < local name > ]* ... <- CALL ( < designation of segment/ peripheral device >
[ , < designation of object > ]...
```

Execute a procedure after having built the corresponding execution space. The local names corresponding to the actual parameters are created, as is the local name of the procedure (\( LN_o \)). If the designations are pathname then the resolution is done relatively to the caller's environment. The procedure's (callee) environment becomes the current environment of the process, the previous execution space being saved into a stack. The procedure may return results (objects), if this is the case these results will be assigned to the list of local name in the left

* [ ] means optional
[ ]... means repetition, including zero times.
part of the call.

```
RETURN ( [< designation of objects >] ... )
```

End the execution of the current procedure, the execution space is pulled, the current environment and the local names are those of the calling procedure with, eventually, creation of new local names as indicated in the call left part, and assignment of the objects given as parameters of the return.

```
< local name > + BIND (< designation of object >)
```

Create a local name referring to the designated object. If the designation is a pathname resolution will be done in regard to the current environment.

```
FREE (< local name >)
```

Delete a local name.

Every object is created by a primitive CREATE, it is not deleted physically by the primitive DELETE, but only when no local name (of any process) refers to it. This property allows to simply realize creation and destruction of temporary objects, which will exist only through local names; they will not have an entry name nor a link towards them. They are created by

```
< local name > + BIND (, < restrictions >, 
< designation of initial value >)
```

and deleted either by FREE or when no local name refers to them (return of procedure).

Binding time

The reader sees that binding is performed by affecting a local name
to an object, so the user will choose explicitly the binding of an object, see 3.4.1 for a detailed example.

ii. Protection.

The object's type defines which operators are applicable to it, so protection is simply done by restricting the use of some of these operators.

```
< restrictions > ::= < addressing space restrictions >
    / < use restrictions >
< addressing space restrictions > ::= CREATE / DELETE
< use restrictions > ::= CREATE / DELETE
    / (READ / WRITE / CALL)
```

iii. Concurrent access: sharing.

The addressing space has been designed so that sharing of objects is natural, nevertheless it is possible that one wants that only one process to use a file at a given time, or a non-reentrant procedure or a peripheral device. As seen before such a property is intrinsic to the object. We call multiprogramming degree of an object the maximal number of processes able to bind the object at the same time.

Synchronizing is done, at the level of the object, by the nucleus; if too many processes ask for it they will be queued on the object and released each time a non-blocked process unbinds the object.

iv. Object descriptions.

```
< object > ::= < directory > / < environment > /
    < segment > / < peripheral device > /
    < link >
```

* Depends of the object type.
< directory > ::= < directory descriptor > [, < reference > ]...
< directory descriptor > ::= < type > = 'directory',
< protection > = CREATE / DELETE,
< maximum number of entries >
< reference > ::= < entry name >,
< addressing space restrictions >,
< multiprogramming degree >,
< universal name >

The descriptor of a directory is an intrinsic characteristic of the
directory, its value (| < reference > | .... ) is the set of descriptors
of the object from the standpoint of the addressing space (extrinsic
characteristics).

< segment > ::= < type > = 'segment',
< protection > = (READ / WRITE / CALL),
< maximum size >,
< vector of words >

< peripheral device > ::= < type > = 'peripheral',
< protection > = CALL

Now we can detail a local name:

< local name > ::= < universal name >,
< object type > = (DIRECTORY /
SEGMENT /
PERIPHERAL),
< restrictions >

This allows us to give the following information on the parameters
of the primitives:
< designation of object > ::= ( < pathname > / LN₁ ),
   < restrictions >

where < restrictions > allows to limit the use which may be done on objects passed through the parameter mechanism.

v. Environment

We remember that the environment is a reference directory for solving pathnames used by a procedure, it also specifies the capabilities of the procedure.

< environment > ::= < directory > ,
   < attribute > = 'environment' ,
   < capabilities >

< capabilities > ::= [ < primitives > ] ...

When a procedure call is made, the first parameter of the CALL is evaluated and if during this evaluation an environment is found as a node in the pathname then this environment will be taken as the new environment. If several environment are crossed, then only the last one becomes the current one. Figure 21 illustrates this mechanism.

In the case b) the directory of entryname C has the attribute 'environment', which is not the case in a); otherwise, the addressing spaces between a and b are identical. Let us suppose a process whose current environment is 0, and that the current procedure has the following primitive call:

CALL (C.A , A [READ], C.B)

The new top of the stack of the process is represented in both cases a) and b), but the reader will remark that in a) the current environment is still 0 while in b) it is 3. If, inside segment 4 we execute

LN₃ ← BIND (B)
this will give respectively objects 2 and 5. In case b), object 1 which has been passed as parameter is only namable through LN₁, with the restriction given through the call( read only is allowed ).

vi. Links

The links allow to access an object from a directory which is not on the direct pathname, the creation of a link is done by the CREATE primitive in which the <designation of initial value> represents the referenced object. This means that the designation mechanism always follows the same rule: the creator must name the referor and the referee. Restrictions may be made on the referenced object.

```
< link > ::= < type > = 'link',
         < referenced name >,
         < restrictions > to referenced name
< referenced name > ::= < universal name > [ < path name > ]
```

There must exist a special primitive to delete an object of type 'link' (DELETELINK), because a link acts like a descriptor (see 2.4.1 section ii.) and is invisible from the user.

vii. Example:

Let us consider figure 22, and suppose that at the beginning the only existing objects were 0 and 2, and that a process whose current environment was 2 created the addressing space, the following primitive calls may have been done:

1. CREATE (, A, type = 'directory')
2. CREATE (A,A, type = 'segment')
3. CREATE (A,B, type = 'segment', restrictions = READ / CALL)
4. CREATE (, C, type = 'directory', attribute = 'environment')
5. CREATE (C,B, type = 'segment')
6. CREATE (C,A, type = 'link', A.B, restrictions = CALL)

(3) created a segment which can only be read or executed, (6) created a link towards the previous object but this object can only be executed.
If object 5 is accessed from O through the pathname A.B or C.A the restrictions applied to it are not the same. Idem, if we say

\[ LN_i \leftarrow \text{BIND (C.B [CALL])} \]

the use of C.B through LN_i can only be a call.

3.3.4. Processes

i. Control point

The idea is to associate, when executing, a control action to each event. This action will be done if the event occurs (generalization of the interrupt mechanism). If we consider an environment as defining a sub-system, it is natural to define the domain of a control point as the sequence of the procedures, in the process stack, belonging to the same or to a nested environment.

Deletion of a control point is either explicit (REVERT) or implicit (when returning from the creating procedure).

ii. Synchronous and asynchronous events.

We associate to each elementary execution (machine instruction or primitive) a set of possible events. For example in the case of machine instructions one may find

- overflow in arithmetic
- addressing error
- invalid instruction

Events may also be programmed, such as an END OF FILE.

We distinguish two types of event:

1. Synchronous events which occur during the execution doing the fault
2. Asynchronous events which, although already bound to an execution, may occur after this execution. (For example the 'break' key on a terminal, or a time out).

iii. Operators.

\[
\text{ON < event > DO < control action >}
\]
Create a control point, and define the action to be taken when the event occurs.

```
< control action > ::= CALL ( ....... )
   NOTHING ,
   RETURN ,
   ABORT
```

CALL specifies the procedure to call
NOTHING acts like a mask, the occurrence of the event is ignored.
RETURN the current procedure returns, execution restart in sequence of its call.
ABORT the same effect as RETURN except that a predefined event is generated (ENVABORT).

```
REVERT ( < event > )
```

Delete a control point whose name is given as parameter.

```
STATUS ( < event > )
```

```
SIGNAL ( < event > , < message > )
```

Generates an event and the corresponding status will receive the message.

```
< event > ::= < primitive event > / < built event >
< primitive event > ::= OVERFLOW / UNDERFLOW / HARDTRAP / ...
< built event > ::= any name
```

any hardware trap
bound to the use of primitives
iv. Process creation, built events.

An asynchronous event is, for a process, an event associated with an activity parallel to the generating process. The activation of a new process (son) is done by

```
FORK (<event>, same parameters as for CALL)
```

The relations between a father and its son are done only through the concept of event. Several processes may be generated on the same event.

The death of the son or the use of a SIGNAL will generate an occurrence of the concerned event. This event may occur when the father is not in a nested (or identical) environment of the creation of the control point environment. In this case the control action is delayed until the previous conditions are true. The control point mechanism allows control of several processes by the same event.

Deletion of a process, also deletes the entire filiation of this process.

A process may wait on one or more events

```
WAIT ([<event>] ....)
```

Sequential execution of the current procedure will resume only when one of the listed events occurs, moreover the control action may have already taken place.

v. Example: (Figure 23) debugging aids. CALL (DEBUG, PROGRAM)

will execute the program named PROGRAM under the control of DEBUG. If an overflow occurs the segment OVERFLOW-TREATMENT will be called after pushing the execution of PROGRAM. The treatment of the fault may involve interaction with the user (through TTY), and execution may resume or abort, depending of the end of OVERFLOW-TREATMENT (either RETURN or ABORT).

3.3.4. The nucleus
The design of the nucleus which realizes the structure of the process, the addressing space and the storage of the values has been made using the same method as for the previous structures (see 1.4.3 and 3.3).

Objects here are modules, a module being a set of data associated with their access procedures; one module can never access directly the content of data associated with another module, it must always call an access procedure.

As consequences:

- as soon as the module semantic is bound, by the definition of its access procedures, the implementation of this module has no effect on the rest of the system. A particular realizator may now take care of it.

- every module may be replaced by an "equivalent" module with better efficiency for example. We will not define more precisely what "equivalent" means.

- every set of shared data is inside one module.

- critical section is always defined at the level of the module (exactly the control part of the object).

Processes are realized by a stack inside the nucleus, and module calls are procedural calls.

Every module acts like an environment: it is related to other modules which are accessible from this module; in the nucleus only calls to an access procedure change the addressing space of the process, the consequence is that there exist modules without ancestor, these modules are the entry point into the nucleus:

- DISPATCH, i/o SUPERVISOR, TIMER correspond to hardware interrupts

- TRAP correspond to user's trap (paging fault, non existent order code, and so on ...).

Figure 24 gives an overall view of the nucleus, and figure 25, the definition of the module DOIO.

3.3.5. Preliminary remarks
At this point we give some remarks, some of which will be explained in more detail later.

i. The type, control and value are intrinsic properties of objects and this for the addressing space of the subsystems or of the nucleus. While the protection (restriction on operators) is a property bound to the names, several names may refer to the same object with a different protection.

ii. Local names are very like descriptor in machine language. In the case of GEMAU a local name has the following format:

```
         Q
         o
     ↓ protection  universal name
     type
```

where protection is a composition of the restrictions accumulated on the pathname used to find the object and of the restrictions specified the < designation of object > in one of the following primitives:

CALL, RETURN, BIND

iii. Expandability.

The way the nucleus was built allows the introduction of new object types to be made easily; work are underway to include explicit synchronization on objects.

From the subsystem point of view expandability is rather forward since an object is used with its operators, and these operators take their significations only in the context of the object. For example, an object of type < peripheral > is an executable object and its only operator is CALL, literals are used to specifies what must be done by this object. For example, if LPT is a line printer, and S1 a segment

CALL (LPT, S1, 'write', n, displacement)

will transfer the n characters located at 'displacement' on S1 onto the physical line printer. If now we replace LPT by a procedure, from the user point of view there will be no change in the CALL instruction.
iv. Specialized processors.

The addition of new data types will make easy the use of specialized processors connected on the main memory. Works is underway to test the feasibility of this, this addition which will include new hardware processors.

v. Subsystem.

Several subsystems have been written on this system machine, and we can say that debugging and coding has been done faster. We also used extensively the substitution mechanism, to change on line, part of the different subsystems.

The subsystems include:

- card reader spooling sub-system
- line printer
- conversational subsystem
- batch subsystem.

3.4. General properties of a naming scheme

In the following subsections we will look, in more detail, to some system aspects and see their relation with the naming scheme of the machine.

3.4.1. Modularity

This is the property of computer systems to divide jobs into parts. A module being part of a program, it has known input values and gives some output values; this is exactly a black box.

A module defines some function to be performed, but no knowledge of inner working is needed; moreover the internal realization of a module may be changed without any effect on the outside world, as long as the function performed by the module has no been redefined, but this function can be, at least, expanded.

Examples of modules may be found in hardware: main memory (as seen before) acts like an object and the replacement by a physical block by some other block (technologically identical or not).
In software examples are in the nucleus of GEMAU; let us take the core management module (CORE), its access functions are BLOCGET and BLOCFREE. How the research of a free bloc is done when someone is requesting for a bloc of some size is irrelevant for the caller and in this case this is particularly useful because we may choose one algorithm or another one depending on the types of demands to improve the speed of the system.

We were talking about substituting one module realization by another functionnally equivalent, the question which arises now is when can this substitution be done? Before going further into this problem, let us look at what is binding. If we consider a user writing programs in a high level language, let say FORTRAN, we can see that several transformations occur before his program is effectively executed; figure 26 gives an example of such transformations. First he has to translate his problem in FORTRAN terms (1), then to punch his program, compile it and execute it. In fact more steps are needed because most of the times he wants to be able to incorporate library subroutines to his program. For example, no one wants to rewrite a "matrix multiply" routine every times he wants to use one, he will use a predefined from a set of useful routines. To incorporate this "matrix multiply" routine to his program may be done at different stages in the compilation-execution process. Let us look at some possible times:

1 : at source coding time, by copying (by hand) the FORTRAN statements.
2 : before compilation by incorporating a source card deck in FORTRAN
3 : at compile time if the compiler provides the user with the possibility of copying code from existing files
4 : after compiling but before loading, by inserting an already compiled "matrix multiply" routine in his compiled program
5 : when loading his program into memory, if for example, there exists a set of preloaded routines
6 : at execution time, which is what occurs in the GEMAU system? This is often called segmentation.

In each of these stages a binding occurs (figure 27).
1: problem variables are given symbolic names
3: variable names are replaced by displacements from the begin-
   ning of the program (this is called relocatable code)
4: several programs may be mixed to form new relocatable code
5: displacements are transformed into virtual addresses
6: virtual addresses are transformed into real memory locations

In general binding is irreversible, and if someone wants to replace
let say the "matrix multiply" routine by a new one, he will have to re-
tart the stage just before the binding occurs. The later the binding the
more flexible the system is.

The time of binding is very much related to the time of substitution,
and to the addressing structure of the machine. Let us take some
examples:

- IBM 360, CII IRIS 80 class of machines

Usually binding occurs at stage (4), and also for the system at sta-
ge (5) for non resident system routines, i.e. the routines of the system
that are not always present in memory (accounting procedures for exampl,e),
they are linked and eventually loaded only when called. A good descrip-
tion of the OS360 linker and loader can be found in\textsuperscript{50}.

- MULTICS\textsuperscript{39}

Procedures call objects by their symbolic names at once, then all
other calls are made by direct addressing, so bypassing the
name interpretation mechanism. In this case, every referenced object is
bound to the process which referenced it. The user is not responsible for
the time at which he binds objects.

- GEMAU

Here linking is done on every call by pathname, but binding time may
choosen by the user. Let us consider the two following programs:

\begin{align*}
\pi_1 & : \quad \vdots & \pi_2 & : \quad \vdots \\
& \text{repeat n times} & LN_5 \leftarrow \text{BIND}(x) \\
\text{begin} & \quad \vdots & \text{repeat n times} & \text{begin}
\end{align*}
call(x)
:::
end;
:::
call (LN_3)
:::
end;
:::

Apparently these two programs are identical and give the same result, but this view ignores that more than one process is existing in the system. If in parallel a second process deletes x and creates a new segment named x, in the case of \( \pi_1 \) two calls to x will call two different versions of x, and in the case of \( \pi_2 \) they will call always the old version which is bound in LN_3.

Possible realization of modularity.

We may realize modules either by procedure calls as in GEMAU or by messages switching among process as in THE, ESOPE. In the second case each module is always active independantly of others. Hardware gives some examples of both cases:

- procedure call : as seen before in the case of the memory
- messages : the i/o channels, or disk file optimizer as used by the Burroughs machines.

3.4.2. **Protection, capabilities**

i. User's point of view.

The user may want protection mechanism to apply to himself for debugging purposes or to protect some of his objects against their use by others (privacy).

To be more precise, the user wants to be able to write in any language anything and not destroy by misuse or facety the operating system or other users, or previously correct programs. Also the protection applying to him must be set up without changes in the system. In fact all these statements may be extended if we replace the word user by subsystem or operating system.

ii. Definition.

The protection is always a restriction on the operators applica-
ble on an object.

iii. Mechanism \(^{23,24,30,31,36,37}\)

Protection is always an interpretation by the system of the access to objects. Two different views of protecting access may be done (see figure 28).

In the first view every process may access every object (MULTICS for example), a list associated with every object must tell for each possible process what protection applies to the object if used by this process. In this case we find some batch system like SIRIS7 where an object may be private, in which case its access list is degenerated, no process except the owner being able to use the object. An object may also be public, all processes are able to access it with the same protection. Other controls may be made through the use of keys, or pass words, or rings as in MULTICS.

In the second view protection is bound to the pathname (GEMAU, B6500), a process may access an object only if it can name it and in this case protection is bound to the pathname.

The reader will find in \(^{38}\) a good model describing protection mechanisms.

In all cases the protection mechanism is bound to the naming scheme because if it is not, it acts just like a gadget, see examples in v.

iv. Capabilities.

Another way of enforcing protection is to restrict the use of some objects (operators, instructions,...), so they could not apply to any object because the user cannot use them. Even if a user can use OPl on object x, if he does not have OPl there is no way to apply it on object x. The privileged/non privileged mode of usual machine is an example of capabilities. In this case, in user mode, some instructions of the machine will either generate a trap when used or have a meaning different from privileged mode, depending on the machine.

v. Example

IBM 360
Main Memory is divided into blocks of 2k bytes, with each block is associated a 4-bit protection key. In the CPU and also for each the i/o channels an internal register of 4-bit exists. When a processor accesses memory, a check to the value of the key-register of the processor against the key of the block (in which are the bytes it tries to access) is made. If the two values are identical, the access is permitted; idem if the value of the key-register is equal to zero. Otherwise a trap occurs.

If the access cannot be performed because accessing is forbidden, then a trap occurs. The keys can only be changed in master mode.

- B3500

In this machine there exists two registers : base and limit, which define the starting address of a program and its ending address. Addresses inside the program are always relative to the beginning of the program. Every time an access is made the real location is found by adding the relative address to the content of the base register and checking that the upper bound limits of the program are no exceeded. These registers are not usable by the programmer.

3.4.3. Synchronization, Control

i. Processes$^9,^{10,27}$

As seen before there exist some independant computations, as for example in hardware with several processing units. A process represents the activity of some program using some data; there may exist several activations of the same program, we call them different processes.

At this stage let us return to the idea of intrinsic and technological properties of a system, and consider a program written by a user; this program is executing and putting some results of computation on a line printer. It is irrelevant, for the user, that this output be done as a procedural call or as a process call, i.e. synchronous or asynchronous to the execution. But from the realization point of view this is an important problem, because it may change the efficiency of the program. We will say that, in this example, program execution and physical input-output have no timing relationship each other, and what happens during execution does not alter an old request for output. We shall say that the
two processes are parallel. To be able to run a program needs some resources, as seen in GEMAI for example it needs to bind objects to its execution space; other processes may use the same program or different programs with some common objects, there may exist competition between processes, for example if one of the object is the last tape driver in the installation. Now let us go to the level of realization, processes have to be implemented on some set of resources: central processing unit, main memory, disk storage,... To be able to physically run a process needs to have the CPU and also some space in main memory, these resources are shared among several processes and there are competition or concurrency between processes for running, so there is a need for synchronizing processing and managing resources.

ii. Critical section.

The previous example shows that concurrency at one level disappears at upward level. A user process does not know all the problems to allow it to run in concurrence with other processes on the same hardware, this is the first reason for virtualization (VIRTUAL PROCESSOR). Let us still stay at the level of the user, there is a need for explicit synchronization with other processes, just because the logical problem imposes one. For example, in the accessing of shared data, where some user may modify them and some others just read them, it is not possible to allow someone to modify an information another one is reading, just because the modification operation is not a unique machine instruction and the coherence of data (pointers in links..., see next section) may be affected for some time. We shall say that these data are in a critical section. Let us take the following example:

a vector \( V \) is a set of \( n \) word of memory and one operation is possible:

\[
\text{move}(V, W)
\]

where \( W \) is a vector, the algorithm of \text{move} is

\[
\text{for } i := 1 \text{ step } 1 \text{ until } n \\
\text{ do } \ W[i] := V[i] ;
\]

Now suppose two procedures \( \pi_1 \) and \( \pi_2 \) activated by two processes \( P_1 \) and \( P_2 \)
\[ \pi_1 : \text{local vector } X [1 : 2] ; \]
\[ \text{move} \ (V, X) ; \]
\[ \text{if} \ (X [1] = 1 \ \text{and} \ X [2] = 3) \ \text{then} \ A_1 \ \text{else} \ A_2 ; \]
\[ \pi_2 : \text{local vector } Y [1 : 2] ; \]
\[ \text{move} \ (Y, V) ; \]

let us suppose that

\[
\begin{align*}
\text{initial value of } & V : V [1] = 1 & n = 2 \\
& V [2] = 2
\end{align*}
\]

and let us represent the sequences of operations executed by the two processes:

\[
\begin{align*}
\pi_1 & : 1 \ X (1) := V (1) & \pi_2 & : 1 \ Y (1) := 0 ; \\
& 2 \ X (2) := V (2) & 2 \ Y (2) := 0 ; \\
& 3 \ \text{test} \ X (1) = 1 & 3 \ V (1) := Y (1) ; \\
& 4 \ \text{test} \ X (2) = 2 & 4 \ V (2) := Y (2) ;
\end{align*}
\]

Then two possible real sequencing in the processor:

**Case 1**

P1.1, P1.2, P2.1, P2.2, P1.3, P2.3, P1.4, P2.4

The result is: X(1) X(2) V(1) V(2)
1 2 0 0

**Case 2**

P2.1, P2.2, P2.3, P1.1, P1.2, P1.3, P2.4, P1.4

The result is: X(1) X(2) V(1) V(2)
0 2 0 0

The result of the operations is non deterministic. Suppose that the user wanted to read the vector in a single instruction (as the move may have seemed to do it for him) or modify it by a single instruction. Then there must exist a way to tell the system that move is in a critical section, i.e. the sequence of execution of its instructions cannot be broken as in ca-
ses 1 or 2. We say also that P1 and P2 are mutually exclusive when using move. The question is then, how to do that:

- explicitly or implicitly?

From the user point of view the answer is very often: implicitly, he will not have to synchronize, in its program, the call to move with other processes. GEMAU offers this kind of synchronization by means of the multiprogramming degree.

Explicit synchronization can be made by a lot of different mechanisms, the simplest and widely known is implemented by means of locks.

A lock is a boolean variable, on which the two following operations can be performed:

\[
\text{lock } (V) : = \left\{ \begin{array}{ll}
\text{if } V = \text{true then goto lock else } T := \text{true} \\
\end{array} \right. \\
\text{unlock } (V) : = [V := \text{false}];
\]

where \([\ ]\) means that these operations are not interruptable (it is a critical section). From this example the reader can see than to realize critical section there must already exist some kind of primitive critical section, this is a fundamental concept and must be realized at the hardware level. On some machine we find a "test and set" operation which read the value of a memory cell and rewrite some value in it, and this in an exclusive fashion. In previous examples we have already seen this kind of exclusion in memory: fetch and store are done in an exclusive way.

iii. Semaphores.

The mechanism to synchronize by way of locks is, very often, too basic and also the waiting is active. E.W. Dijkstra introduces the mechanism of semaphores. Instead of active waiting, processes are put in some dormant state were they do not compete for physical resources (CPU, memory, ...) we say that they are put into a waiting queue. Now we try to describe semaphore: a semaphore is a couple \(S = (s, Q)\) where

\[
\begin{align*}
& s \text{ is an integer} \\
& Q \text{ a queue}
\end{align*}
\]

and the three possible operations are: I, P and V;
I(S, s₀) : [ if s₀ > 0 then S . s := s₀ ]

P(S) : [ S . s := S . s - 1;
       if S . s < 0 then
       begin
         current process state := dormant;
         put it in S . Q;
         find a ready process to activate;
       end ];

V(S) : [ S . s := S . s + 1;
       if S . s ≤ 0 then
       begin
         remove one process from S . Q;
         activate it;
         comment current process is still active;
       end ];

These mechanisms shows a higher level of realization than lock. The example of the vector can be written:

procedure Move (V, W);
begin
  P (mutex);
  for i := 1 step 1 until n do w(i) := v[i];
  V (mutex);
end;

where mutex is a semaphore which initial value is equal to 1 (I(mutex,1)).

iv. Problems with mutual exclusion.

If a process aborts or is deleted in the middle of a critical section, it will block other processes, so it is necessary to detect that this process is inside a critical section and artificially do a V; but then we are faced with a logical problem: if the user put a critical section it is for some imperative reasons (coherence for example) and an arbitrary decision made by the system will almost surely be bad. We see, a little later, that a naming scheme of the type of GEMAU solve this problem (see vi).
v. Deadlocks.

Processes sharing variables need to synchronize themselves, for this they can use semaphores. Let us take the two following programs:

\( \pi_1 : \) \text{global } A, B; \\
\text{semaphores } S_A, S_B; \text{ comment initial value } = 1; \\
P(S_A); \text{ comment block processes which want to use } A; \\
\vdots \\
do \text{ something with } A; \\
\vdots \\
P(S_B); \\
\vdots \\
do \text{ something with } B; \\
\vdots \\
V(S_B); \text{ comment release the use of } A; \\
V(S_A); \\
\vdots \\

and

\( \pi_2 : \) same global as \( \pi_1 \) \\
P(S_B); \text{ comment take } B; \\
\vdots \\
P(S_A); \text{ comment take } A; \\
\vdots \\
V(S_A); \text{ comment release } A; \\
V(S_B); \\
\vdots \\

The figure 29 shows the possible progress paths of both processes, one sees that if we have the following state

\[ P_1 : \text{ has done } P(S_A) \implies S_A^s = 0 \]
\[ P_2 : \text{ " " } P(S_B) \implies S_B^s = 0 \]

then the system is blocked, because to progress \( P_1 \) will do \( P(S_B) \) which will block it and \( P_2 \) will do \( P(S_B) \) give \( \text{ have the same result. Each of the two processes is waiting for the other one: dead end.} \)
There exist several solutions to this problem, we can class them in two parts:

- curative
- preventive.

A curative solution imposes that we can detect (which is not easy) when processes are in deadlock situation, then to remove the deadlock to kill one process and we are back to the problem of critical section announced earlier.

Preventive is the best, and we see that if we impose that an order exists between semaphores the problem will disappear. Here we may say that one must no do $P(S_A)$ if it has done $P(S_B)$ before.

vi. Critical sections. Case of GEMAU.

The naming scheme of GEMAU allows for controlling deletion and interruption inside critical section. Remember that there is a multiprogramming degree associated with every object. Let us take, as example, figure 30.

The process $P_1$, (executing in environment 3 object 5), creates a control point by

```
ON son DO NOTHING
```

then creates a son process by

```
FORK son CALL (B, B)
```

The new process is in environment 6 executing object 8, and suppose that in object 8 we have

```
CALL (A)
```

This will push the stack of the son and put it in environment 2 executing object 4; now that $P_1$ executes an order of end of execution of object 5

```
RETURN
```

Then the control point 'son' is deleted and process $P_2$ must be deleted, but has it is in a non nested environment (from $P_1$ current environment).
The deletion is delayed until P2 returns from 4 (returning may be an explicit RETURN or an ABORT from inside 4).

This will bring it in environment 6 which is included in 3, so the execution of 8 will be aborted, and so on, until the process is terminating the initial procedure; then P1 may resume. If P2 never returns from 4 that means that object 4 is not reliable and only a process in an environment seeing 2 will be able to do the destruction. This is exactly what happens on current system when a call to a system procedure fails.

3.4.4. Reliability, coherence of information

i. Discrete value of an object.

Let us take as example a time sharing system where a user is editing a file which records are card images. The user will insert new records, delete some old ones or even change some. Let say that his program (the editor) is in main memory making the appropriate changes to the file. An operation of inserting a record will modify the different variables describing the internal structure of the file, for example if a list type structure is choosen. These modifications are not elementary at the level of the system and they may involve several input/output transfers, we will say that during the execution of the operation the internal value of the file is not coherent; for example the forward pointer of a record is written but the backward pointer is not yet updated. We shall say that the object file has only discrete values between operations, or may be only for some of the operations if they do not modify the value of the object.

Suppose now that the system fails, this failure being either software or hardware, several possibilities exist:

- the crash occurs between two operations on the object, its value is still defined
- the crash occurs during an operation on the object, its value may be correct but the coherence of it is not sure.

Let us find why there is a problem, and then look for solutions. There is a problem because most of the time after a failure some informations are lost (very often processor state, main memory and hopeful-
ly very few time background storages such as disks, drums and tapes). The system being, as seen, composed of heterogeneous elements, each of them being asynchronous from the other; values will be correct on some elements and incorrect on others.

The solution is to restart with the maximum of coherent information (disk, drum, and sometimes main memory). But this information is only coherent if it has been modified at some discrete time, which means that we have to keep an image of an object every time we modify it, until the modification is completed. If a crash occurs we restart from the last correct image.

ii. Mechanisms.

We shall talk only about some software mechanisms. The system, every time part of an object is modified can take a copy of the previous value or a copy of the modification. The question arises, at this point, of who decides to make a copy: the system or the user. In conventional batch system the technique known as checkpoint restart is very often used. The user decides that every given slice of time a complete copy all his objects be made, and in case of crash to restart with the last or at least previous copy. In modern systems, and in the case of data bases copies should be taken without the knowledge of the user. But there still exist the problem of deciding when to update the object value. In the case of GEBAU we decided that the user (or the subsystem) does it explicitly through a SAVE primitive.

3.4.5. Conclusions

The reader sees that the architect's job is not easy and need a good knowledge and understanding the various parts of the computer science. For example, in the system point of view it is almost impossible to design protection without defining how the naming scheme will be.

The other problems we did not treat in these sets of lectures concern primarily the area of realization and especially of scheduling the real resources to implement the virtual processor and memory notions. This depends heavily on the performance needed and on the availability of adequate technology. The designer should always keep in mind that he must
leave an open-door, to be able to expand its system when new technologies become available. System measurement is necessary to improve the realization.

If we consider scheduling, one sees several levels at which a scheduling is done: on a short term basis (like paging mechanisms, caches or pipe-lines in hardware), or on a medium term basis which can take the properties of the manipulated objects into account to allocate resources. For example, the structuration of programming allows very often the system to know the working set of a program, and to predict its behaviour.

We will not treat how to realize virtual memory (paging, segmentation, caches, memory hierarchy) nor the virtual processors (mono or multiprocessing systems).

But these issues are very important and the designer should not forget them, but he must use the requirements of system and language of the user as a primary design consideration, the other considerations will be said secondary.

In fact one may, very often, improve the performance by using the concept of parallelism or by distributing the processing power (remote processor, specialized processor,...), this can be done if the design is neat.

Lectures by Pr.Sumner and Dr.Mazare treat some of these aspects.

### REFERENCES

   CS 70 158, 1970
9) R.C. DALEY and J.B. DENNIS, "Virtual Memory, Processes and Sharing in Multics", CACM, May 1968
12) P.J. DENNING, "Virtual Memory", Computing Surveys, Sept. 1970
14) J.B. DENNIS, "Segmentation and the design of multiprogrammed computer systems", JACM, 12, 4 - Oct. 1965, pp. 589-602
20) D.A. FISHER, "Control Structures for programming languages", Burroughs Corp., TR 70-6, May 1970
21) FRASER, "On the meaning of names in programming Systems", CACM, June 1971
26) P.B. HANSEN, "The Nucleus of a Multiprogramming System", CACM, April 1970
30) B.W. LAMPSM, "Dynamic Protection Structures", FJCC 1969
35) MAC KEEMAN, "Language-directed Computer Design", FJCC, 1967
44) Software Engineering, Nato Science Committee, Garmisch 1969 and Rome 1970
47) H. WEBER, "A Microprogrammed Implementation of Euler on IBM 360/30", CACM n° 9, 1967
48) P. WEGNER, "The Vienna Definition Language", ACM Computing Surveys, March 1972
<table>
<thead>
<tr>
<th></th>
<th>49-56</th>
<th>57-64</th>
<th>65-+..</th>
</tr>
</thead>
<tbody>
<tr>
<td>technology</td>
<td>vacuum tubes</td>
<td>transistors</td>
<td>integrated circuits</td>
</tr>
<tr>
<td></td>
<td>ms, some µs</td>
<td>10 µs, some µs</td>
<td>MOS LSI, MSI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>some 100 ns, and less</td>
</tr>
<tr>
<td>main memory</td>
<td>magnetic drum</td>
<td>core</td>
<td>integrated circuits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOS LSI</td>
</tr>
<tr>
<td>secondary</td>
<td></td>
<td>magnetic tapes</td>
<td>large scale magnetic devices (random access)</td>
</tr>
<tr>
<td>storage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peripheral</td>
<td>paper tapes</td>
<td>punched cards</td>
<td>terminals</td>
</tr>
<tr>
<td>devices</td>
<td></td>
<td></td>
<td>(TTY, CRT display, ...)</td>
</tr>
<tr>
<td>languages</td>
<td>assembly (very simple)</td>
<td>FORTRAN</td>
<td>ALGOL 68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALCOL 60</td>
<td>PL / 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL</td>
<td>APL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assembly</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>do it all by yourself</td>
<td>batch processing</td>
<td>multiprogramming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I/O management</td>
<td>time sharing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>virtual memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>computer network</td>
</tr>
<tr>
<td>debugging</td>
<td>oscilloscope</td>
<td>memory dumps</td>
<td>interactive compilers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and interpreters</td>
</tr>
</tbody>
</table>

*figure 1: historical evolution of computers*
Figure 2: Price evolution of some small computers

Figure 3: The complexity wall
# Figure 4: Methodology

<table>
<thead>
<tr>
<th>USERS</th>
<th>TOP DOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM/LANGUAGE</td>
<td></td>
</tr>
<tr>
<td>ASSEMBLER</td>
<td></td>
</tr>
<tr>
<td>MICROPROGRAM</td>
<td></td>
</tr>
<tr>
<td>LOGIC DESIGN</td>
<td></td>
</tr>
<tr>
<td>CIRCUITRY</td>
<td></td>
</tr>
</tbody>
</table>

# Figure 5: The Program / Processor Relativity
multi-language processor

specialized processor

figure 6: classes of processors

<table>
<thead>
<tr>
<th></th>
<th>interpreters</th>
<th>compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interpreter 114000</td>
<td>editor 6000</td>
</tr>
<tr>
<td></td>
<td>garbage collector 36000</td>
<td>syntax 6000</td>
</tr>
<tr>
<td>EULER</td>
<td>150000</td>
<td>code generator 18000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I/O interface 3600</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>26200</td>
<td>compiler 27400</td>
</tr>
<tr>
<td>B1700 (FORTRAN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I/O interface 2100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29500</td>
<td></td>
</tr>
</tbody>
</table>

figure 7: compared size of compilers and interpreters (in micro-instructions)
<table>
<thead>
<tr>
<th></th>
<th>FORTRAN compaction</th>
<th>COBOL compaction</th>
<th>RPGII compaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1700</td>
<td>280</td>
<td>450</td>
<td>150</td>
</tr>
<tr>
<td>B3500</td>
<td>450 40%</td>
<td>1200 60%</td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>560 50%</td>
<td>1490 70%</td>
<td></td>
</tr>
<tr>
<td>S/3</td>
<td></td>
<td></td>
<td>310 50%</td>
</tr>
</tbody>
</table>

scale: in k bits

Figure 8: Comparative sizes of programs

<table>
<thead>
<tr>
<th></th>
<th>A := (B + C) x (D + E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load,R1</td>
<td>B</td>
</tr>
<tr>
<td>add,R1</td>
<td>C</td>
</tr>
<tr>
<td>load,R2</td>
<td>D</td>
</tr>
<tr>
<td>add,R2</td>
<td>E</td>
</tr>
<tr>
<td>mult,R1</td>
<td>R2</td>
</tr>
<tr>
<td>store,R1</td>
<td>A</td>
</tr>
<tr>
<td>name</td>
<td>A</td>
</tr>
<tr>
<td>load</td>
<td>B</td>
</tr>
<tr>
<td>load</td>
<td>C</td>
</tr>
<tr>
<td>add</td>
<td>D</td>
</tr>
<tr>
<td>load</td>
<td>E</td>
</tr>
<tr>
<td>add</td>
<td>R2</td>
</tr>
<tr>
<td>store</td>
<td>A</td>
</tr>
</tbody>
</table>

Op-code address registers

<table>
<thead>
<tr>
<th></th>
<th>6 x 6 = 36</th>
<th>6 x 9 = 54</th>
<th>6 x 9 = 54</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 x 6 = 96</td>
<td>16 x 9 = 144</td>
<td>16 x 5 = 80</td>
</tr>
<tr>
<td></td>
<td>4 x 6 = 24</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Size of program

|          | 156        | 198        | 134        |

Compaction ratio

|          | 21%        | 0%         | 32%        |

Comparative figures on program sizes

- one address 16 registers no tag field single length instruction
- one address stack no tag field single length instruction
- one address stack no tag field variable length instruction
figure 10: example of objects

figure 11: synchronization on access to an object
figure 12: multiplexing and synchronization

figure 13: components of an object
begin
  integer X ;
  integer Y ;
procedure P1 ( A : integer ) ;
begin
  real X ;
  X := 0 ;
  Y := A + X ;
end ;
begin
  integer X ;
  integer Y ;
level 1
  
  P1 ( X ) ;
end ;
end ;

figure 14 : calling context and execution context
figure 15: different transformations of a name

figure 16: file address space of SIRIS7
figure 17: sub-system hierarchy

different sets:
° $S_{PN}$: names in the tree (pathnames)
° $S_{LN}$: local names
° $S_{UN}$: universal names
° $S_{O}$: objects

figure 18: different addressing spaces in GEMAU
Figure 19: Example of naming space in GEMAU
$\mathcal{E}_c$: current environment

$\implies$: this directory is an environment

Figure 20: Example of execution spaces
Figure 21: Environment mechanism
figure 22: restrictions
DEBUG : ON OVERFLOW DO CALL ( OVERFLOW-TREATMENT, LN1 );

ON CALLABORT DO CALL ( ABORT-TREATMENT );

ON BREAK DO CALL ( BREAK-TREATMENT );

ON ENVABORT DO NOTHING;

FORK BREAK CALL ( TTY );

CALL ( LN1 );

RETURN;

figure 23: debugging aids
figure 24: the nucleus of GEMAU
an entry in the process stack:

- internal general registers
- address of current instruction
- master/slave flag
- real/virtual flag
- identity of current module
- address of local storage (for reentrant code)

figure 25: definition of a module
figure 26: different stages from compilation to execution
main program

integer I;

I := I + i;

library

figure 27: different kinds of binding
view 1: access list

view 2: environment

figure 28: different mechanisms for protection
figure 29: deadlocks
figure 30: critical sections in GEMAU
MULTIPROCESSOR SYSTEMS

Guy MAZARE,
CII Scientific Centre, Grenoble, France

1. INTRODUCTION : DEFINITION OF "MULTIPROCESSOR SYSTEMS"

1.1. Why several processors?

Contrary to the majority of usual (monoprocessor) systems, multiprocessor systems are characterized by the use of several processing units capable of doing work. The principal motivation for this is the desire to go more quickly with several units rather than with just one. In fact, it is necessary to make a distinction between two very different aspects of this motivation.

First, efforts in the past have been largely towards making ever more powerful machines which would operate more and more rapidly. But in each computer generation, performances were limited by the technologies used; one can of course imagine still more rapid technologies, but there exists a limit: the speed of light is limited, electrical impulses take nano-seconds to cover the distances separating the different units. Besides the needs of the users continue to grow and there will be progressively more and more complex problems calling for longer and longer solutions than those we deal with presently. To meet this need, an effort has been made to use several processors and accomplish several tasks at the same time. "Giant" machines were thus created (as Iliiac IV\textsuperscript{8}).

But in the final analysis this concerns only a very limited part of the applications, as the section of the computer market represented by these large systems remain very small. In addition a rapid technological evolution is occurring with the apparition and use of medium and large scale integration (MSI, LSI): instead of increased performances, one notes that the prices of medium sized processors are lowering. Whereas up to the present the performance/price ratio was much better for the powerful processors and the large systems than for the small ones, a trend in the opposite direction now seems to be developing. This is confirmed and
accentuated by the present development of the mini-computer market. As a result, computer architects are seeking to replace large systems by a number of small ones, or to reconstruct powerful systems by using several processors of weaker performances.

The above partially explains the "state of the art" in the field of multiprocessor systems: an older and sustained interest turned, for the most part, towards a search for high performances and towards scientific applications; and, on the other hand, a more recent interest in all types of configurations and applications, but which has at this time given few concrete results.

1.2. How to use several processors

1.2.1. Separated machines

The first approach consists in using N configurations, each including a central memory, a central processing unit and peripheral units. The configurations can remain completely independent from each other: the work being distributed among them manually by the action of the operators; in the same manner the data files are assigned to them.

One example of this would be the replacement of a time-sharing system (TS) by N mini-computers, each user of the TS working alone and directly on the mini. This trend exists and even appears viable to users having a fairly constant workload; in fact, its disadvantage resides more in the inefficient use of resources than in the weak performances of the hardware (response time). In effect, the editing manipulations and the reflexion time of the user immobilize the system for relatively little work.

Another example is that proposed by Even 6): a very large sort can be accomplished by N mini-computer systems, each including four magnetic tape units. It can be shown that the sort is processed N times more quickly than on a single processor, if one divides the files into N sub- files for sorting, and distributes the latter manually over all the mini- systems. This obviously takes time and the results are viable only if the tasks to be done in a parallel fashion - here the partial sorting - are sufficiently long. This can be expressed by saying that parallelism is
only usable on a gross level.

1.2.2. Networks

These disadvantages can be largely eliminated if the mini-systems are connected by I/O lines: the allotment of data and of works can be automated. First of all this permits the distribution of data between the systems and the sharing of data at a lesser cost than that of a complete set-up. This should also permit a better division of the workload by permitting execution of the work by any one of the systems. And since the exchange of work and data is more rapid, a finer parallelism is possible: the tasks to be done in parallel can be of a shorter duration, as long as the initialisation of the work remains cheap compared with the length of execution time. Of course, all this must be done at the price of a more complex organization - but which is accomplished by the machine.

1.2.3. Several connected processors and memories

The processors and other components of the system can also be more tightly connected. "Parallelism" can thus be used at a still finer level than formerly. This is done by grouping several processors around a common central memory and by permitting these processors to exchange certain signals - whether it be specialized processors, for example the I/O channels, or non-specialized processors. In other systems, this is done by closely connecting, by buses, a certain number of processors which may or may not possess a local store.

In any case, the programmer (or the compiler) remains conscious of the different processors which his program puts into play, and he must therefore have a thorough knowledge of the hardware used.

1.2.4. Several distinct functional units at the interior of the processing unit

Here we are concerned with all possible uses of "parallelism" inside the processor itself, while executing a number of tasks simultaneously.

Parallelism is of course transparent to the programmer, as it is detected and put into operation by the hardware. Insofar as the processor is the unit which executes the machine code instructions written by the
programmer, such systems are not multi-processor systems. But if, by processor, one means something capable of realizing a more elementary function, then it may be included in the field under discussion.

1.3. **Advantages of multi-processor systems**

A first advantage has already been mentioned: the gain in execution speed, which can extend to N if one uses N processors. However, one is not always able to use them all at the same time: certain tasks calling for a certain amount of sequential work, there is relatively little parallelism. In addition, this increases the overhead of the operating systems. Several studies $^{10}$ $^{13}$ have been made and tend to show that the gain is rather proportional to LogN.

Another advantage of this system is the increase in reliability, which results from the structure comprising several processors: if each function can be executed at several points in the system, dynamic reconfiguration can be performed, and a "reduced functioning" with the aid of the remaining elements is possible.

Thus it becomes possible to accomplish (under system control) maintenance tests on the different units without stopping the overall functioning, and to detect failures before they cause damage. On the level of 1.2.1. and 1.2.2., this is fairly obvious; but it is less obvious at 1.2.3. and 1.2.4., and it is the operating systems responsibility to organize the processors and the tasks to be executed.

1.4. **Multiprocessor systems considered here**

We are interested here in the multiprocessor systems 1.2.3. and 1.2.4., and will present two examples of the first and one of the second. In fact, it is these which are the originals, the others, whatever their advantages, being composed of mono-processor systems and a connection system (hardware, software or file and data manipulation organization).
2. THREE DIFFERENT TYPES OF MULTIPROCESSOR SYSTEMS

2.1. Array processor

2.1.1. Principles

An "array processor" is composed of a control unit, which fetches the instructions, decodes them and transmits the execution orders of the operations to a great number of arithmetic and logical units called "processing elements" (PEs).

Consequently the PEs execute all at the same time the same instruction. They are arranged in array, that is, connected each to their four neighbours: in this way they can exchange data. They each dispose of a small quantity of local memory, with which they work.

The benefit of these machines lies essentially in the highly parallel execution of all matricial or vectorial operations: if one places the vectors with one element per local memory, in each PE, the addition of two vectors, for example, will be made in one single step, each element carrying into effect an addition.

2.1.2. ILLIAC IV

The diagram of this is given in Fig. 1.

ILLIAC IV is an array of 256 PEs regrouped in 4 subarrays of 64, each one driven by a single control unit (CU). Each PE manifests relatively high performances: it works on 64 bit words and accomplishes a floating-point addition in 240 ns, a floating-point multiplication in 400 ns. Each one can communicate with its four neighbours by means of special instructions.

The whole system can function as one array under the direction of a single CU; it can also work as two sub-arrays, or even as four. Each processor disposes of a local memory of 2 K-words, where its data is stored; in addition, it disposes of a local index: local memory addressing can use this index which permits a certain flexibility in the use of all these elements.

The system does not possess a central memory: this is represented only by the 256 local memories. I/O, as well as loading operations of
these memories, are managed by a peripheral computer: this is in fact a B 6500.

The organization of data, and their distribution in these local memories, present some difficult problems. In effect, if one wishes to execute, for example, the product of two matrix, say $U.V$, it would be necessary to arrange $U$ by row and $V$ by column, in such a way as to achieve in one operation the product of a row of $U$ by a column of $V$; but if $V$ is arranged thus, difficulties can raise in other cases.

The proposed solution $^8$ consists in arranging the matrices "diagonally" which permits every operation: that is what we call "SKEWED array storage" (see fig. 2).

2.1.3. Advantages and disadvantages of this type of organization

Such organization is obviously very effective for treating certain problems in which can be found a great number of vectorial or matricial operations. This is the case with certain scientific applications and that makes of ILLIAC IV a highly specialized machine which is well situated as such in a network like ARPA.

Even in the best-suited operations, there remains a non-negligible share of simple operations, which develop in sequence on a single processor; besides, the matrixes used do not necessarily have the exact size required (64). This leads to unused processors.

In addition, one must know how to organize all these processors and these memories. In particular, all these vectorial operations must be made clear as vectorial operations, and not be sequentialized by the use of a sequential language - such as FORTRAN. This is one of the software problems which will be treated in § 3.

2.2. Pipe-line processors

2.2.1. Principles

The principle of "pipe-line" architecture is the following: suppose we wish to have a processor execute a series of operations each lasting $T$ time units; if each one can be broken down into $n$ steps of duration $t$, and if these steps are independent from each other, one can construct the
processor by \( n \) independent stations, each specialized in the treatment of one step (see Fig. 3). This permits the whole set to initiate a new operation every \( t \) time unit rather than \( T \). The execution time of each one stays the same, but the throughput is multiplied by \( n \).

Such an architecture is therefore achieved by the division of the processor in \( n \) stations (very specialized) and the addition between two stations of intermediary registers destined to receive the result of the upstream station and to wait there to be treated by the downstream station (see Fig. 4). The "operation flow" runs off then from station to station, and is treated in a continuous fashion - whence the name of "pipe-line".

2.2.2. **Examples**

(We will not enter into great detail here, but merely refer the reader to the report of Prof. Summer).

The pipe-line mechanism can be put into operation on different levels:

i. in the realization of an operator - that is, for example, the floating-point adder: four steps can be distinguished here: subtraction of exponents, shifting of a mantissa, addition of mantissae, normalization. One sets up then the operator on a schema of a four-stations pipe-line;

ii. in the setting up of a processing unit, there are several distinct steps for the execution of an instruction. The simplest technique ("overlapping") uses two steps:
step 1: instruction fetch and decode, address calculation, step 2: operand loading, treatment.

However, more evolved machines make more distinctions: up to ten steps in the present high performance machines (CDC 6600, 7600, IBM 360-91).

2.2.3. **Advantages and limits**

This kind of organization presents an obvious interest: it increases the throughput \( n \) times, while adding a minimum of hardware, or it keeps it while using less rapid components and therefore less expensive
ones.

Unfortunately, the problem is not really so simple: it is necessary to know how to manage and control all the access conflicts, if the different stations can utilize one same unit (memory or address calculation); moreover, the instructions can be dependent: thus the \((i+1)^{th}\) can use the result of the \(i^{th}\), or the operation code of the \((i+1)^{th}\) can be modified by the execution of the \(i^{th}\); and all the branch instructions break off the instruction flow. All of this is controlled by supplementary hardware which can become very complex; and yet the whole may be relatively inefficient if there are many branch instructions or other conflicts.

For this reason it is interesting to concentrate such architecture on repetitive operations - for example vectorial operations: as the work is repeated for all the elements of the vector, one can organize a very efficient pipe-line. The other solution consists in having a processor execute \(n\) independent programs, written as though they were to take place in parallel, on distinct processors: the only real processor taking up, by turns, an instruction from each sequence (for example: the STAR system). Dependency conflicts then no longer occur and the necessary hardware is considerably reduced (see Fig. 5).

2.3. Multiple independent processors (MIP)

2.3.1. Principles

Here we are concerned with organizations in which several processors (instruction processing units) are connected to a central memory. The processors are totally independent, exchanging at the most a few elementary signals; they communicate by the central memory and can each execute any instruction.

Such systems, composed of two, three or four processors, have been sold for several years (IBM 360-65, CII IRIS 80, GE 600 serie). Other experimental systems, comprising a greater number of processors, exist (C.mmp \(^{16}\) has 16 processors). One project under study at the CII is concerned with the realization of such a system, but carrying a large number of slow processors.
2.3.2. Interest

The first advantage of those architectures is in increasing the performances of a limited equipment by using a number of samples of the same processor and by adding the least possible quantity of hardware; but each unit must be capable of executing all the functions of a processor. Since the cost of mass-produced processors is steadily decreasing, such an organization has become economically interesting.

Another advantage is the increased reliability: as the processing equipment exists in n samples, there results a possibility of "reduced functioning" (see § 1.3.). Moreover, such systems are modular, and allow the constitution of nearly continuous power series.

2.3.3. Interconnection problems

The creation of this type of systems depends on the choice of a connection system between the different components.

i. Processor to processor liaisons:
   as soon as the number of processors becomes considerable, each processor cannot be connected to all the others - other than by a communications bus. In general only elementary signals can be transmitted in this way, and the processors must exchange their information by means of the central memory: here will be organized waiting queues of tasks to be performed and the processors must come here to find work. The elementary signals transmitted will then be of the type "request for registering of a task which has just been placed in a waiting queue".

ii. Processor to memory liaisons:
   the memory which is used by all the processors is organized in banks, in order to allow simultaneous access. A processor-memory connection by means of a crossbar-switch is possible only if the number of processors and banks remains small. Otherwise this matrix becomes very complex. Therefore solutions are being sought which comprise one bus per processor, or even a single general bus. However, it is this bus which risks creating a bottleneck.
iii. Processor to exchange unit liaisons:
here again diverse solutions are possible. One can assign a pro-
cessor to one or several exchange units: this processor is the
only one to exchange signals with that unit (such as SIO, or in-
terrupt), and it dialogues with other processors and with the opera-
ting system according to the general mechanism. Other architectu-
res exist in which a single master-processor controls all the in-
terruptions and the work of all the other processors (such as the
GE 600 serie); the drawback in this is the decreased availabil-
ity: if the central processor breaks down, no functioning is pos-
sible without it. Lastly, other systems centralize all interrupt
requests in order to assign the most pressing tasks to the proces-
sors (such as in the CII IRIS 80) (see 7). Here again, this in-
terruption unit is indispensable but represents much less hardware
than a processor.

2.3.4. Utilisation problems
In order to use these architectures, several independent tasks to
be conducted "in parallel" are always necessary. This is a problem which
becomes crucial as soon as we envisage such systems composed of a large
number of processors. Therefore, one must be able to show the parallelism
in these tasks, and there exists for that different techniques which will
be detailed in § 3.

2.3.5. Performances
The calculation power of the system increases when one increases
the number of processors; but this does not mean that the instruction
throughput will be augmented: all these processors utilize together a
certain number of resources, the memory for example, and this is where
the bottleneck is! In fact, such systems must be well balanced to avoid
under use of the equipment, resulting from a bottleneck in the architec-
ture.
Concluding remarks

The three types of multiprocessor systems which we have just described illustrate in fact the three existing manners of making several "processors" work simultaneously. This is true in the organization of the hardware components of a system, as we have seen; but it also remains true in all kinds of organization whose goal is to coordinate the work, in parallel, of several cooperating processing units, whatever they be: men or machines.

Let us imagine a certain rather repetitive task to be done: for example a truckload of bricks to be unloaded and placed in a heap a little further on; and let us suppose n workers or slaves on the worksite to accomplish this.

A first way of proceeding consists in attributing to each worker one or several piles on the truck and in having him reconstruct these where the bricks are to be placed. Each one gets on the truck, takes a brick, gets off, carries the brick to the designated place and leaves it, then repeat the action. If the truck is big enough for all workers to get on at once, there is no slowdown due to the fact that they are numerous, and the truck is n times more quickly unloaded. The workers have no initiative and operate at the same time; if their work were more complex, one could imagine an overseer shouting out the orders ("take a brick .. get down .."). Obviously this kind of organization works only under certain conditions: first that the bricks be arranged in N orderly and distinct piles, and then that the piles to be created be of the same number. If it were necessary to prepare them, prior to unloading them, then to reorganize them once carried, this work would be much less well adapted to this organization (or perhaps an additional computer would be required to do this). Next, the paths must not cross and all the workers must be able to get on and off without obstruction, or the system will not work at all!

In this case, another organization would consist in making a chain: one worker picks up a brick in the truck, passes it to a second worker who bends to pass it to a third on the ground, who passes it on ...
the \( N^{th} \) worker can place it on the heap. The throughput of this chain can be better than that of the preceding, but one must note that here all works move at the speed of the slowest worker: if one task (picking up the brick) is more complicated than the others, the \( N-1 \) workers will be under-employed. On the other hand, as each one achieves a very simple and very particular task (picking up, passing ..) they do not need a great deal of knowledge and can go quite quickly. Let us note again that this work is well adapted to this kind of organization, but would be less adapted if each worker had decisions to make (making a pile of big bricks and a pile of little ones). Besides, the bricks are treated in order, sequentially; if one wished to number them, this would be possible (one worker counts and marks) and much more complicated in the other organizations.

A third method, using freer and more intelligent workers, capable of accomplishing all tasks, would consist in letting them work independently each one getting on, taking a brick, getting off, .. so long as there are bricks. But here one encounters synchronization problems, if only one worker can place his brick on the heap at a time (look out for your fingers !), if only one can get on or off the truck at a time. And if there exists some wheelbarrows on the site, they can be used if available, .. in short, the workers must know how to wait for resources temporarily used by the others; there must be systems of waiting queues to avoid conflicts, ..: the entirety appears more decentralized but more difficult to manage - all the more so because at any given time no one can say what the others are occupied in doing. However, as each one knows how to do everything, one can conceive of working with a few workers or even with just one: a "reduced functioning" becomes possible.

These three types of organization are not mutually exclusive: any combination of the three can be imagined: supposing, for example, three workers make a chain to lower the bricks and the workers of \( N-3 \) transport them freely; or else, two chains can be organized in parallel; or teams could be created around a wheelbarrow, making a chain at the points of unloading and loading; the different teams working independently with respect to each other. It is essentially a question of level: on a certain level one can choose one kind of organization (pipe-line on the
interior of a processor) ; and on another level, one has other choices (parallel processors).

Thus in the complete organization of a data processing system, all of these techniques can be found:
on the hardware level, within the processors themselves,
on the architectural level, in the organization of processing units,
on the software level, by the way in which the processes, the tasks or the jobs are coordinated,
even on the exterior level, in the use made of different computers:
an application, composed of separate jobsteps, can be initiated on a computer and produce an output tape ; this being inputted to a second computer for the second step, while the first computer is used again elsewhere, etc ..
up to the various services of the society, which can be used in one way or another.

3. SOFTWARE ASPECTS OF MULTIPROCESSORS

3.1. How to use them ?

Multiprocessor architectures presented have one point in common : they appear much more complex than the usual mono-processors. They are composed of a greater number of units (processors, memories), joined to each other and to the exchange units, according to rules which though variables must be taken into account. It is easy to see how the control of these sets by an operating system will be more difficult than that of more classical systems - and yet the complexity of present systems is well known ! As far as memory management is concerned, and that of the Input/Output (I/O) operations, the same type of problem would seem to occur as with the classical O.S., complicated only by more complex rules of connection of the elements to each other.

On the other hand, where the processors are concerned, the problem is new ; it can be stated in these terms : how should work be distributed among all the processors ? First, one must find enough tasks to be executed simultaneously, each on a different processor, so that the rate
of employment of these "workers" will be satisfactory and the problem
will be commensurately more important and difficult as the number of
processors increases. In fact, in a system comprising a single processing
unit, three or four jobs are present in the memory and capable of being
carried out; but they are often stopped by I/O operations, and it can
even happen that the P.U. may be inactive ("wait state"). On a multipro-
cessor, the same organization would probably lead to a serious under-
employment of the hardware. In a system like ILLIAC IV, for example, the
processors are much better employed during the execution of vectorial or
matricial operations, but every program includes sequential sections
(data initialization, preparation of the results, conditional jumps, ...):
if in such program areas only one of the 256 PE is active, it is evident
that the occupation rate of the latter will be poor. It is imperative
therefore that we be able to detect, in the input job-stream, the maxi-
mum of tasks which can be done in parallel: this is what we mean by
"detection of parallelism"; and there exists, to achieve this, a comple-
te series of techniques which will be fully explained in chapter 3.2.

Next, it is necessary to be able to make use of this parallelism;
in other words, one must control the processors in such a way that during
the course of execution of a program containing parallelism, the proces-
sors may be quickly mobilized, the work distributed and the necessary
synchronizations provided. Here it is a question of the O.S. organization
which can call for the use of appropriate hardware. We will explain in
§ 3.3. the software architecture of MIP systems, showing how it allows
the use of the different parallelisms exhibited in § 3.2.

3.2. Parallelism exhibition

A certain number of techniques exist which can permit the exhibition
of parallelism; before examining them, it is important to note that
they are not exclusive: several of them can be combined, used at the
same time in order to bring out the greatest possible amount of parallel-
ism - in the same way as the three techniques allowing the combination
of processors (§ 2.) can be used together, but at different levels.
3.2.1. **Independent tasks or jobs**

Already, in multiprogramming systems, it was common practice to use several jobs together in order to make most efficient use of the central and I/O processors. As these jobs were completely independent and working on different data, it is perfectly simple to treat them in parallel; unfortunately the memory size limits their number rather quickly, and some of them must always be in waiting position for I/O.

At the interior of certain partitions (conversational sub-systems, time-sharing, . . .) the work can again be divided into several tasks capable of being carried out simultaneously: they are, for example, one task per user of the T.S., plus a certain number of service tasks: spool control, memory management, etc . . .

These diverse tasks are less independent than the jobs, in that they can have access to common data - therefore they must take all necessary precautions - and in that they may need to be mutually synchronized.

All of this already exists in the classical systems and will therefore be used to the maximum in systems destined to function on multiprocessors. This is what is being done on bi- or tri-processors which can now be found on the market: only this level of parallelism is utilized. But it would appear insufficient if a large number of processors is being planned. To find more parallelism it is necessary to go into detail concerning the execution of a program.

3.2.2. **Parallelism concepts and exhibition by the programmer**

In order to do this, the simplest thing is to ask the programmer writing down the program to describe what can be independently executed. Then a certain number of mechanisms permitting this will be put at his disposal.

**FORK / JOIN**

On the assembly level, the basic concept is that of FORK/JOIN, introduced by Conway : the following "primitive operations" are placed at the disposal of the programmer:
FORK A:
allows the initialization of a parallel task at address A, while the current work is continuing.

FORK A,J:
initiates a parallel job at address A, increments the counter situated at address J - or initializes it to N : FORK A,J,N.

JOIN J:
decrements the counter at J; if the latter becomes zero, execution is continued at address J+1; if not, processor is freed.

EVENTS

First of all this mechanism allows the classification of a part of the parallelism of a program; however, the only mean of synchronization between these different tasks is JOIN, which seems insufficient. Let us suppose, in fact, that once we have come to (1), we wish to make sure that branch A has indeed accomplished some work and reached (2) (for example, has filled a buffer with information) : we must be able to allow branches to wait on each other in some other way than by ending : whence the introduction of the "events", widely used in O.S. construction - in which for instance, A would be an I/O task. Therefore we introduce "primitives " : WAIT(event)
SIGNAL(event)
which permit this sort of diagram :
Those primitives are used in the construction of the O.S., but can also be placed at the disposition of the application programmer. Thus in PL/1 the latter would dispose of:

CALL(procedure,TASK option):
which has the same effect as FORK, initiating a parallel sub-
task; in addition, this subtask can be associated with an event.

RETURN:
same effect as JOIN, terminating the subtask and "signaling" the event.

EXIT:
allows the termination of a task and cancellation of all the subtasks which are still in execution.

WAIT:
permits one to await the end of a subtask associated with an event.

The exclusion problem:

Suppose that two programs started this way in parallel both need access to a common data, in order to modify it: if each one reads it, then modifies it, a false result may be obtained. If, for example, both wish to increment a variable A, the following sequence may result:

Program 1:
Load A
Acc + Acc + 1
Store A

Program 2:
Load A
Acc + Acc + 1
Store A

time
As a result, $A$ will be increased by 1 and not by 2!

The example given here is simple; in reality a user may want all access to certain variables to be forbidden to other programs during the entire course of a "critical section": this is the case in a system of airplane reservation: between the moment when one checks to see if there remains a seat, and the moment when one takes it, no one must have grabbed it!

Therefore something permitting us to express this has been introduced in the O.S. and then in the high level languages allowing parallelism - An effective way to do this will be studied in § 3.3. - They are, for instance, the "critical sections" of Hansen:

\begin{verbatim}
Critical section A do;
  A := A + 1;
end;
\end{verbatim}

The achievement of this guaranties that whatever be the number of users which are trying to increase $A$, at each instant there will be only one at the inside of such a critical section.

Such exclusions can easily be effected with the assistance of **semaphores**: let us remember that a semaphore is an object with $N$ possible states (a counter $\text{CPTR}$) to which can apply the operations:

\begin{verbatim}
P(semaphore) : if $\text{CPTR} > 0$ then $\text{CPTR} \leftarrow \text{CPTR}-1$
else "wait" (and re-try P)

V(semaphore) : $\text{CPTR} \leftarrow \text{CPTR}+1$
\end{verbatim}

Then it is sufficient to associate one semaphore named "Sem-A" to this section and to replace the beginning of the critical section by $P(\text{Sem-A})$ and the end by $V(\text{Sem-A})$; however, these primitives $P$ and $V$ are more flexible, which allows the achievement of the event mechanisms: $\text{WAIT} = P$, $\text{SIGNAL} = V$ - or of other even more complex synchronizations.

Thus in ALGOL 68, one can write "collateral clauses": by replacing the ";" of ALGOL by ",", one means that the instructions or blocks so separated (which can obviously be procedure calls) can take place in parallel; and the use of semaphores allows the realization of nearly all types of synchronization needed.
Use by the programmer:

In order to use these multiprocessor architectures, the application programmer is of course advised to exhibit the parallelism of his program, with the means which are given to him by the language he is using. In this way, he will bring out a certain amount of parallelism.

But on the system level, to achieve all the kernel routines and the other components of the O.S., as well as the subroutine libraries, a maximum parallelism will be used. Since a great part of the work time is passed in these modules, a good deal of independent tasks can be set free. But to do this, a system programming language which makes the parallelism clear more easily than in terms of FORK and JOIN, is desirable.

3.2.3. Specialized languages

The high level languages presented merely took up again in a more or less improved form, the concepts which had been devised during the construction of multiprogrammed operating systems, in terms of parallel tasks. Thus they permit the exhibition of a certain parallelism, but are not conceived for that.

On the other hand, with the appearance of array-processors like ILLIAC IV, the development of specific languages was aimed at, taking into account the particularities of the hardware: this is TRANQUIL, a now abandoned ALGOL-like language.

In the latter, it was possible to declare SKewed or STRAIGHT arrays according to memory organization (remember § 2.1.2.). Subarrays could be defined on these arrays and operations on these subarrays were allowed:

\[ X \leftarrow A[1] + B[7]. \]

in which X is a vector, A and B two-dimension arrays.

Parallel execution of sequences S1, S2, .. can be initiated by:

SIM BEGIN (S1; S2; ...; Sn) END

In addition sequential and parallel loops exist, possibly with several indexes; the laws of composition for the indexes can be one for one, or according to the cartesian product:
Examples:

\[ I \leftarrow (1,2,3,\ldots,10) \]
\[ J \leftarrow (5,10,15,\ldots,50) \]
\[ \text{FOR (i,j) SEQ(I,J) DO} \]
\[ A(i) \leftarrow B(i) + C(j) \]

has the effect:

\[ A(1) \leftarrow B(1) + C(5) \]
\[ A(2) \leftarrow B(2) + C(10) \]
\[ \ldots \quad \ldots \quad \ldots \]

On the contrary

\[ \text{FOR (i,j) SEQ(I\times J) DO} \]
\[ A(i) \leftarrow B(i) + C(j) \]

will have the effect:

\[ A(1) \leftarrow B(1) + C(5) \]
\[ A(1) \leftarrow B(1) + C(10) \]
\[ A(1) \leftarrow B(1) + C(15) \]
\[ \ldots \quad \ldots \quad \ldots \]
\[ A(2) \leftarrow A(2) + C(5) \]
\[ \ldots \quad \ldots \quad \ldots \]

and parallel loops, all executions of which can be performed simultaneously, are written in the same manner:

\[ \text{FOR i SIM I DO S} \]
\[ \text{FOR (i,j) SIM(I\times J) DO} \ldots \]

In addition, condition evaluations can be performed about entire vectors: this would be written:

\[ \text{IF ANY } X \leq Y \text{ THEN } S \text{ ELSE } \ldots \]

or

\[ \text{IF ALL } Z \geq 0 \text{ THEN } S_1 \text{ ELSE } S_2 \]

in which \( X, Y \) and \( Z \) are vectors.

A routine \texttt{SUM} permits the addition of all the elements of a vector.

Such a language, though it reflects some particularities of the hardware, is as poor for using and reading as any other high level language.
3.2.4. Detection of parallelism at compile time

Another mean for bringing parallelism to light and preventing the programmer from having to do this work, consists in giving the compiler the responsibility of detecting what can be done independently and of detailing this in the object code.

Independent instructions:

Consider two statements, S1 and S2, of a high level language program; if Ii and Oi represent respectively the inputs (read data) and the outputs (modified data) of Si, a necessary and sufficient condition so that S1 and S2 will be independent - that is executable in any order or simultaneously - is:

\[ I1 \cap O2 = \emptyset \]
\[ I2 \cap O1 = \emptyset \]
\[ O1 \cap O2 = \emptyset \]

The detection algorithms for independent instructions proceeds then in this way: given that an algorithm, that is a sequence of statements, and the rules of precedence caused by jumps, the analysis of inputs and outputs of each statement permits the determination of what can be done in parallel.

Unfortunately, the custom the programmers have of re-using the same variables to accomplish several different tasks, may cause the above method to fail; for example, the four statements:

\[ A \leftarrow B + C \]
\[ D \leftarrow A \times E \]
\[ A \leftarrow F \times G \]
\[ H \leftarrow A + 1 \]

will have to be executed in sequence, whereas by re-writing them as:

\[ A \leftarrow B + C \]
\[ D \leftarrow A \times E \]
\[ A1 \leftarrow F \times G \]
\[ H \leftarrow A1 + 1 \]

we will know how to execute the two first and the two last in parallel: this is the "memory deallocation" problem. It is possible to detect such a re-use of a same variable, but the algorithms for that are of course quite costly!
Parallel evaluation of arithmetic expressions:

Complex arithmetic expressions can easily give rise to parallel evaluation: it suffices to examine their decomposition in binary tree (or the prefixed notation) to have a first idea of what can be done simultaneously.

Given that we calculate \( E + A + B + C/D \) which can be represented by:

```
          +
         /  
        +   /
       A   B
```

Obviously, \( A + B \) and \( C/D \) can be calculated at the same time and then the last addition can take place: \( E \) is evaluated in two steps (rather than three). And the construction of this binary tree is in fact done by any compiler.

In some cases, the commutativity of the arithmetical operations will need to be brought into play in order to reduce the height of the tree and so increase the parallelism; therefore we will transform

\[
A + C/D + B
\]

(three steps)

```
          +
         /  
        +   /
       A   B
```


to obtain the tree shown above.

We can also use the distributivity of these operations to reduce tree height; so we can transform

\[
A \times (B \times C \times D + E)
\]

into

\[
(A \times B) \times (C \times D) + A \times E
\]

which reduces from four to three the height of the tree.

Unhappily, the cost of such algorithms using distributivity is high at compile time and often serves no purpose, since one cannot be sure in advance of reducing the height of the tree in this way.
Loops:

Some loops are of such a nature that any execution of the contained instructions can take place in parallel. For example, this is the case when doing:

DO I = 1 TO 10;
    A(I) = B(I+1);
END;

So ten parallel tasks are distinguishable and this by examining only the inputs and outputs of the set of instructions contained in the loop.

The deallocation memory problem can also occur, which one resolves as before by allocating as many variables as necessary.

More complex is the case of linear recurrences; it can be observed that

DO I = 1 TO N;
    X(I) = A(I) + X(I-1)
END;

can in fact be written

\[ X(I) = \sum_{J=1}^{I} A(J) \]

and calculated in a parallel way: so \(X(1)\) to \(X(8)\) could be calculated in three steps (see fig. 6). But obviously, such transformations are not easy to detect and achieve.

**Amount of parallelism detected in this way:**

Rather extensive studies \(^10\) have shown that by using these techniques and by applying fairly sophisticated algorithms, up to 16 processors can be employed for one FORTRAN program - and this for a speed-up ratio of approximately 4. These studies were made on benchmarks of small programs, but it is probable that for larger ones the results would be even better.

Most studies in this field have involved FORTRAN: firstly because high power computers are essentially destined to effect scientific calculations (ILLIAC IV), secondly because FORTRAN generally works on arrays resident in core memory, and asks for little I/O operations. However, commercial applications were limited by these I/O operations: a pay-roll
application, working with two input tapes and one output tape, will just be able to perform one sheet at a time - because of the sequential nature of the peripheral equipment. But the program itself is perhaps very "parallel" and the basic loop can be executed independently for each employee. In recent data banks systems, where all files are located on direct access devices, one can hope to find a very great amount of parallelism. The appearance of such data banks systems, at the same time as the lowering cost of the mini-processor, will perhaps lead to the much wider use of medium-power multiprocessing systems.

With respect to the exhibition of parallelism by the programmer, this type of detection is obviously not advantageous: it is ridiculous to ask someone to code sequentially an algorithm which may be of a parallel nature, then to try to find in the sequential expression what can be set in parallel. The only justification for this detection is the commercial advantage of being able to reuse all the existing programs by merely recompiling them.

3.2.5. Detection of parallelism at execution time

Methods for detection of parallelism at execution time consist, starting from a single running instruction flow, in detecting what can be done independently and in doing it simultaneously on several processors. Such methods can only be of hardware type, given the time necessary for detection. They are then transparent to the programmer and to the compiler who generate a series of sequential instructions; but once again it is a question of detection of parallelism and we mention it under this perspective.

The most widespread method is the pipe-line one: one begins execution of the instructions in parallel (slightly staggered); if one observes a conflict, one can conclude that the instructions are at fault, not being completely independent of the preceding ones; so they are begun again.

Another method, called "Leapfrog", consists in having two processors execute one instruction stream, one processor effecting instructions i, i+2, i+4, ..., and the other, i+1, i+3, i+5, ... The two processors share a control memory (to which they have access during the first part of the
instruction) and a local memory containing the registers (second part). Conflicts are avoided by staggering the instruction:

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Inst.1</th>
<th>Inst.3</th>
<th>Inst.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor 2</td>
<td>Inst.2</td>
<td>Inst.4</td>
<td>Inst.6</td>
</tr>
<tr>
<td>first part : fetch</td>
<td></td>
<td>second part : operation</td>
<td></td>
</tr>
<tr>
<td>and decode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Of course, there can also be dependent relations between instructions which are detected by hardware and which slow down the general functioning; moreover, execution time of an instruction is not constant and often one processor may wait on the other.

The system proposed by Tjaden and Flynn \(^{15}\) is more complex: N instructions (about ten) are introduced in a "pre-decoding stack" which analyses the dependences between them - in the same way as compiler, by studying inputs and outputs - It then detects \( M < N \) instructions capable of being done immediately and in parallel, and sends them on to independent processors. So that this analysis be rapid, the operation-code was modified in order to easily bring out relations which exist between instructions at pre-decode time ("dependency field" = inputs, "effect field" = outputs). Each processor assumes the complete decoding of the instructions he has to effect.

These diverse techniques lead to relatively weak gains in speed: a respective increase of 1,20 (Leapfrog) or of 1,50 (pre-decoding) in throughput! This is undoubtedly why such architectures are not very widespread.

3.3. Utilization of parallelism in MIPs

Once the parallelism has been detected and exhibited in the code which the multiprocessor will execute, it becomes necessary to put it into effect, or in other words, to distribute the independent tasks among the different processors and to synchronize them. To do so several techniques will have to be used; the first consists in taking up what is being done in classical multiprogramming systems.
Processes and processor:

Let us keep in mind that in this kind of system, several tasks can be accomplished in "pseudo-simultaneity": the CPU works for a certain time ("time slice") for a user, processing his program; upon reaching the end of the time slice, the CPU puts the registers and PSW away in a control block associated with this execution, and searches another task: it fetches the next control block, loads the registers and PSW, and continues the execution of this second user; and so on... At the end of a certain time, it begins to work for the first user again, taking it up in the same condition in which it had been left.

What we call processes are the "executions of a program for a user account", which are representable by these control blocks and which are activated in their turn by the CPU. The CPU multiplexing allows them all to be achieved in pseudo-simultaneity.

If a process is waiting for the end of an I/O operation, it should be removed from the list of processes which the CPU must activate each in turn; but the I/O-complete interrupt will put it back in place. In the same way, if a process is waiting for a logical resource which is used by another (a data set, or a critical section, or...), or for an event, this process is removed from that list. Only the freeing of this resource (V(Sema)), or the activation of this event (SIGNAL(event)) will replace it in the list of candidate processes for the CPU.

The mechanism which involves in allocating the CPU is called a "scheduler" or "dispatcher"; it can take into account the different processes priorities.

Case of the multiprocessor - Exclusion problem:

A MIP can work in the same way - the only small difference being that several processors execute processes at each instant - when one has finished (Proc1 has finished P2), it comes to see the state of the list, and decides to execute the next process (P4) (repositions the index):
List of processes:
1 = in effect
0 = waiting for a processor

Processors:

It is therefore as simple a way to activate N processes with P processors as with a single one. But to do this at least the "dispatcher" (consultation and positioning of the index) must run in a critical section: one can imagine a case where the two processors take the same process. If it happens and supposing the two processors to have access to memory without warning each other, a mechanism must exist to accomplish this exclusion. If we dispose of exclusion only on the memory access (one access at a time) and can use only the load and store operations on the latter, then exclusion will be possible but very difficult to achieve.

This is why we find a special instruction on multiprocessors: TEST & SET whose effect is to:

. read a byte in the central memory,
. position the condition code, according to the value of this byte,
. store zero in this byte,

in a single memory access, that is with the guarantee that no other processor is allowed to read or store this byte between this read and this store. Thus, a processor can enter a critical section A by:

x : TEST & SET (byte-A)
if condition-code = 0 then goto x

and go out of the critical section by:

RESET (byte-A)
in other words by storing 1 in the byte associated with A.

FORK implementation:

The execution of the FORK primitive operations by a process is therefore equivalent to the creation of a new process, initiated to the content (registers and PSW) of the processor doing the FORK and effecting the indicated address.
The system will take this process into account when it is its turn: the two "branches" issued by the FORK will take place either in real simultaneity, or in pseudo-simultaneity: that is of minor importance. What is necessary is that the processors are constantly utilized so that work will be done at a maximum throughput.

On the contrary, the end of such a branch (JOIN) sometimes precipitates the continuance of a calculation and sometimes the end of the running process - releasing the control block -

These mechanisms (initialization of a control block, and restoration) are accomplished by software and represent a certain quantity of work. Therefore it can only be used if the branch executed in parallel is sufficiently long, so that the overhead is negligible.

Rapid mechanisms:

This is not the case, however, when one wishes to execute several instructions or even the different operations of an arithmetical expression in parallel. The most rapid mechanisms must be imagined, putting into play liaisons between processors and rapid shared local memories, where one can store the parameters 'registers and address to be carried into effect'. This is true for the MC6 project of CII where rapid execution ("mini-fork" and "mini-join") sets of primitives are being achieved. Other projects are making use of more appropriate hardware 11).

Conclusion: an economic point of view

It is probable that this sort of architecture, composed of a great number of processors, will be more and more widespread. One of the most original characteristics is the decreasing cost of computation work, whereas the cost of memory or I/O operations varies little. Moreover, if a very large number of processors exists (say, about 100), it is probable that they will be poorly employed and some of them will always be inactive.

The average cost of instruction execution will therefore be smaller than at the present time, with respect to other operations (memory or I/O) which represent bottlenecks in such systems.
For an efficient use of all these resources it will be necessary to translate the price of the components into terms of cost and the invoice will need to include this. CPU time will no longer be the essential element in the cost of a job, but rather the residence-time within the memory. Thus the users who have detailed their parallelism, or who wanted to compile their programs with an appropriate compiler (more expensive to use!), will have the advantage: apparent execution time for their programs will be shorter. On the other hand, a user who forces his work to run off on a single processor will be penalized.

Execution time is no longer the costly aspect of a system and this should lead to the accomplishment of too much work (instruction processing) if the result can be obtained more quickly. We will prefer starting two different algorithms to solve the same problem, for example to inverse a matrix: the one which performs more quickly will terminate first and will cancel the other - the result being therefore available more quickly than if one had used a single method! A profound change in current programmer practice will undoubtedly occur, as far as these practices are a consequence of the relative costs of the components of the 60's computers.

REFERENCES

1) J.L. BAER, "A survey of some theoretical aspects of multiprocessing", Computing Surveys, Vol 5, nb 1, pp. 31-80
2) M. CONWAY, "A multiprocessor system design", Proceedings AFIPS FJCC, 1963
3) E.W. DAVIS, "Concurrent processing of conditional jump trees", COMPON 1972
4) E.W. DIJKSTRA, "Solution of a problem in concurrent programming control", C.ACM 8,9, pp. 569-570
6) S. EVEN, "Parallelism in tape sorting", C.ACM, 17,4, April 1974
8) D.J. KUCK, "ILLIAC IV software and applications programming", IEEE TC C 21, December 1972
9) D.J. KUCK and Y. MURAOKA, "Fast computers from slow parts", COMPCON 72
13) J.L. ROSENFELD, "A case study in programming for parallel processors" C.ACM, 12,12, December 1969
14) J.L. ROSENFELD and R.D. VILLANI, "Multimicroprocessing, an approach to multiprocessing at the level of very small tasks", IEEE TC, C 22,2 February 1973
Fig. 1. ILLIAC IV general architecture
Fig. 2. Skewed matrix storage (a(n,n) matrix)
Fig. 3. Functioning diagram of a five-steps ($\Phi_1, \ldots, \Phi_5$) pipe-line

Fig. 4. Organization of a five-station pipe-line
n parallel programs

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst 1,1</td>
<td>inst 1,2</td>
<td>inst 1,n</td>
</tr>
<tr>
<td>inst 2,1</td>
<td>inst 2,2</td>
<td>inst 2,n</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.......</td>
</tr>
<tr>
<td>inst i,1</td>
<td>inst i,2</td>
<td>inst i,n</td>
</tr>
</tbody>
</table>

n processors (real or virtual)

**Realisation** : (n = 3)

Fig. 5. Utilisation of a "pipe-line" to execute parallel sequences of instructions
Fig. 6. One way to perform:

\[ \text{DO } i = 1 \text{ TO } 10 \quad X(i) = A_i + X(i-1) \]

in three steps, with eight processors P1 ... P8
SPECIAL PURPOSE PROCESSORS

C. Verkerk
CERN, Geneva, Switzerland

INTRODUCTION

The subject of these lectures will be special purpose processors implemented in hardware. The subject matter is difficult to treat in a general way since due to their specialisation it is only practical to present specific examples of processors. At best one can hope to distinguish some classes of processors, where the classes correspond in general to the application area, rather than to the architecture or design philosophy. Different pattern recognition and picture processing machines have points in common and Fast Fourier Transform processors bear a resemblance to each other. Indeed when one takes a closer look at the different special purpose machines proposed or built, one is struck by the very close relation between the hardware and the algorithm the machine has been built to execute. This could possibly be transposed at least for the purpose of the present lectures into a definition: a special purpose processor is a piece of hardware designed to fit as closely as possible the algorithm it is built to execute.

The point is that the hardware has been designed for the algorithm \[1\] and that no programming has been done to implement it on a machine which can be reprogrammed. A close relation exists between the algorithm and its implementation in special purpose hardware.

The incapacity of being programmed in the usual sense of the word does not necessarily mean that special hardware is completely rigid and unadaptable. Of course, it would be unthinkable to adapt a Fast Fourier Transform processor for use in, say, a numerical control application. But the FFT processor can be used for many tasks (transform, convolution, correlation, digital filtering). Often the change from one task to another can be made at the flick of a switch. The definition would exclude special purpose systems which are based on a single microprocessor (computer on a chip, or rather a handful of chips) or a minicomputer, microprogrammable or not. Neither are larger special systems, implemented around a general purpose machine a subject for these lectures. Parts of such a system could well fit our definition (e.g. I/O channels, or the special processor for filtering slice scan data in the Erasme system \[58\]).

In agreement with the rather restrictive definition of special processors given above, the present lectures will try first to develop some general points on the applications and motivations of hardware processors. More specific points will be discussed in the examples which will be covered in some detail: Fast Fourier Transform Processors and hardware for track recognition in high-energy physics experiments. The emphasis will be on processors for wire chamber data, but a rapid survey will be made of special hardware for filtering of digitized coordinates from bubble chamber film.
Why and When Special Purpose Processors?

Let us try to answer the question "why special purpose processors?". The answer is already contained to some extent in what was said before: for some problems programming a general purpose computer is not necessarily the optimum solution. Special hardware can be more cost-effective than the running of a programme on a general purpose machine or - if it is not - it can overcome a real-time constraint. The FFT hardware provides an example for both motivations: it was soon realized that FFT processor would be very cost-effective [2] and that they would allow real-time analysis of Doppler Radar returns [3], [4] or of human speech [5]. Picture processing (image restoration, image enhancement, difference detection) and pattern recognition in photographic images also provide strong motivations for studying special hardware solutions. The reason, of course, is that the large number of picture cells to be handled, combined with grey-level and colour information make the processing task formidable and time-consuming on a sequential machine. The tendency in this field is to think in terms of highly parallel structures (cellular automata). References [6] to [10], far from being exhaustive give some examples. An exception to this cellular approach seems to be the Control Data picture processing machine [1].

Both motivations also exist in high-energy physics. Many analysis programs spend a large fraction of time in a few inner loops. The calculations in these loops are often very simple and it can be cost-effective to perform them outside the large computer. The data presented for further analysis have then already been filtered elsewhere. The real-time constraint can be present when a selection of events has to be made at an early stage (i.e. before recording on magnetic tape, or before triggering the flash lights of a rapid cycling bubble chamber [11]).

Some of the answers to the next question: "what kind of problems are good candidates for implementation in special hardware?" have become apparent already.

i) Obviously the first condition is that there be a sufficiently continuous stream of data to analyse, in order to give a reasonable life-span to the processor. In other words: no special hardware for a one-off problem.

ii) Secondly the algorithm should be to a large extent iterative or repetitive, which implies that some kind of a loop-structure should be present. Nested loops are ideal, because a large part of the overheads normally present in the loop control can be eliminated.

iii) The simpler the arithmetic or logical operations to be performed repetitively, the more suitable the algorithm is for hardware implementation.

iv) The required precision should be rather small, so that 12 or 16-bit fixed point arithmetic may be used. Floating point operations on longer words are not excluded, but the cost of the hardware would increase sharply.
v) A simple control structure will result in simple hardware. A straightforward algorithm, where no special cases have to be considered, would be ideal. This point should not be exaggerated though. We will see later that IF-statements, conditional jumps and the like can be implemented, often more easily and more efficiently than in a program. The question of control is also closely linked to the problem of flexibility, to which we will come back.

vi) Finally, a last asset for hardware implementation of an algorithm is that it consists of independent parts. When different calculations are independent of each other, they can be performed in parallel. Another possibility is that calculations can be performed in a pipeline. This is the case when later steps in the algorithm do not make use of hardware necessary for the execution of earlier steps. A continuous stream of "events" can then be maintained, the different events in the pipeline residing in different sections of hardware and being in different stages of progress.

Influence of Technology

It is commonplace to say that the extremely fast development of the integrated circuit technology over the last ten years has brought about a revolution in digital design. I apologise for repeating it, but it is worthwhile to point out that the present variety, availability and price of integrated circuits has brought it within the possibilities of individuals in a laboratory to do things they would not have dreamt of five years ago. This is particularly true since MSI and LSI circuits are readily available. This has really brought it within the reach of many an electronic designer to build special purpose systems. Until recently his designs were often limited to data acquisition and interfacing problems, without arithmetic capabilities. To open the way to the design of special purpose processors, it is sufficient to become aware of the possibilities offered by including arithmetic. By studying carefully the algorithm he is to implement he will be able to find elegant solutions which speed up execution or reduce the amount of hardware. The chances are that the average designer will find original solutions, if he works towards the closest fit between the hardware and the algorithm. If not, and if he remembers too much von Neumann, the result will probably be another item on the already long list of microprogrammed or microprogrammable minicomputers.

It is probably worth mentioning that the full impact of the integrated circuit revolution on the computer industry is still not felt. There are some signs of it coming \[2\]. Significant are the laments that computer architects are not sufficiently aware of the present hardware possibilities. These laments are accompanied by recommendations to stimulate hardware research in American Universities \[3\]. This prospect of entrusting more and more system tasks (processes) to hardware processors in the framework of general purpose computers constitutes in fact the bridge between the present lectures and the other lectures on computer architecture. Conceptually, very little would change for the systems architect if, for instance, a loader would be implemented in hardware or in software. The position taken in the present lectures contrasts however with the general trend in systems programming on one point: binding. Whereas the systems programmer delays binding to the ultimate moment, this is not the case - yet - in special purpose hardware. There every datum is fixed to its place right from the beginning.
This early binding and the fact that the hardware is made to fit the algorithm as a glove relieves us from many of the problems encountered in general purpose systems: e.g. fast operand fetching, adoption of programs to multiprocessor systems, optimum use of arithmetic and other resources. Early binding should not be confused with lack of flexibility! We will come back on the presumed inflexibility of hardware in the examples we will treat in detail.

**FAST FOURIER TRANSFORM PROCESSORS**

A vast amount of literature exists on the FFT (see ref. [2] for an extensive list of earlier references). We will therefore not dwell upon details, limitations, pitfalls, etc., but limit ourselves to one specific example of a hardware implementation [3] which illustrates best the points we want to make.

**The Algorithm**

The discrete version of the Fourier transform, applicable to time samples of a continuous function \( x(t) \) is:

\[
X(j) = \frac{1}{N} \sum_{j=0}^{N-1} x(k)e^{-i \cdot 2\pi jk/N}
\]  

(1)

with its inverse:

\[
x(k) = \sum_{j=0}^{N-1} X(j)e^{i \cdot 2\pi jk/N}
\]  

(2)

both for \( j = 0,1,\ldots, N-1 \) and \( k = 0,1,\ldots,N-1 \)

Both the spectrum \( X(j) \) and the time series \( x(k) \) are, in general, complex series. Replacing \( e^{2\pi i/N} \) by \( W \), we can write the essential part of (1) or (2) as:

\[
X(j) = \sum_{k=0}^{N-1} A(k)W^{jk}
\]  

(3)

For the sake of clarity we will limit ourselves to the case \( N=8 \).

We will then write \( j \) and \( k \) as binary numbers: \( j=j_2 \cdot 4 + j_1 \cdot 2 + j_0 \) and \( k=k_2 \cdot 4 + k_1 \cdot 2 + k_0 \).

Eq. (3) then becomes:

\[
X(j_2,j_1,j_0) = \sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} A(k_2,k_1,k_0)W^{(j_2 \cdot 4 + j_1 \cdot 2 + j_0)(k_2 \cdot 4 + k_1 \cdot 2 + k_0)}
\]

Splitting up the exponent into three partial products and remembering that \( W^8=1 \) we obtain:

\[
X(j_2,j_1,j_0) = \sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} A(k_2,k_1,k_0)W^{j_0 \cdot 2 \cdot 4 (j_1 \cdot 2 + j_0)k_1 \cdot 2 W(j_2 \cdot 4 + j_1 \cdot 2 + j_0)k_0}
\]
This can be calculated in three steps:

\[
A_1(j_0, k_1, k_0) = \sum_{k_2=0}^{1} A(k_2, k_1, k_0) W_{j_0}^{k_2^4}
\]

(4)

\[
A_2(j_0, j_1, k_0) = \sum_{k_1=0}^{1} A_1(j_0, k_1, k_0) W_{j_1^2+j_0}^{k_1^2}
\]

(5)

\[
A_3(j_0, j_1, j_2) = \sum_{k_0=0}^{1} A_2(j_0, j_1, k_0) W_{j_2^4+j_1^2+j_0}^{k_0}
\]

(6)

with final transposition:

\[
X(j_2, j_1, j_0) = A_3(j_0, j_1, j_2)
\]

(7)

Figure 1 - FFT Data Flow for N=8

Note that each step calculates 8 new terms from the old ones. Only the results of step \(m\) are needed in step \((m+1)\) and the older results can be discarded. The importance of the FFT lies in the fact that the \(N^2\) complex operations (one multiplication and one addition) to calculate (3) are replaced by \(N \log_2 N\) operations. By noting that \(W^k = -W^{(k+N)/2}\) we see that we have to perform only \(\frac{N^2}{2} \log_2 N\) times the following calculations:

\[
x_{out} = x_{in} + y_{in} W^Z
\]

(8)

\[
y_{out} = x_{in} - y_{in} W^Z
\]

(9)

(or, in the previous notation, for instance:

\[
A_2(5) = A_1(5) + A_1(7)W^2
\]

\[
A_2(7) = A_1(5) - A_1(7)W^2
\]

The process, still for \(N=8\) is illustrated in Figure 1 (from [4]). Obviously the general algorithm [5] is not limited to \(N=8\), nor even to \(N=2^m\), although most hardware implementations have the last restriction. For earlier hardware implementations see Bergland [6].
Hardware Implementation

The heart of a FFT processor is the arithmetic module to perform the complex calculations (8) and (9). In terms of real numbers, this "butterfly" can be implemented as shown in Figure 2. It is important to realize that this implementation constitutes a big advantage over the general purpose approach where – generally speaking – only a single arithmetic operation would be in progress at any instant in time. Note that a further speed-up could be obtained by organizing the butterfly in a pipeline: a new multiplication could start as soon as the past one is ready and before the results of the additions and subtractions are known. This would lead, however, to complications in addressing the data. Other, cheaper designs of the butterfly are possible, requiring less multipliers and adders \[ \text{[13]} \], but they are slower, due to their sequential operation.

Bergland \[ \text{[14]} \] distinguishes four organisations of the FFT processor:

i) Sequential. A single arithmetic module (butterfly) performs all operations sequentially in a total time \( T \cdot \frac{N}{2} \log_2 N \). [5], [8], [19], [24].

ii) Cascade. There is one butterfly for each iteration. This requires \( \log_2 N \) butterflies, each performing \( N/2 \) operations sequentially. It will still take \( T \cdot \frac{1}{2} \log_2 N \) before the spectrum of a data set is available, but the throughput is increased by a factor \( \log_2 N \), since \( \log_2 N \) data sets can be simultaneously in the pipeline \[ \text{[2]}, \text{[3]} \].

iii) Parallel iterative. Here the parallelism is in the horizontal direction in Figure 1. So there are \( N/2 \) processing elements and the execution time is \( T \log_2 N \). [21], [22].

iv) Array. Parallelism is extended in both directions. \( \frac{1}{2} N \log_2 N \) butterflies are needed, the throughput is one data set every \( T \) seconds. [23].

It is clear that the last two organisations are rather unrealistic for data sets of 1024 or more samples. In what follows we will therefore give as an example a cascade (or pipeline) implementation, which is the one built by Goginsky and Works \[ \text{[3]} \].
Figure 3 - Flow Diagram of FTT

Figure 4 - Module m of Pipeline FFT

Figure 5 - Block Diagram of FFT Processor
Let us first have a look at Figure 3, which is another drawing of the flow of data in the FFT algorithm. We can then note the following:

i) each iteration needs only the results of the preceding stage.

ii) the first iteration uses data samples separated by a distance \( N/2 \), the second by \( N/4 \), etc.

iii) the stages are independent in the sense that one stage can be working on data from one source, while another stage treats data from another source (storage must be provided between stages).

iv) the rotation vector required at each stage has the same periodicity as the displacement between data samples.

These considerations lead to the implementation of stage \( m \) which is shown in Figure 4. The delay line (actually a shift register) can contain \( 2^m \) samples (total delay time \( \Delta = 2^m \)). \( \delta \) when \( \delta = \) sampling time interval. The arithmetic unit performs calculations (8) and (9). The switches are controlled as follows: first \( 2^m \) data samples are stored in the delay line and when it is full a rotation vector \( W^j \) is obtained and the switches are thrown. So the arithmetic module now starts working, receiving one input \( (x_m) \) per sampling interval \( \delta \) from the delay line, the other input \( (y_m) \) directly from the data source. One output \( (x_{m+1}) \) is passed on immediately to the next stage, to be stored in the delay line there (the latter can contain only \( 2^{m-1} \) samples). The other output is fed back into the local delay line, so that it will become available to the next stage after \( 2^m \) sampling intervals. This is just what is needed (see Figure 3). In stage \( m \) the delay line is thus alternatively filled with fresh data and with results, in blocks of \( 2^m \). The next stage, working on blocks of \( 2^{m-1} \) data needs to throw switches twice as often as stage \( m \). The control of the switches can therefore be performed with the help of a binary counter! The total organisation then becomes as in Figure 5.

The rotation vectors are stored in a (read-only) memory in the order required for the last iteration. The binary counter provides the addresses for this memory. The stages upstream, which need a lower periodicity for the rotation vectors, strobe the correct value into an internal register at the moment its switches are thrown.

There is still another trick present in this implementation. As can be seen from Figure 3, when two independent data streams are interleaved (even and odd samples belong to different data sets) then the spectral components of both sets are available at the intermediate output of stage 1, interleaved also. In general, for \( 2^k \) interleaved sets, the interleaved spectral components (in the same channel order) are available at the output of module \( k \). So one can easily choose between a single stream of 1024 samples, or 4 streams of 256 samples each, etc.

This processor has in fact been built and it processes eight range channels of a Doppler radar, taking 512 complex samples per channel. The throughput rate achieved is 128K samples per second. The processor uses 12-bit fixed point arithmetic.

As with most other processors, the spectral components within a channel are produced in scrambled (bit-reversed) order, which is inherent to
the Cooley-Tukey algorithm (see Figure 3 and equation 7).

This example illustrates very well the point made before: by looking closely at the algorithm an elegant and fast processor has been produced.

SPECIAL PROCESSORS FOR HIGH-ENERGY PHYSICS

Bubble Chambers

Apart from the one already mentioned [7], other proposals [25 - 35] have been made to perform track-element search by special hardware and a number of projects [27 - 30 - 32 - 33] have been launched in the past to replace by hardware at least a major part of the track-following programs for data from flying spot digitizers. Most of these hardware approaches (LER, [25 - 26], BRUSH, [27 - 29], PANGLOSS, [34]) are based on histogramming technique. A

Figure 6 - Pattern for Line Element Search

histogramming technique which does not use arithmetic, but only logic. The hits of the flying spot on track images are not represented numerically as the value of the coordinate, across the film, but rather as bits in a shift register. The bit pattern in the shift register is a representation of the analogue signal produced by the photo-detector. A number of scans is represented in an array of registers. The shift register array (which is shorter than corresponds to a complete scan-line, scan-lines being shifted through it) contains pre-wired
patterns, approximating straight-line segments under different angles (see Figure 6). At every shift a hit count is established for each direction. A maximum must be sought in successive hit counts (always for each of the fixed directions) and a decision taken on the possible presence or absence of a line-element. The shift register array contains only a limited number of scan-lines (slice). Next the line-elements found in successive slices must be linked into tracks, which is another non-trivial problem. BRUSH [28] has in fact been implemented, and is undergoing extensive tests.

Coccinelle [32 - 33] followed another approach, purely analogue. A very strict synchronisation between line sweeps is imposed. A line element under a given angle is then characterized by the recurrence of a fixed time delay between successive signals. A set of delay lines is used to detect recurrent pulses. Output pulses are fed back into the same delay line, after being added to the input signal present at this particular instant in time. Pulses separated by the delay time build up to large amplitudes, dominating a sea of small pulses.

SATR [30 - 31] adapted a three-dimensional approach. It needs precise knowledge of the fiducial positions before processing can start. Every hit on the four stereo-views is transformed into a light ray in space. Small regions of space are then searched for close crossings of light rays. A track element detected is followed in three dimensions.

It is not known to the author if SATR reached a full scale implementation.

A less ambitious approach was taken at the Zeeman Laboratory in Amsterdam, where special hardware was built to follow beam tracks only [35]. A histogramming technique is followed, but using the numerical coordinate values. The hardware is thus a close replica of the kernel of a track-following program. Moreover, approximate position, incident angle and curvature are known for beam tracks. This approach seems to be successful and capable of following beam-tracks and detecting their disappearance at an interaction vertex. Finally, one should mention a special purpose processor under development for ERAMSE at CERN [36]. Here the aim is to speed up the analysis, decentralising still further the different tasks [58]. The analysis of a slice scan to find the parameters of a track segment will thus be done in specialised hardware, which has taken the form of a micro-programmed processor with a specialised instruction set.

**Wire Chambers**

For the treatment of wire chamber data a number of special processors have been built or proposed. The first to be mentioned (although chronologically one of the last) is MEDEA [37], a special processor to separate a continuous stream of coordinates into different sets, each set originating from a different wire plane. In addition, clusters of "sparks" are detected and transformed into a coordinate corresponding to the center of the cluster. The wire planes can have an arbitrary number of wires. Note that this processor could have been avoided with another design of the read-out system.

Three groups (not counting the detailed example to be given later) tackled the problem of detecting straight particle tracks in wire chamber set-ups.
The first one [39] uses a purely arithmetic method to detect if in a single particle event the points in 3 detectors lie on a straight line. If these detectors are indicated by A, B and D respectively, then the relation checked is:

$$|X_A - X_B + X_D - C_X| < \epsilon$$

and similarly for the Y-coordinates. $C_X$ and $C_Y$ are constants. Three adders and a comparator are used to check (10) for one coordinate. When (10) is satisfied for both X and Y, the event is rejected, since the experiment is intended to study elastic scattering and the chambers A and B are in front of the target and chamber D behind it. This hardware has been effectively used in an experiment.

The second approach [39, 40] is very similar to a number of the bubble chamber approaches (BRUSH, LER): the "sparks" for each detector are stored as bits in shift registers, the position of the bit being an image of the position of the spark. Instead of using pre-wired patterns to search for coincidences, the different angles are scanned by first off-setting the shift registers by the amount required for the search angle. The complete shift register array is then shifted along a single coincidence unit. By ORing successive bits in this coincidence unit, the resolution can be varied. The method is not very fast: for four 128-wire proportional chambers, 1024 4-bit wide searches must be made, taking a time of 100 $\mu$s. Adding I/O time and the time for high-resolution searches one arrives at $\approx 600 \mu$s per view. This time only depends linearly on the number of tracks.

The third processor is more sophisticated [43]. First it finds in a set-up of 4 parallel wire chambers all combinations of 3 points on a straight line. This is done in the "Intelligent Memory", for a maximum of 16 particles. The IM has 256 locations where first all possible values of $C_{1j}X_i + C_{2k}X_j$ are stored for $i=1,\ldots,16$ and $k=1,\ldots,16$. These values are then compared with the values of $C_{2j}X_k$ (for $j=1,\ldots,16$). When a match is found, the three points $X_i, X_j$ and $X_k$ in chamber 1, 2 and 3 lie on a straight line. A bit is then stored in another memory C, at an address which is directly related to the address in the Intelligent Memory where the match was found. This address is in fact a pointer to the combination of points on the line. The process is repeated for chambers 1, 2 and 4, then for 1, 3 and 4 and finally for 2, 3 and 4. A cleaning-up process is then performed to combine for each particle the four lines through 3 points into a single one through 4 points. The approach seems neat and able to deal easily with inefficiencies.

Lastly, and before we leave this survey to treat in detail a hardware processor which is able to deal with curved tracks as well, it is worth mentioning the polynomial generator of McPherson and Wilde [43]. This processor can evaluate other formulae as well, and its application is not limited to high-energy physics. A block diagram is shown in Figure 7. The structure reveals a nice implementation of Horner's rule for the evaluation of a polynomial:

$$P_n(x) = c_0 + c_1x + c_2x^2 + \ldots + c_n x^n = ((\ldots(c_{n-1}x+c_{n-2})x+c_{n-3})x+\ldots)x+c_0$$

(11)
Figure 7 - McPherson-Wilde Polynomial Evaluator

The processor is flexible due to its micro-programmability. It probably will not beat a large computer in speed, but it can be very useful as an extension to a minicomputer without hardware multiplier.

SPECIAL HARDWARE PROCESSORS FOR A SPECTROMETER SET-UP

We will now describe in more detail a number of processors which have been designed (and some built) for track-recognition in a spectrometer set-up using proportional wire chambers. We will describe the processors, investigate why they are faster than programs running on a Control Data 7600, describe how further improvements can be obtained and how the operation of different processors can be co-ordinated to perform the overall task.

Lay-Out of the Experiment

Figure 8 - Schematic of Experimental Lay-out
The spectrometer set-up is shown schematically in Figure 8. In this set-up one wants to study the secondary particles produced in head-on collisions of two protons. The particles are produced in a relatively small region (~20 x 4 x 2 cm²), indicated at the left. Note that the vertical scale in Figure 8 is much larger than the horizontal. The secondary particles traverse a number of detectors, where the X and Y coordinates of the traversal point are measured. One of the detectors is placed in a more or less uniform magnetic field. Inside the magnetic field the particle trajectories are curved and the amount of curvature is in fact a measure of the momentum. Figure 8 shows a complicated event, where a larger number of positively and negatively charged particles are produced and traverse simultaneously (i.e. within the resolution time ~0.1 µs of the electronics) the set of detectors. The problem we are faced with is: how do we distinguish tracks in this set up? In Figure 8 the complete trajectories are drawn in, but the only information we obtain from the experiment are a set of (X,Y)-pairs in each detector. So, it looks in fact as in figure 9 which represents the same event as figure 8! We want to obtain from this, for every particle in the event, a list of five (X,Y)-pairs, representing five points along its trajectory and thus describing this trajectory entirely. This then will allow us to determine the three components of the momentum, make kinematic calculations and analyse the event to extract the physics information of interest.

The problem is slightly more complicated than this, for we do not directly measure (X,Y) coordinate pairs to start off with! The detectors used are multi-wire chambers, consisting of several wire planes. A large number of thin wires are strung parallel to each other in a plane and kept at a positive potential with respect to the outer planes, parallel to the wire-plane and at a distance ~1 cm. When an ionising particle passes through such a chamber, electron multiplication takes place in the strong electric field around the wire and the wire nearest to the traversal of the particle will produce an electrical signal which can be detected. Since the number of the wire which gave the signal is known, we have measured one coordinate (say X) on the trajectory of the particle. With the help of another chamber, with the wires strung in a direction perpendicular to the first, we can measure the Y coordinate. For a single particle event we thus measure the (X,Y) pair but in a multiparticle event we obtain a set of X-coordinates independently.
of the Y-coordinates which have to be combined into (X,Y) pairs. This pairing gives rise to ambiguities. To solve the ambiguities that occur in pairing the X's with the Y's (see Figure 10 for the simple case of 2 particles), more wire planes are added and sandwiched together with the X and Y planes in a single module. The added planes will have the wires strung under an angle with the X-axis. A typical module is schematized in the inset of figure 8. The wires here are strung at $\pm 45^\circ$ and they will measure what we will call the U and V coordinates.

To summarize: the electronic read-out system will give us sets of X, Y, U and V coordinates, all independent of each other, for all five detector modules and for a number n particles (a total of 5x4xn coordinates). From this we must find first (X,Y) pairs in each module (a total of 5n (X,Y) pairs). After this point-finding we must sort the points into tracks (to get n tracks).

**The Pattern Recognition Process**

i) **Point-finding.** When we apply a simple coordinate transformation from the X-Y-system into the U-V system we see immediately that the four coordinates measured for a particle must satisfy the following relations:

$$ U = \frac{1}{2}X + \frac{1}{2}Y $$  \hspace{1cm} (12)
$$ V = -\frac{1}{2}X + \frac{1}{2}Y $$  \hspace{1cm} (13)

In the particular case we are considering the distance between the X and Y wires is 2 mm, but the U and V wires are spaced by $2\sqrt{2}$ mm. The result of this is that we get rid of the $\sqrt{2}$ in (12) and (13), when we express the coordinates in wire-number, instead of millimeters. The relations (12) and (13) are only satisfied for a particle whose trajectory is perpendicular to the planes. When this is not the case, then due to the finite thickness of a module, (12) and (13) can only be satisfied
approximately. So, the four coordinates of a particle must satisfy the following constraints:

\[ |X+Y-2U| \leq \epsilon_1 \]  
\[ |-X+Y-2V| \leq \epsilon_2 \]

Thus, the problem is to find all combinations of an X, a Y, a U and a V, satisfying (14) and (15) simultaneously. This involves a nest of loops, as in figure 11-i. Matters get somewhat more complicated if we

![Diagram of loop structure for point-finding]

**Figure 11 - Loop Structure for Point-Finding**

realize that the efficiency of a detector plane is not exactly 100%. In other words, one can expect to find particles for which, say, the U coordinate is missing, because the particle did not produce an electrical signal on a wire. In that case (15) is the only constraint that can be satisfied. Similarly for a missing V only equation (14) can be satisfied. In order to find also particles for which the X or the Y coordinate is missing, we must use the inverse relationships:

\[ |U+V-Y| \leq \epsilon_3 \]  
\[ |U-V-X| \leq \epsilon_4 \]

For this purpose we must also perform a search with the loop-structure of figure 11-iii, after executing the loops of figure 11-i.

In real life we must also deal with spurious signals, without any relation with an incident particle. Since all the constraints are to be satisfied within a finite value \( \epsilon \) (\( \epsilon \) is normally 5 or 6) there is a non-negligible probability of finding spurious solutions. These so-called ghosts consist of combinations of X, Y, U and V, which satisfy the constraints by pure chance. Since at the level of a chamber module there is no way of distinguishing a ghost from a true solution, the ghost solutions must be carried through into the track-finding. They increase the number of points participating in the track-recognition process. The number of ghost solutions increases with the square of the total number of signals per wire plane (partic.e+spurious signals) and the problem can become serious when there are many "noisy" wires.
ii) Track-finding. We now turn our attention to the track-finding process. The method of principal components offers an elegant method of treating the problem. Recall figure 9 to realize that the problem is not trivial!

![Diagram](image)

**Figure 12 - Principal Components in 2-Dimensions**

Instead of presenting immediately a formal mathematical description of the method, we will first give an example in two-dimensional space. Suppose we measure two quantities \( x_1 \) and \( x_2 \) to determine the value of a parameter \( p \) describing a particle trajectory. We suppose we only want to determine one parameter and therefore the measurement of two variables is redundant. In other words, a constraint equation relates the value of \( x_1 \) to the value of \( x_2 \). See figure 12 for a representation. The curve represents the constraint, and the parameter \( p \) takes on different values along the curve. A measurement of \( x_1 \) and \( x_2 \) which does not correspond to a point on the curve (or near to it) must be rejected as non-physical. When the curve is sufficiently close to a straight line (or rather when the probability distribution is well allongated), the test to check if a \( (x_1, x_2) \) pair represents a physical situation or not can be simplified by applying a coordinate transformation from the \( (x_1, x_2) \) system into the \( (z_1, z_2) \) system (see figure 12). The test criterion then simply becomes: \( |z_1| < \varepsilon \) where \( \varepsilon \) is small. The value of \( \varepsilon \) is a direct measure of the parameter \( p \).

In the case we are considering of particle trajectories in 3-dimensional space, we can apply the same method, but we must increase the dimensions of the space we are working in. In fact, a particle trajectory has 5 degrees of freedom, when a magnetic field is present. Or, to put it differently, a trajectory is entirely determined by 5 parameters: \( p_x, p_y, p_z \) (the 3 components of the momentum) and \( (X, Y) \) at some plane \( z = \text{constant} \). But we measure 10 values (5 coordinate pairs) along the trajectory in the set-up of figure 8. Thus there must be 5 constraint equations between the measured values \( x_1, \ldots, x_{10} \). Each trajectory can be represented by a point \( (x_1, \ldots, x_{10}) \) in 10-dimensional space. It now turns out that, for a wide range of experimental conditions, the points plotted this way for a large number of tracks, lie close to a 5-dimensional hyperplane. (Compare with figure 12 where the 2-dimensional plot is close to a 1-dimensional "hyperplane"). This, of course, simply means that the constraint equations are linear. Or, in other words, it means that we can define a coordinate transformation
\[ \xi = W \cdot x \]  \hspace{1cm} (18)

which transforms from the system \( x_1, \ldots, x_{10} \) into the system \( \xi_1, \ldots, \xi_{10} \) in such a way that the \( \xi_1, \ldots, \xi_5 \) axes subtend the hyperplane and that the \( \xi_6, \ldots, \xi_{10} \) axes are perpendicular to it. This then means that for a given trajectory and thus a point \( (x_1, \ldots, x_{10}) \) in the 10-dimensional space

\[ \xi_6^2 + \xi_7^2 + \xi_8^2 + \xi_9^2 + \xi_{10}^2 \]

is the distance of the point to the hyperplane. This quantity is small for a possible trajectory and can take any value for an arbitrary point in the 10-dimensional space.

![Figure 13 - Range of values for \( \xi_1 \) to \( \xi_{10} \)](image)

The range of values taken by \( \xi_1 \) to \( \xi_{10} \) for a set of 1000 tracks is illustrated in figure 13. For each value of the index \( i \) is plotted the largest value of \( |\xi_i| \) found amongst the sample of 1000 tracks. The dotted line interconnects the points obtained when the original \( x \)-coordinates are known without error. This line is a measure of the "flatness" of the hyperplane. When measurement errors are introduced the smaller values of \( \xi_i \), occurring for the larger indices \( i \), are drowned in the errors. The solid line in figure 13 shows this effect.
This now leads to the algorithm to recognize tracks. We first rename our coordinates: \((X,Y)\) in the first chamber module becomes \(x_1, x_2\), in the second module \(x_3, x_4\), etc. We then calculate for every possible combination of one point in the first module, and one in the second, one in the third, etc...

\[
\xi_i = \sum_{j=1}^{10} W_{ij} x_j \quad \text{for } i=6, \ldots, 10. \quad (19)
\]

When each \(\xi_i\) is small:

\[
|\xi_i| \leq \varepsilon_i \quad \text{for } i=6, \ldots, 10. \quad (20)
\]

and/or when

\[
\sum_{i=6}^{10} \varepsilon_i^2 \leq \Delta \quad (21)
\]

we accept the chosen combination of points as one defining a possible trajectory. If the conditions (20) and (21) are not satisfied we reject the combination.

To apply the method we must of course know the matrix \(W\). This knowledge can be obtained with any precision required from a Monte-Carlo sample of events. \(W\) is nothing more than the set of eigen vectors of the covariance matrix of the sample \([44], [46]\).

A very important thing to know is how well the method does discriminate against wrong combinations. Figure 14 illustrates the quality of the discrimination attained \([46]\). The peak at the left contains all the correct tracks and the large hill at the right all the wrong combinations. Note that the horizontal scale is linear from 0 to 1 and then becomes logarithmic.

![Figure 14 - Separation of Good Tracks](image)
The number of combinations that can be formed with $n$ points in each of the five detectors is $n^5$ and they should in principle all be tested. Obviously this becomes a big and time consuming task for larger $n$. The CPU time it takes - in Fortran - on a CDC 7600 is shown in figure 15 (curve marked "brute force"). It is therefore essential to reduce the number of combinations to be tested and fortunately this is possible in most practical cases: when the magnetic field is reasonably uniform and the particles have a high momentum, the tracks are close to straight lines in one projection (say $Y$). A check to see if $x_2, x_4, x_6, x_8$ and $x_10$ lie approximately on a straight line can be done in a time proportional to $n^3$ (instead of $n^5$ as with "brute force"). Only for those combinations which pass this straight line test does one then need to calculate (19). Figure 15 shows the gain obtained.

From the point of view of a hardware implementation, this track recognition algorithm is ideal: a straightforward, simple calculation which can be performed in a nested loop structure, followed by simple tests. It should be pointed out, however, that the method has its limitations:

i) the detector planes must be parallel.

ii) the transformation $W$ is only valid for tracks crossing all five detectors.

For other combinations of detectors (the first three, for instance) another transformation must be applied.
Point-Finding Processor

The block diagram of a hardware implementation of the point-finding process is shown in Figure 16. This processor works on data from one single 4-plane module at a time. The X, Y, U and V coordinates are stored in independent scratch-pad memories, in the order given by the read-out system, i.e. in increasing order. The scratch-pad memories can store 16 or 32 words of 16 bits. They have their individual address register (AR), word count register (WCR) and number of active coordinates register (ACR). The word count register furnishes in fact a pointer to the last address in memory containing useful data. When a memory is loaded, all three registers are incremented until the last coordinate from the wire plane has been stored and a switch-over is made to the next wire-plane and the next memory. To perform a loop over the data contained in a memory three control signals are enough: set address register to zero (ZAR), increment address register (IAR) and "last address reached" (LA). The last is a signal delivered by the memory module when the contents of AR equal the contents of WCR. It indicates that incrementing of AR must be stopped because the following locations in memory contain rubbish.

Nested loops can be easily implemented in the following way: first all address registers are zeroed. Then AR is incremented until the signal LA1 is produced. The presence of LA1 causes the following two actions: ZAR1 and IAR2. This is repeated until LA2 and LA1 are present, causing ZAR1, ZAR2 and IAR3, etc.

Note that the relation (14) and (15) are independent of each other and can be tested simultaneously if the necessary arithmetic elements are available. A sufficient number of adders and comparators have been implemented to make this possible. The loop structure then becomes as in Figure 11-ii. One loops simultaneously over the U and the V coordinates, until a "hit" is found. When this hit is for instance on a U coordinate (which means that (14) is satisfied), one stops incrementing the address of U, but the search over the V coordinates is continued. Two things can happen: either a hit on V is found, or the last address of V is reached without finding a hit. In the first case a 4-plane solution has been found, in the second a 3-plane solution with V missing. The latter has, by the way, a fair chance of being spurious.
The adders and comparators at the right hand side of Figure 16 are a kind of a luxury. They allow us to check relations (16) and (17) simultaneously and thus detect some of the 3-plane solutions with either $X$ or $Y$ missing at an early stage. This proved to be unrealistic and now these elements are only used when a specific search for these 3-plane solutions is made with the loop structure of Figure 11-iii.

Whenever an acceptable solution has been found, the coordinates participating in this solution are tagged by writing a "1" into an unused bit position (11 or 12 bits are sufficient for a coordinate). In addition the solution is written into four scratch pad memories which serve as output buffer. On subsequent passes through the data in a particular memory a tagged coordinate can be treated in two ways:

i) The tag can be simply ignored. This will yield all possible solutions, which in the presence of noisy wires can easily amount to 20, 40 or more solutions, with 3 or 4 true particles hidden amongst them.

ii) One can skip rapidly over a tagged coordinate without performing all the arithmetic. A tagged coordinate is thus eliminated and only a limited number of solutions is obtained in which a coordinate appears once and only once. Unfortunately these solutions are not necessarily all correct!

To deal with chambers with noisy wires the following procedure has been adapted: tagged coordinates are ignored. The processor makes four passes through all the data. During pass 1 and 2 only 4-plane hits are accepted. Pass 1 is performed with $c=2$ and pass 2 with $c=6$. Pass 3, with $c=6$ accepts only 3-plane hits where $U$ or $V$ are missing and finally pass 4, also with $c=6$, finds the 3-plane hits where $X$ or $Y$ are missing. Since the probability of finding a spurious 3-plane hit is proportional to $c$ and for a spurious 4-plane hit proportional to $c^2$, this strategy reduces the chances of accepting spurious and wrong solutions. (The trouble is not so much the presence of a wrong solution, but the 3 or 4 correct solutions you may miss when tagged coordinates are ignored.)

Another, better, strategy is to suppress pass 1 and to ignore the presence of a tag during pass 2, but not during pass 3 and 4. All possible 4-plane solutions are then accepted and the search for 3-plane hits is performed amongst the residue of unused coordinates.

We realize by now that the box "CONTROL" in Figure 16 has become rather complicated. It has to do all the necessary things to mest the loops, to deviate from normal procedure when a hit on, for instance, $U$ has been found, to organize different passes and different loop structures. By far the optimum way of implementing this control is by the use of a Programmable Logic Array (PLA). A schematic of a PLA is shown in Figure 17. The left hand side, marked CAM, contains AND-gates, one gate per line. The condition signals are the inputs to the AND-gates. In the case of an implementation with AND-gates, each condition $C_j$ should be presented in the straight ($C_j$) and negated form ($\overline{C}_j$). This allows the use of either one as input to the AND-gate or neither of the two, which represents a "no care" condition. An AND-gate which is satisfied activates an address line which in the part ROM produces the required action signals. The columns in ROM form OR-gates, with a selection from the address
Figure 17 - Programmable Logic Array

It is important to realize that all the combinations of conditions which have been programmed in the PLA are checked simultaneously and the action signals produced immediately. A stored program computer would have to perform a long series of conditional branches (if-statements) to do what a PLA does in some 50 ns. Note that a PLA can be programmed to perform certain steps in sequence. It is sufficient that an action signal sets a flip-flop and that the output of this flip-flop is fed-back as a condition signal. In this way the different passes of the point-finding processor have been programmed. While the processor is executing pass 3, the program for pass 3 is enabled while the other three programs are disabled.

Finally, the similarity between a PLA and a decision table [48, 49] should be pointed out (see Figure 17).
Performance of the Point-Finding Processor

Figure 18 - Photograph of Point-Finder

A photograph of the complete point-finding processor as designed and built by A. Fucci, is shown in Figure 18, with all logic circuitry pulled out of the cabinet. The processor is implemented in normal TTL logic and it contains some 540 IC's, distributed over 44 circuit boards. The component cost (including cabinet, switches, etc.) is ~14,000 SFr., labour for assembly represents ~5,000 SFr.

Figure 19 - Long and Short Cycles in Point-Finder

A complete cycle - from the issue of an increment address signal to the next issue of an IAR - is ~300 ns when a non-tagged coordinate is read and the complete arithmetic is performed. This reduces to ~130 ns when a skip cycle on a tagged coordinate is performed and it extends to ~600 ns when a hit is made and a solution must be stored. Figure 19 shows a part of the execution on an extended time scale, beginning at the start of pass 2.

The PLA has been implemented as a diode matrix (see Figure 18) and
this implementation accounts mostly for the fact that the cycle time is some 50% longer than intended. The program contained in the PLA has about the complexity of the flow-diagram of Figure 20 (this flow-diagram is not claimed to be correct, it is only included for comparison). The PLA uses 20 input signals and produces 18 output signals; 33 sets of conditions are programmed. The program has been translated back into FORTRAN, resulting in a listing of over 2½ pages, compiling into 397 locations on the CDC 7600.

Figure 20 - Flow Chart of Point Finding Program

The execution time for the point-finding process is found to be proportional to the third power of the number of signals per plane, as expected. An event with 6 particles is handled in 74 μs (not counting I/O time). This is 25 times faster than the Fortran program on the 7600, compiled with the latest version of the compiler (FTN 4.1 + 69) using full optimization. (For an earlier version of FTN the 7600 performed the algorithm 40 times slower [50 - 52].)
The reasons for this increase in speed can be easily identified:

i) parallelism. The use of several scratch-pad memories avoids sequential access to data.

ii) fast access. The memories have an access time of 35 ns, which is increased to ~50 ns because the AR must be incremented. Also the arithmetic is fast: ~35 ns per addition.

iii) duplication. In order to check two relations at a time, the arithmetic has been duplicated. In fact every addition uses a different adder, which saves gating delays at inputs and outputs of adders.

iv) use of PLA. This gives without doubt the most important contribution to the speed-up factor.

Hardware Design for Track Finding

![Block Diagram of Track Finder](image)

Figure 21 - Block Diagram of Track Finder

The algorithm of equations (19), (20) and (21) is easily implemented in a fast hardware processor represented in Figure 21. The trick consists in performing all the multiplications required by (19) beforehand, outside the nested loop. Suppose we want to check only $\xi_7, \xi_8, \xi_9$ and $\xi_{10}$ and not $\xi_6$. Suppose also that we limit ourselves to a maximum of 8 points. We will then store in, for instance, the 3rd scratch-pad memory in Figure 21 the following data: at address 0, 1, etc.: $W_7, X_3(1), W_7, X_3(2), W_7, X_3(3)$, etc.

At addresses 8, 9, etc.: $W_8, X_3(1), W_8, X_3(2), W_8, X_3(3)$, etc.

At addresses 16, 17, etc.: $W_9, X_3(1), W_9, X_3(2)$, etc.

and finally at addresses 24, 25, etc.: $W_{10}, X_3(1), W_{10}, X_3(2)$, etc.

The subscript refers to the $i$ and $j$ in (19), while the index is the ordinal number of the point ($X_3(2)$ is the x coordinate of the second point found in module 2). The point-finding process has established already the linkage between the X and Y coordintes of each point. We will therefore use
all possible combinations of points by performing over the data a series of loops, nested 5 deep (and not 10 deep as Figure 21 might suggest). Moreover, we loop only over the first 8 locations (or less if there are less points) of each memory.

For every combination of points we form in the course of executing the loops, \( \xi_7 \) will be calculated by the adder tree. When it turns out that \( \xi_7 \leq \delta \) we have a possible track candidate. \( \xi_8, \xi_9 \) and \( \xi_{10} \) are then rapidly calculated by increasing all memory addresses by \( 8 \). We have still some liberty to choose the selection criteria for accepting a track: require (20) to be satisfied for all \( i=7, \ldots, 10 \), or require that only 3 relations out of 4 need to be satisfied. These criteria are illustrated in Figure 21. One can also require that (21) be satisfied (using for instance a look-up table to find the square of a small number), or replace the sum of squares by the sum of absolute values. Each coordinate must be multiplied in the beginning by 4 constants. These 4 multiplications can be performed in micro-seconds. Thus when the processor is loaded from a minicomputer, no time is lost in the multiplications. The loading of the processor with, for instance, 4 points per chamber module can be performed in \( 2 \times 4 \times 5 = 40 \) \( \mu s \). Outputting the final track data can also be done in \( 40 \) \( \mu s \) or less. The loops can be executed in a maximum time of \( n^5 \cdot \tau \) and an average time of \( \frac{4}{3} n^5 \cdot \tau \). For \( n=4 \) particles and a cycle time \( \tau = 500 \) \( \text{ns} \) we obtain an average time of \( \frac{4}{3} \times 1024 \times 0.5 \) \( \mu s = 256 \) \( \mu s \). This neglects the few microseconds required to check \( \xi_8, \xi_9 \) and \( \xi_{10} \). Once \( \xi_{10} \) has been accepted. This design can obviously be modified for another number of detectors, another maximum number of points, etc. It suffers however from a few shortcomings:

1) the \( n^5 \) dependence of the execution time. For 8 points per module the execution time increases to \( 8 \) \( \text{ms} \).

2) the processor is capable of finding only those tracks which have given rise to a point in every chamber module.

This is not necessarily the case for all tracks, either because of inefficiencies or because a track has been bent too much in the magnetic field and it misses the last two detectors. To find these special tracks requires another loop structure (which is not difficult as we have seen), but it also necessitates going back to the beginning and multiplying the coordinates by another matrix \( W \). This is rather awkward in this structure.

More Sophisticated Track-Finding Processor

The remedy to both shortcomings is to perform first a straight-line search on the Y-coordinates alone. This, of course, will give a good pre-selection of track candidates only when the magnetic field is sufficiently uniform so that vertical focussing or defocussing effects are small enough. The search for a straight line can be made in a time proportional to \( n \), where \( n \) is as always the number of points per detector. Particularly for large events a considerable speed-up can be expected. When a straight line has been found, it must be confirmed that this is indeed a track by performing the 3-dimensional principal components algorithm (19) on the points. Since the selection criterion for finding a straight line will be quite loose, formula (19) will have to be evaluated more often than corresponds to the number of tracks. This drawback is however largely offset by the overall gain in execution time. The special cases of tracks not making the full traversal of all detectors or suffering from inefficiencies can be treated at the level.
of the straight-line search.

When such a "special" straight line has been found, the combination \( K \) of detectors in which it was revealed will then determine immediately the matrix \( W \) which must be applied in (19) to confirm the existence of a track. The elements of all the possible matrices \( W \) can be stored inside the track-confirmation processor. The physicist should however exert some self-discipline in this respect. If in a set up of 8 chambers one wants to detect the tracks which traverse 8, 7, 6, ..., 3 chambers in all possible combinations one needs

\[
1 + \binom{8}{2} + \binom{8}{3} + \binom{8}{4} + \binom{8}{5} = 219
\]

different matrices, varying in size from 16x11 down to 6x1, making a total of 8352 coefficients! A careful choice must therefore be made of the limited number of combinations of detectors to be considered.

![Diagram of Line Finder](image)

**Figure 22 - Diagram of Line Finder**

A design for a straight line finder is shown in Figure 22. The algorithm is the following: choose a \( Y \)-coordinate in the "first" chamber and one in the "last". Connect the two by a straight line and predict the \( Y \) coordinate for the three "intermediate" chambers, using:

\[
Y_{\text{pred},i} = Y_{\text{first}} + \beta_i (Y_{\text{last}} - Y_{\text{first}})
\]

The predicted values are then compared with the measured coordinates in the inner chambers. The inner loop over the data for the three intermediate chambers can be performed in parallel. Also the three predictions can be calculated in parallel. In the present design this is done sequentially, to avoid the need for 3 expensive, fast multipliers (2000 SFs. each). For fast and parallel access the coordinates are stored in scratch-pad memories, one for each detector. So \( Y_{\text{first}} \) and \( Y_{\text{last}} \) (OF and OL in Figure 22) are subtracted and multiplied by three \( \beta \)'s which are read sequentially from a central memory (CM). The resulting predictions are stored in three of the five registers LR\(_{1}\) to LR\(_{5}\).
The inner loop is then performed by presenting all Y's for the intermediate chambers (20Y, 30Y and 40Y). Figure 22 shows that the choice of the "first" and the "last" chamber is arbitrary. For each choice another set of ε's and β's must be addressed in CM and the control logic must choose the correct combination of LR's and CMP's to be used. (The latter choice can even be avoided by using always 5 values of β, two of which are β=0 and β=1. All 5 LR's and all 5 CMP's will then be used, always.) Figure 23 shows how the choice of first, last and intermediate chambers is made: one of the signals ENF and one of the ENL's are active. Note that the data for the innerloop are available without additional gate delays. Figure 23 shows also the storage for the X-coordinates and the gates necessary to sequentially read out 10 coordinates for track-confirmation.

Figure 24 - Indirect Addressing Scheme
In order to deal with the different combinations of $k$ out of $m$ detectors, several passes are performed through the data. A pass counter (see Figure 24) forms the basis for an indirect addressing scheme to retrieve the correct $\varepsilon$'s, $\beta$'s and $W_{ij}$'s from CM. An undisturbed copy of the starting addresses is kept in CM itself. From here addresses can be transferred into IAL at the beginning of a new pass, if necessary after some manipulation. In IAL the addresses can be freely incremented to perform sequential accesses. The pass counter is not the only variable entering into the addressing scheme. Other factors are the kind of variable that must be retrieved: $\varepsilon$, $\beta$ or $W_{ij}$. In addition there is an entry "arm". The whole processor is in fact designed for a two-arm spectrometer. Using the fact that one is treating arm 0 or arm 1 as a parameter allows the access of entirely different constants for the two arms. In other words: the two arms can have different configurations. It is obvious from Figure 22 and 23 that the design of the straight line finder can be easily adapted to any number of detectors. Also a slight modification in the gating in Figure 23 would adapt the processor to find straight lines in X as well as in Y. The line-finder is therefore a processor in itself which could find application in experiments where the principal component approach is not needed.

A preliminary version of this line-finder has been built and it has been tested on Monte-Carlo data. At present it is being tested on real data from an experiment, recorded on magnetic tape. The aim is to compare its performance with the results obtained by software.

![Figure 25 - Diagram of Track Confirmation](image)

The track-confirmation part of the track-finding processor, shown in Figure 25, is simple. The coordinates are obtained sequentially (that is in the order of $X_1, X_2, X_3, \ldots, X_{10}$) from the scratch-pad memories. At the same time the appropriate $W_{ij}$'s are obtained - also sequentially - from CM. After multiplication the terms are accumulated in REG1. A first comparator compares the $\xi_i$ obtained with an $\varepsilon_i$ retrieved from CM. An adder adds or subtracts $\xi_i$ depending on its sign from the sum accumulated in REG2. REG2 will thus contain the sum of absolute values $\sum |\xi_i|$, which at the end is compared with another $\varepsilon_{\text{total}}$.
It is evident that the same processor can be used for track confirmation in an arbitrary number of detectors. It is thus very general.

The results from the track-confirmation are written back into CM, from where they can be transferred to the control computer for recording on magnetic tape. In order to avoid detecting in later passes segments of a track already found in an earlier pass, coordinates which lie on a track are tagged. The tagged coordinates are ignored in later passes where the number of detectors considered is smaller.

The track-finding processor is controlled by PLA's. Instead of one very large PLA controlling everything, the control has been split up. There is one PLA for overall control and a number of others controlling sub-units. The design of the track-finding processor is mainly due to W. Vree.

POSSIBLE IMPROVEMENTS

Use of Content Addressable Memories

Both in the point-finder and line-finder the inner loop consists of the comparison of a predicted with a measured value. This is done by looping over all measured values \( V_m \), subtracting them from the prediction \( V_p \), and comparing the absolute value \(|V_p - V_m|\) with a small and positive \( \varepsilon \). When there are \( n \) measured values this takes a time \( nT \). With the use of a Content Addressable Memory (CAM) this time can be reduced and thus the operation of the processor speeded-up.

A CAM is orthogonal to a random access memory (RAM). To the latter one presents an address and it produces as output the contents of the location addressed. To a CAM a value is presented and, if this value is stored in a location in the CAM, the output produced will be the address of that location. A CAM will therefore indicate if there is an exact match between a predicted value and a measured value, stored together with all other measured values in the CAM. This comparison of the predicted value with all measured values is done in one single memory cycle. An additional facility of a CAM is that certain bit positions can be masked out, so that they do not take part in the comparison. In our case we are not interested really in exact matches, but only in approximate ones, within \( \pm \varepsilon \). This complicates the use of CAMS.

A first idea that comes to mind is to mask out the least significant bits when a comparison is made. This does not work nicely however. Suppose that the predicted value is \( 162_8 \). Masking out the 3 least significant bits will then reveal matches falling in the range 160-167, that is between the asymmetric limits +5 and -2. But things can be even worse: suppose that \( V_p \) is \( 177_8 \). To find then a match within a range of \( \varepsilon = +5 \) for instance, a completely different pattern (200_8) must be presented. Sumner has proposed a scheme where two comparisons at most are required, followed by a final precise check using a subtractor and a comparator. Suppose \( \varepsilon \leq 4 \). At first a comparison is made with \( V_p \), masking out the 3 least significant bits. The 3 least significant bits of \( V_p \) are inspected and a second comparison is made either with \( V_p - \varepsilon \) when the 3 least significant bits form a number \( \geq 4 \). In both cases the 3 least significant bits are masked out. When a match is found, the final check is made, comparing \(|V_p - V_m|\) with \( \varepsilon \) in full precision. Due to the possibility of finding multiple
matches, the control logic becomes quite complicated.

Fucci has proposed another scheme, which is a priori slower, but simpler to implement and where multiple matches present no problem at all. His scheme leads to a new and really splendid design of the point-finder; shown in Figure 26. As in the old design, two programs are executed, one with the outer loops over X and Y, the other with the outer loops over U and V. During execution of the first program any combination of X and Y is handled by the arithmetic to form X+Y and Y−X. These two values are loaded into registers/counters, shown in the top right hand corner. The search for matching U and V values now starts. If a match is found immediately, fine. If not, the register containing Y+X is incremented by one and a new comparison made with the U values. It goes without saying that the search for V proceeds in parallel. This is repeated until the value Y+X+ε is reached. Multiple matches within this range will be found in sequence and so present no problem. In reality the register containing Y+X is doubled: one is incremented until Y+X+ε, the other decremented until Y+X−ε.

![Diagram](image)

**Figure 26 - Point Finder Using CAMs**

The design provides for overlapping of the search and increment/decrement operations: while Y+X+ε something is presented to the CAM, Y+X−something is decremented and vice-versa. At first sight the gain seems minimal: n comparisons have been replaced by 2ε+1 searches. But the cycle times must be taken into account as well: 320 ns total time for the inner loop for one, against (2ε+1)·80 ns for the other. The method using CAM's is therefore faster whenever ε ≤ (4n−1)/2 or ε ≤ 2 n.

The second program is executed by looping normally over the U, V memories and performing the search in the X, Y memories. This design still holds another surprise: the two programs can be executed simultaneously! For this it is sufficient to alternate increment/calculate with search, but in both halves of the processor at the same time.
Another use of CAM's should be mentioned, where exact matches are in fact required. This to eliminate a posteriori point or track solutions which have been found already before. If for instance a 3-plane solution is found which is already contained in a 4-plane solution, a search in a CAM containing all solutions would reveal this. The missing coordinate must of course be masked out. It is more economical to work with pointers than with coordinates (less bits).

A sombre note on which to end is that CAM's are expensive and difficult to obtain.

Pipelines

Pipeline structures have been invented to improve the throughput of arithmetic units [55] and of complete CPU (instruction pipeline of M55, [56]). We are rather interested in pipelining macro-operations to improve throughput. This can often be achieved at the price of increasing the storage capacity inside the processor.

\[ \text{Figure 27 - Overlap of Processing and I/O} \]
A first example is the overlapping of Input/Output and processing. This requires that the memories for the coordinates be doubled, so that input can proceed into memory I, while processing is done on the contents of memory II. This is illustrated in Figure 27. It is clear from Figure 27 that the effectiveness of pipelining depends on the duration of the different parts. If the time for (P+I+M) would largely exceed (L+S) it might pay off to pipeline point and track-finding. Intermediate storage between the two is then required.

Other examples where something can be gained can be found in the track-finder: when 10 registers are added to store one set of $X_1, ..., X_{10}$, track-confirmation and straight line finding can be overlapped. If it pays off depends on the ratio: time to confirm a track/time between finding a line and the next. This ratio is

$$\tau \approx \frac{50 \times 400 \text{ ns}}{18 \mu\text{s}} \approx 1.$$ 

for 6 points per detector, giving rise to 4 tracks in one single pass. In this particular case, it would pay off, but matters are complicated because $r = \frac{1}{n^2}$.

Another example is in the calculation of $\xi_i$: the multiplication and the addition can be pipelined. But as long as both can be performed sequentially within one CM cycle, there is no point in doing it.

Every possibility for pipelining must be judged on its merits, which in general can only be done when the design is already well advanced.

**Cooperation Between Processors**

The total pattern recognition process for the spectrometer set-up of Figure 8 can be organized in a number of ways. A single point-finder can handle the five chamber modules one after the other, or five point-finders can be used, operating on the five chambers simultaneously. The choice will depend on a trade-off between the throughput required and the money available. Whatever the choice, track-finding cannot start before all points in all chambers have been found. This asks for some degree of cooperation between the different processors. The cooperation must extend over two fields: overall control and the communication of necessary data. The first can in general be solved by the implementation of an overall control program, using a PLA. The PLA produces start and stop signals, interrupt requests, and it can emit enable and disable signals to other, more local PLA's. It can thus control the processing on the basis of availability of input data, busy or ready states (including the minicomputer), etc.

The communication of data presents other kinds of problems. In our particular case the U and V coordinates were essential for point finding, but they are redundant for track recognition. Should they be discarded when no longer needed? Probably not, because their information content is not negligible and they can contribute to the precision of the final momentum calculation (performed almost certainly on a large scale computer). So they must be stored somewhere. If we continue to use scratch-pad memories for this purpose, we will end up by using really large numbers of them, also at
places where the fast access and control facilities are of no use. If in
addition we think of overlapping point and track finding these intermediate
storage requirements are doubled.

It therefore seems best to communicate data from a processor to
the next via a Central Memory, which must be rather big (1K-2K of 16-bit words),
but need not have very fast access. A cycle time of 300 ns seems adequate
and can be easily obtained with present semiconductor memories. We get a
lay-out for the CM more or less as in Figure 28. The general data area can

![Memory Layout Diagram](image)

Figure 28 - Memory Layout

have an arbitrary length. It contains general event data, like date, time,
scaler read-outs, magnet currents, etc. They are completely useless for the
processor, but it is convenient to treat them the same as the other event
data. The bulk of the event data consists in fact of "spark" coordinates,
which are stored in separate areas. From here they are transferred area after
area into the scratch pads of the point-finder. The point-finder writes the
results back in the area, overwriting in fact the raw data. Possibly the
"unused" data (the spurious signals, not attributable to a point) will be appended. When all points have been found the necessary coordinates are transferred to the straight-line finder and the track confirmation processor, which writes pointers to coordinates back into the last area of CM.

Since access to CM is mostly required for four words at a time, the bandwidth can be improved by making the memory 64 bits wide. The way points and confirmed tracks can be stored is indicated in Figure 28.

USE IN THE EXPERIMENT

Comparison of the input/output times (assuming 1 μs/word DMA transfers without overheads) and inspection of Figure 27 make it clear that maximum throughput can be obtained only when the special processor is placed between the data source (CAMAC) and the data acquisition computer. This however presents some difficulties, not the least important being the necessity that the experimenter has full confidence in the processor. There are others as well: the processor must be loaded with constants and the final values of these constants can only be found by carefully analysing a sample of events. It remains to be investigated how precisely the constants must be known to perform track recognition without any pretension to calculate momenta.

For this reason and for the time being special purpose processors will probably be connected as a peripheral to the minicomputer. Data will therefore pass a number of times through the core memory of this mini. In the experiment for which the point and track finder are being constructed the raw data will in a first stage be written onto a digital video tape. The processors will only intervene during play-back of the video tape. The results from the processors will then be used to select events presenting a desired topology. The selected events will be further analysed, the other discarded (the raw data of the discarded events is however kept on the video tape).

Another point of consideration in the use of processors is the format of the input and more importantly, of the output. There is little or no point in producing the final output in such a cryptic form, that to unpack it a large fraction of the original processing gain is lost. Efficiency of the processor required however that one should store pointers, rather than coordinates at some places. A post-processor in the output data path is the solution. This post-processor would perform with great ease the unpacking and retrieving tasks which are often stumbling blocks in later processing.

In the same way a pre-processor will often be needed. A very good reason for having it is that all our formulae - and therefore the processors - supposed that all coordinates are measured in one unique system. This is not the case and a simple transformation must therefore be applied to every coordinate:

\[ X' = K \pm X. \]  (23)

This puts the origin at the right place and it can correct for a read-out which runs the wrong way round!

There may be other reasons why a pre-processor is needed. MEDEA in fact can be considered as a kind of pre-processor.
CONCLUSIONS

An important question remains to be answered: "How much processing on the 7600 do we really gain by having special processors?". At present we can only extrapolate to arrive at a guess. The factor 25 in execution between a processor and Fortran on the 7600 is only applicable to a fraction of the total analysis. Figures from a previous experiment using a similar spectrometer indicate that some 20% of the total time was spent in point and track finding. This increases to ~30% when no histogramming is done. In this experiment an average of 1 ½ particle per chamber was found. Assuming the n² dependance we extrapolate that for an event with 4-5 points per chamber some 80% of the analysis time will be spent on point and track finding. This fraction of the time will be eliminated by the use of a hardware processor.

Being pessimistic we can thus expect a gain of a factor 2 to 3 in the computer time needed for analysis.

This is valid for the events which are analysed. But much bigger gains can be expected when an event selection can be made on the basis of the topology which is known after track recognition.

The objection which is most often voiced against hardware is its supposed inflexibility. Obviously hardware is "harder" to change than software. (This is not only a disadvantage, it is also "harder" to make a mistake in hardware since there will be a tendency to check carefully before implementing.) We hope to have shown that none of the processors is really dependent on the details of the experimental layout and that they can all be extended or modified to another number of detectors, other relative positions, etc. Also the use of PLA's make it possible to drastically change the logic of a processor in a matter of hours.

All in all, we hope to have shown that special purpose hardware can be designed, and that it can be constructed with today's easily available techniques without having recourse to advanced technologies which are outside the possibilities of a non-specialized laboratory. Moreover we believe that the use of special hardware can save considerable amounts of computer time. But one cannot construct a processor without thinking well beforehand!

Acknowledgements

C. Rubbia first suggested to us the development of special processors for track recognition. We are very grateful for his and P. Zanella's continual interest and encouragement.

The author is grateful to the Institute of Electrical and Electronics Engineers for permission to reproduce Figures 1 (from reference 4), 2 (from reference 17), 4 and 5 (from reference 3). I want to thank my colleagues Adolfo, Brian, Carlo, Joop, Michel, Mike, Paolo, René and Win for their contributions, helpful discussions and especially for their enthusiastic collaboration. I thank Frank Summer very warmly for the many things he taught me. Finally I must thank the Organizing Committee for their kind invitation and the secretaries and typists for their hard labour.
References


35. L.O. Hertzberger and W. Vree, private communication.


44. J.H. Friedman, "Data Analysis and Presentation", these Proceedings, pp. 271


47. M. Hansroul, D. Townsend, P. Zanella, "The application of multi-dimensional analysis techniques to the processing of event data from large spectrometers", presented at: Meeting on Programming and Mathematical Methods for


53. F.H. Summer, private communication.

54. A. Fucci, private communication.


56. F.H. Summer, "The architecture of the MU5", these proceedings, pp. 65


58. W. Jank, "ERASME, a bubble chamber film measuring system", these Proceedings, pp. 367
FUTURE DEVELOPMENTS IN COMPUTER ARCHITECTURE AS A
RESULT OF IMPACT OF THE MICROCOMPUTER TECHNOLOGY

L. Monrad-Krohn,

1. GENERAL REMARKS

Predicting the future is always difficult and it is specially difficult when there is turbulence and rapid development. It seems, however, that the times of rapid development has gone from the computer science and that the task of predicting the future will become a question of analyzing very carefully our past, recognizing the significant trends (which may be the very difficult part of it) and project the significant trends into the future.

Now, what are the significant trends in the field of computing today? Circumventing that question for a moment, we can ask another question: Where in the field of computing today do we have the most rapid development?

The most rapid development cannot be in the field of languages, where we have seen that the latest introduced languages, like PL/1 and APL yet has failed to gain the universal acceptance as was predicted. We even have seen that the trend has reversed itself in the business processing's use of RPG, which has recently become more popular with the smaller minicomputers used for business applications. We have the heroic work behind ALGOL 68, another development which has certainly failed to achieve the same development speed as the equivalent language ALGOL introduced in 1958 or -60. If there is any development in the field of software today the development is certainly proceeding at a much slower pace than ten years ago and the prospects of a new language winning universal acceptance by virtue of its cleverness, good design or outstanding capabilities seems to be very slim indeed. So, predicting the future in the field of languages seems today to be a far easier task than it was ten years ago.

Turning to computer hardware and computer architecture we can similarly ask ourselves a question of how the speed of development compares today with the development speed of yesterday, and I believe we grossly will find

x) DATA INDUSTRI, Pilestredet 75 c, OSLO 3, Norway. Tel.: (02) 69 49 95
the almost equivalent answer. The concepts of von Neumann are still with us, apart from the realization of virtual memories and virtual machines. The last ten years of hardware development has seen only two real significant innovations:

a. The introduction of a minicomputer, and
b. The emergence of the microcomputer

The minicomputer revolution did nothing with the basic concepts of computing, but it had to turn around a lot of old conceptions about the economics of using computers and the conceptions of where it would be economically justifiable to use a computer.

Today the revolution of microcomputers will do the same thing on a different scale in numbers, and on a different scale in economics. The main concepts of computer architecture has barely been changed. The impact of the microcomputer generally will be felt by the enormous number of computers for special applications that will be used, their low costs and their inclusion into other apparatuses as a component in much the same way a relay was used in the former days' electrical systems. The complexity of the component, of course, has increased by a factor of 10,000.

So, today we are most likely to find the technical and economical adventure in computing centered around the microcomputers.

One of the impacts of the microcomputers will be on the larger systems architecture. This impact will, however, not be the most significant one, but it may, nevertheless, be interesting to examine because it is the first possibility for any radical step away from the conventional architecture of computing systems that has been currently used over the last ten years.

2. MICROCOMPUTER TECHNOLOGY

The microcomputer technology started with the invention of the transistor and the general discovery of the phenomena of moving of electrons and holes in semiconductors. The first transistors were point contact transistors whereas
the next generation based on germanium was manufactured through a rather
complexed crystal drawing technique with impurities controlled in the molten
form of the germanium material. With the introduction of the gas diffusion
 technique into silicon, which gave rise to the so called planar silicon tran-
sistor, the technology came that should ultimately enable the silicon pro-
cessors to manufacture on one silicon chip a large number of transistors and
resistive components. The invention of the metal oxyde transistor - MOS
transistor - gave the tool that should make it possible still to push to a new
height the number of components manufactured on one single silicon chip.
The silicon processing industry now has increased in number of components
on a single chip by a factor of 1,000 over the last nine years. Of these chips
several hundreds may be had out of one silicon wafer and up to hundred wa-
fers may be processed in the same batch. Thus the modern silicon chip is a
mass produced product, economical only if it can be produced in large
quantities, between 100,000 units and several millions. There is a high
degree of developmental engineering overhead in the design process and with
 corresponding low prices for mass produced items. There can hardly be
found in the world today an industrial process with such dramatic increase in
productivity.

3. SEGMENTATION OF SYSTEMS. MINIATURIZATION AND THE
  CONNECTOR LEADS PROBLEM.

Parallel with the dramatic increase in the silicon processing industries
productivity there has been a certain work going on, specially for military
purposes, in miniaturization of electronic systems. As the components of the
system become smaller, two boundaries to miniaturization raised themselves:

a. Heat dissipation problems
b. Connector lead problems

As the components become smaller and are condensed in a more tightly
packed case, the power dissipation per inch$^3$ is rising, and there is a serious
problem of getting rid of the dissipated heat. Also as the components become
smaller, the size of the wiring becomes a larger percentage of the total
volume. So, to minimize the volume taken up by the passive and unproductive
component that the interconnecting lead constitutes, there has been a constant attention brought to the question of how to segment larger electronic systems into smaller parts that would have few interconnecting leads. These parts could then be miniaturized as separate building blocks.

Now, considering the computer and the Central Processing Unit, the heavy set of interconnecting wires between logic modules have been most prominent in all computer constructions. This is because as a single unit one has never before been able to build with reasonably large printed circuits which again with yesterday's technology could not represent more than a small part of the total computer system with a high number of interconnections to other parts of the computer as a result. When suddenly, however, the possibility comes of including the Central Processing Unit in one unit as a throw-away item, the microcomputer architect finds that he now has a tool to concentrate within one chip all the necessary logic to, if necessary, multiplex the interconnecting wires leading out of the chip and to include as much logic on the chip that he makes sure that he does not cut across any boundary inside the computing system that will require a great number of leads. Thus, the complete computer CPU on a chip makes a very logical unit that can be built with very few wires interconnecting to the rest of the world.

4. MICROCOMPUTERS

In view of the foregoing, it is clear that microcomputers is a result of the silicon processing industries success in combining a large number of components on a single silicon chip with few interconnecting leads to the outside. Thus, microcomputers have nothing to do with microprogramming, with small computers built by conventional components, nor with any special gadget with a special low computing performance. Not everywhere are these terms quite clear, but for a fruitful discussion it is necessary to make this distinction. One could admit into the class of microcomputers for discussion purposes also the Central Processing Unit that were built up by a limited number of large-scale integration silicon chips, but the dramatics of the development based on this concept is not as interesting. The microcomputer component is associated with low cost. We can expect that the reasonably powerful microcomputer CPU will be produced by the millions at a price
below $20 if enough different uses can be found.

With a background of the silicon processing industries skills, the microcomputers first started as a special purpose chip to be manufactured for a Japanese company for use in a calculator, where the calculator function was to be realized by programming of a four-bit general purpose CPU. Thus, we have the zero generation of microcomputers which should rather be called calculator chips, designed to realize decimal arithmetic in extended calculator functions.

The first generation of true microcomputers came when a terminal manufacturer asked one of the semiconductor houses for a special design which was to be a general programmable 8-bit CPU with a poor, but still decent instruction repertoire. As the history goes (three years ago) the semiconductor house did not quite meet the speed requirements of the terminal manufacturer, but found quite an interest in the product on its own. Thus, we can define the first generation microcomputers as slow general purpose CPUs based on 4- or 8-bit parallel arithmetic and with an instruction repertoire not quite up to the level of the smallest minicomputers.

The second generation of microcomputers was announced in 1973 and came available early 1974. This product has an instruction repertoire comparable to a small minicomputer and a speed somewhere around 1/3 of the conventional minicomputer speed. The second generation microcomputer is based on the so called N-channel MOS-technology.

Some manufacturers now claim to have introduced the third generation microcomputers, which are manufactured with bipolar technique or with silicon on sapphire (SOS) technique. These microcomputers are the full fledged competition to the lower end of the minicomputers and are now generally recognized as a product in two versions:

a. Chip version manufactured by the semiconductor houses,
b. Card version manufactured and sold by the minicomputer manufacturers or by a range of new companies that are formed to make the most out of this new technology (R2E in France and DATA INDUSTRI in Norway as
well as several others in Europe and United States).

5. IMPACT ON COMPUTER ARCHITECTURE

When it comes to the conventional systems today of reasonable size, the microcomputer will have negligible impact. Even on today's machines the Central Processing Unit cost constitutes such a small part of the total system cost that even if the cost of the CPU declined to zero, the overall system cost will not be dramatically influenced. The designer of the conventional systems of today's Central Processing Units will, however, ask themselves questions how can they today utilize the new component that the microcomputer constitutes. He will not be under great pressure to use it, as nothing is more disturbing to the established computer manufacturer than a new change in technology that will make their production method, training courses, maintenance schemes etc. obsolete. For those that are not inhibited by such commercial factors, the application of microcomputer technology to computer architecture is a very interesting subject. Reviewing the obvious facts that will govern the application of microcomputers we find:

a. The CPU unit can be regarded as a low cost item, thus the load factors on the CPU can be disregarded.

b. The constant search for speed in main computer CPUs will drive us to look for parallel processing.

The economic drive behind the microcomputer and its demand to be used in parallel in great quantities in conventional CPUs is a strong motivator for finding new concepts in programming languages to be able to express a maximum of parallelity in problem solving algorithms. Up till now most computers has had but one single "mind" where problems had to be expressed in a sequencial fashion. Now suddenly we find ourselves faced with the opposite task – How to desequencialize the events that take place and express this in a suitable language form. This is probably the greatest challenge today to the language field within computing and it is in this field that we might see the next steps forward within the field of computing.

Looking at the conventional CPU structure today, however, we find immedia-
tely a number of candidates for parallel processing:

a. Interrupt handling routines, which are routines essentially unsynchronized to the main program and constituting routines that can almost in its entirety be processed in parallel.

b. Peripheral equipment driver routine.

These facts are so obvious that within a few years we will see computer structures announced with "Separate dedicated processors for each interrupt routine and for each peripheral driver".

Another candidate for being pushed out to a separate dedicated microcomputer processor is the language compiler. New languages to a certain computer installation will not come in forms of tapes but in form of Read Only Memories with the associated microprocessor that attaches on the central computing facilities by ways of conventional I/O units of today.

6. IMPACT ON SYSTEMS DESIGN

The introduction of the microcomputer component can be compared to the introduction of the integrated circuit some nine years ago and we can put up the following list of comparisons, as we know what the introduction of the integrated circuit meant to the design of electronic systems with transistors. We can draw some of the corresponding conclusions when the systems designer has got the new building block to use. The table may look as follows:

<table>
<thead>
<tr>
<th>Main Points when using integrated circuits instead of transistors</th>
<th>Main Points when microcomputers are used as building blocks instead of integrated circuit designs</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Logic designers are eliminated</td>
<td>* Designers of CPU structures are looking for jobs</td>
</tr>
<tr>
<td>* The development of logic systems become cheaper</td>
<td>* The development of information processing systems become cheaper</td>
</tr>
<tr>
<td>* There is no observable trends towards larger systems</td>
<td>* We guess that the microcomputers will not lead to larger systems at such, but to more dedicated use of microcomputers and their processing power in scattered locations</td>
</tr>
<tr>
<td>Main Points when using integrated circuits instead of transistors</td>
<td>Main Points when microcomputers are used as building blocks instead of integrated circuit design</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Cont'd</strong></td>
<td></td>
</tr>
<tr>
<td>. Transistors designs are rapidly made obsolete</td>
<td>. We may guess that today's computer will be obsolete in a few number of years</td>
</tr>
<tr>
<td>. There was observed a strong trend towards standardization of signal levels</td>
<td>. We may observe a stronger tendency towards standardization of microprocessors instruction repertoire</td>
</tr>
<tr>
<td>. Better computer aided design tools are being developed</td>
<td>. We may see automation of information systems design</td>
</tr>
<tr>
<td>. There is a continued demand for special circuits based on transistors</td>
<td>. The conventional CPUs of today will still be used in special applications like program development, research environments and educational institutions?</td>
</tr>
</tbody>
</table>

7. CONCLUSION

The impact of microcomputers on the computing science as a whole will not be great. In the specific area of computer architecture there will be several trends that can be compared to the transition from transistor logic to integrated circuits. The emergence of the microcomputer will stir further development in language theory, as a need for expressing algorithms for parallel processing is a key point to the effective utilization of the microcomputer as a mass product.
DATA ANALYSIS TECHNIQUES FOR
HIGH ENERGY PARTICLE PHYSICS*

Jerome H. Friedman

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305

ABSTRACT

Useful techniques for the statistical analysis and presentation of high energy particle physics data are described and discussed.

1. INTRODUCTION

The purpose of this report is to acquaint high energy physicists with a variety of techniques for presenting and making statistical inferences from counted data. The attempt will be to introduce new techniques that are not commonly used in high energy particle physics as well as to place those methods that are familiar into the general framework of statistical data analysis. This report will not deal with the equally important problem of data reduction. That is, reducing the raw digitizations from particle detectors to more useful quantities such as particle momenta and angles. Although these calculations are often quite complex they seldom require statistical inference. (A notable exception is hypothesis discrimination in kinematic fitting.) The computer codes that perform these computations can usually be thought of as computing engines that transform the data from the raw experimental variables to those that are more convenient for further calculations.

This report is concerned with these further calculations; that is, how to discover properties of the particle interactions from the data, and deduce as well as present, statistically meaningful statements about those properties.

The methods discussed are general in the sense that they can be applied to data from any science that have similar properties to those encountered in particle physics. In fact, many of the techniques that are discussed, although new to particle physics, are commonly used in other sciences, especially pattern recognition and artificial intelligence. The emphasis, however, will be on those methods that can be most profitably applied to the types of data usually encountered in high energy particle physics experiments.

*Work supported by the U. S. Atomic Energy Commission
In an effort to keep this report as self-contained as possible, the first sections will deal briefly and very superficially with those concepts from probability theory and statistics that are necessary for understanding what follows. 1) The next sections will discuss various ways of analyzing and presenting univariate or one-dimensional data, that is, when only one measured aspect of the data is considered at a time. Quite often in particle physics experiments several aspects of an event are measured, and the problem is to try to understand and describe the interrelations among these quantities. This requires multivariate (or multi-dimensional) data analysis techniques where several aspects of the data can be simultaneously considered. The final sections describe some new techniques for multi-dimensional data analysis.

The emphasis throughout is on ideas and concepts rather than on specific details. When possible the procedures will be described and discussed in terms of their effect on actual or simulated data rather than with detailed analyses of their statistical properties. When needed, these properties will simply be stated and references provided where interested readers can find detailed analyses and proofs.

2. COUNTED DATA AND DENSITY ESTIMATION

Nearly all analysis on counted data centers on probability density estimation. The several measurements, \( \overline{x} \), made on the events are regarded as random variables drawn from (distributed according to) a probability density function, \( p(\overline{x}) \). If the variables can take on only discreet (rather than a continuum of) values then \( p(\overline{x}) \) is referred to simply as a probability distribution. There are several definitions of probability and probability density but the most intuitive is the frequency ratio definition

\[
\lim_{n_i, N \to \infty} \frac{n_i}{N} = \int_{r_i} p(\overline{x}) d\overline{x}.
\] (1)

Here \( n_i \) is the number of counts appearing in a sub-volume, \( r_i \), (cell) of the measurement variables and \( N \) is the total number of counts recorded. Constructing \( r_i \) as a little sphere about some point \( \overline{x} \) and letting the volume approach zero as \( n_i \) and \( N \) approach infinity, one can define the notion of the probability density, \( p(\overline{x}) \), at \( \overline{x} \). Obvious properties of the probability density are

\[
\int_{R} p(\overline{x}) d\overline{x} = 1
\] (2)
and \[ p(\vec{x}) \geq 0 \quad \text{for all} \quad \vec{x} \in \mathbb{R} \]

where \( \mathbb{R} \) is the total region of measurement space.

It is clear that \( p(\vec{x}) \) contains all of the information of the experiment. The purpose of experimentation is to infer properties of \( p(\vec{x}) \) from the observed distributions of the measured counts. Conversely, it is the purpose of theory to calculate \( p(\vec{x}) \) from mathematical models and infer from it the results of experiments.

Data analysis is divided into two types, **parametric** and **non-parametric**. In parametric (or model dependent) analysis, \( p(\vec{x}) \) is assumed to be a member of a parameterized family of distributions

\[ p(\vec{x}) = p(\vec{a} ; \vec{x}) , \tag{2a} \]

where \( \vec{a} \) is the set of parameters (either discreet or continuous or both) that specify the particular distribution from the family of possible distributions. The problem of determination of the probability density function then becomes the problem of determining the appropriate values for the parameters \( \vec{a} \). The particular parameterized family can come from the researchers intuition, invariance principles (such as angular momentum conservation) or specific dynamical models. For example, the Lorentz invariant amplitude squared for a reaction is the probability density in the Lorentz invariant phase space.

In non-parametric (model independent) analysis no a priori information is assumed about the probability density function. In this case one infers the probability density function directly from the counted data, with very little or no information about what form it might take. Histogramming is an example of a non-parametric (one-dimensional) density estimation.

There are relative advantages and disadvantages to both types of analysis. When it is properly applied parametric analysis is usually statistically much more powerful than non-parametric analysis. This is due to the tremendous increase of information in restricting the set of all possible probability densities to those of a particular parameterized family. The results of the analysis, however, crucially depend upon the correctness of this assumption. If the probability density function that gives rise to the data is not a member of the supposed parameterized family, then at best the statistical power is reduced compared to non-parametric techniques, and at worst (usually the case) the results are meaningless. Non-parametric techniques have the advantage of being applicable to a wide range of problems since they require few assumptions concerning
the data. It should be kept in mind, however, that even though non-parametric techniques are usually formulated independently of specific probability densities their statistical performance usually varies with the actual probability density of the data.

Statistical theory is far more developed for parametric analysis than non-parametric. This is especially true for the family of normal or Gaussian distributions

\[
p(\mathbf{\mu}, \Sigma; \mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp \left[- \frac{1}{2} (\mathbf{x} - \mathbf{\mu})^T \Sigma^{-1} (\mathbf{x} - \mathbf{\mu}) \right]
\]

where the parameters are the location vector \(\mathbf{\mu}\) and covariance matrix \(\Sigma\). A great many of the statistical techniques in common use were designed to be optimal for normal distributions and are referred to as normal theory techniques. These techniques can lose considerable statistical power when applied to data with non-normal density distributions.

3. A MINI-INTRODUCTION TO ESTIMATION THEORY

This section introduces the few necessary concepts in Statistics that are required to understand the sections that follow. As noted above, the set of measurements \(\{\mathbf{x}_i\}_{i=1}^N\) comprising an experiment can be thought of as random variables drawn from a probability density function \(p(\mathbf{x})\). The purpose of data analysis is to make inferences concerning \(p(\mathbf{x})\). In parametric analysis one usually wishes to infer likely possible values for the parameters. In non-parametric analysis the density itself is to be inferred. This process of statistical inference is called estimation. Particle physicists quite often (incorrectly) use the terms "measurement" or "determination" for statistical estimation.

Consider a parametric example. Suppose that the set of measurements \(\{\mathbf{x}_i\}_{i=1}^N\) are known to be distributed according to \(p(a; \mathbf{x})\) for some (unknown) value of \(a\). The desire is to estimate the parameter, \(a\), from the values of the measured random variables \(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\).

Any function of a set of random variables

\[
Y = \phi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)
\]

is itself a random variable with a probability density function \(p_N(a; Y)\) that can (at least in principle) be calculated from \(p(a; \mathbf{x})\). If one is sufficiently clever in choosing the function, \(\phi\), then \(p_N(a; Y)\) might be large only for those values of \(Y\) near \(Y = a\). That is, for any set of possible values for \(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\) drawn
from \( p(a, \bar{x}) \), \( \varphi(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N) \) always has a value near \( a \). Thus, for the particular set of values of \( \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N \) we happen to measure, the value of
\[
\bar{Y} = \varphi(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N)
\]
will be a good approximation to the value of \( a \). The function
\( \varphi(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N) \) is called a statistic and its value for a particular set of
\( \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N \) is called an estimate of \( a \).

Consider the following example of how one might construct a statistic for performing an estimation. The integral
\[
I(a) = \int_R f(\bar{x}) p(a; \bar{x}) d\bar{x}
\]
is a function of the parameter, \( a \). Here \( R \) is the region of all possible values for the measurements \( \bar{x} \). This integral is called the expected value, \( E[f] \), (or sometimes the average or mean value) of the arbitrary function, \( f(\bar{x}) \), with respect to \( \bar{x} \). If the integrand is integrable then the explicit functional form of \( I(a) \) can be calculated. From the central limit theorem (law of large numbers), one has the result that for sufficiently large \( N \),
\[
\bar{Y} = \varphi(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N) = \frac{1}{N} \sum_{i=1}^{N} f(\bar{x}_i)
\]
has a normal probability density function
\[
p_N(a; \bar{Y}) = \frac{1}{\sqrt{2\pi} \sigma_N} e^{-1/2 (\bar{Y} - a/\sigma_N)^2}
\]
where
\[
\sigma_N = \left( \frac{1}{N} \int_R (f(\bar{x}) - E[f])^2 p(a; \bar{x}) d\bar{x} \right)^{1/2}
\]
The integral is called the variance, \( V[f] \), of the function \( f(\bar{x}) \). Thus,
\[
\sigma_N = \sqrt{V[f]/N}
\]
As \( N \) increases, \( \sigma_N \) decreases as \( \sigma_N \sim 1/\sqrt{N} \). Thus, for sufficiently large \( N \), \( p_N(a; \bar{Y}) \) will be a very narrow distribution centered at \( \bar{Y} = a \). The quantity
\[
\frac{1}{N} \sum_{i=1}^{N} f(\bar{x}_i)
\]
is called the sample mean of the function \( f(\bar{x}) \). The central limit theorem tells us that the sample mean is a good statistic for estimating \( I(a) \) [expected value of \( f(\bar{x}) \)] for sufficiently large \( N \). One can then take as an
estimate, \( \hat{a} \), for the value of the parameter, \( a \),
\[
\hat{a} = \Gamma^{-1}\left[ \varphi(x_1, x_2, \ldots, x_N) \right] = \Gamma^{-1}\left[ \frac{1}{N} \sum_{i=1}^{N} f(x_i) \right].
\] (10)

Since the function, \( f(x) \), was somewhat arbitrary it is clear that this procedure can be used to construct a variety of statistics for estimating the parameter, \( a \). However, some will be better than others. For example \( \sigma_N \), which regulates the precision with which the parameter, \( a \), is estimated, depends on \( f(x) \) (for \( N < \infty \)) through Eq. 8.

The field of Statistics is concerned to a great degree with finding good statistics for estimation and determining their properties. Statistics used for estimation (usually called estimators) are rated in terms of four basic properties of their probability density distributions \( p_N(a; Y) \); these are consistency, efficiency, bias, and robustness.

3.1 Consistency

An estimator, \( Y = \varphi(x_1, x_2, \ldots, x_N) \), is consistent if the following condition holds
\[
\lim_{N \to \infty} p_N(a; Y) = \delta(Y - a).
\] (11)

That is as the number of samples gets arbitrarily large, \( p(a; Y) \) becomes an arbitrarily narrow function of \( Y \) about \( a \), and the estimator provides an arbitrarily precise estimate of the parameter, \( a \). Note that Eqs. 7 and 8 show that the estimator defined by Eq. 6 is consistent. Consistency is nearly always required for an estimator to be considered useful.

3.2 Efficiency

Consistency is concerned with the precision of the estimator for infinite sample size. (In the field of Statistics, a result that holds in the limit of infinite sample sizes is called an asymptotic result.) Efficiency is concerned with the precision of the estimator for finite sample size \( N \). An estimator is called efficient if the variance of its probability density function
\[
V_N = \int_{\mathbb{R}} (Y - a)^2 p_N(a; Y) dY
\] (12)
is as small as possible\(^3\) for a given \( N \). The square root of the variance, \( \sigma_N = \sqrt{V_N} \), is characteristic of the width of \( p_N(a; Y) \) about \( a \), and thus is directly related to the precision of the estimator. Therefore, an efficient estimator for
a given $N$ is one that (loosely speaking) has maximal precision. The relative efficiency between two estimators is the inverse ratio of the variances of their probability densities for a given sample size, $N$. The efficiency of an estimator is its relative efficiency to an efficient estimator (i.e., efficient estimators are said to have 100% efficiency).

This definition of efficiency can be related to the intuitive meaning of the word in the following manner. For large sample size, $N$, the variance of most estimators decreases as, $V_N \sim 1/N$, for increasing $N$ (i.e., $\sigma_N \sim 1/\sqrt{N}$). Then the efficiency of an estimator is the inverse ratio of the number of samples (events) it requires to the number an efficient estimator requires for the same precision. Clearly high efficiency is a desirable property for an estimator. However, an estimator with the highest efficiency is quite often not the most desirable. Sometimes the computational complexity of the most efficient estimator makes it more expensive for a given precision than a less efficient estimator even though the less efficient estimator requires more events.

3.3 Bias

Like efficiency, bias refers to a property of estimators for finite sample size, $N$. Specifically the bias of an estimator is defined as

$$b_N = \int_{R} Y \, p_N(a;Y) \, dY - a$$

(13a)

i.e.,


(13b)

A biased estimator is one with an expected value that is different from the true value of the parameter being estimated. The bias is just the difference between $E_N [Y]$ and the true value of the parameter.

Note that, although it might appear to be contradictory, a biased estimator can also be consistent and conversely an unbiased estimator can be inconsistent. If a biased estimator is consistent, then from Eq. 11

$$\lim_{N \to \infty} b_N = 0 .$$

It may at first seem that bias would be a very undesirable property for an estimator to have. This is generally not the case. It is only important that the bias be relatively small compared to the square root of the variance (Eq. 12) (standard deviation) of the probability density function. Most of the commonly used estimators in particle physics are in fact biased. There are various techniques for reducing bias in estimators but they usually do this at the expense
of precision (increasing the variance). Thus, one is forced to compromise between degree of bias and precision.

3.4 Robustness

The concept of robustness is strongly tied to the notions of non-parametric data analysis. As pointed out above parametric techniques are usually more efficient than non-parametric techniques so long as the actual probability density of the data is in fact a member of the supposed parameterized family of density distributions. If this assumption is not quite correct then the parametric estimator loses efficiency. However, some estimators lose efficiency more rapidly than others as nature deviates from the experimenter's assumptions. Estimators that maintain reasonable efficiency over a wide range of data probability densities are called robust estimators.

Robustness is an often under-rated quality of estimators. Both physicists and statisticians usually seek maximum efficiency at the expense of robustness. This effort can often be misguided since if the data deviate from the a priori assumptions then the supposed most efficient estimator can, in fact, have poor efficiency. Even when the theory from which the parameterized density function is constructed is on solid ground, measurement errors on the data points can cause the data to deviate from the parameterized family of densities.

Most highly efficient parametric estimators gain most of their information from the low density regions (tails) of the distribution. Thus, if only a few data points in these tails deviate from the parameterized probability density function the estimate will be severely effected. Physicists usually refer to this phenomenon as the "tail wagging the dog". What has actually happened is that their estimator was very non-robust. The least squares estimator, which is one of the most popular in particle physics, is a good example of an extremely non-robust estimator. Generally, order statistics such as medians, and percentiles are much more robust than arithmetic statistics such as means and standard deviations.

For example if one wished to estimate the width of a symmetric distribution he could calculate its standard deviation about the mean or he could take half the difference between its 32 and 68 percentiles. The former would be more efficient if the data had exactly a normal distribution. However, if just one of the data points near the edges of the distribution was mismeasured so that it was somewhat farther from the center than it should be, the standard deviation estimate will be severely effected. This is because the standard deviation estimate
weights each point by the square of its distance from the center. The percentile estimate on the other hand will be completely unaffected by the mismeasured point.

For exploratory data analysis especially, robustness is essential. Robust estimators generally maintain from 60% to 90% efficiencies over wide ranges of data distributions while non-robust estimators tend to have near 100% efficiency when the data distribution exactly follows the predicted probability density function, and low efficiency when it does not.

4. ANALYSIS AND REPRESENTATION OF ONE-DIMENSIONAL DATA

With the preliminaries of the preceding section out of the way, we are ready to discuss and evaluate various techniques for analyzing and presenting data. We will start with univariate or one-dimensional data analysis. That is when only one measured quantity is considered at a time. We will discuss multivariate analysis in the following sections. Univariate analysis techniques are far more developed than corresponding multivariate techniques. This is especially true for non-parametric methods. There are many large text books devoted to statistical techniques for univariate analysis. Thus, there will be no attempt in this brief report for completeness. The purpose will be to introduce some techniques not commonly known to high energy particle physicists that could be valuable tools for analyzing particle physics data, and to relate them to the more commonly used techniques.

4.1 Non-Parametric Univariate Density Estimation

Let \( \{x_i\}_{i=1}^{N} \) be a sequence of independent identically distributed random variables with some unknown probability density function \( p(x) \). We wish to construct estimators \( \hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \delta(x - x_i) \) for \( p(x) \) that depend only on the observations, \( \{x_i\}_{i=1}^{N} \).

4.1.1 The Histogram Approach

Histogramming is the most commonly used method in particle physics. In this method the real line is divided into \( M \) regions, \( r_i \) (bins, channels) and \( \hat{p}(x) \) is taken to be constant over each region \( r_i \):

\[
\hat{p}(x) = \hat{p}_i \quad \text{if} \quad x \in r_i, \quad i = 1, M.
\]

Let \( g_i(x) \) be an indicator function for each region, i.e.,

\[
g_i(x) = \begin{cases} 
1 & \text{if } x \in r_i \\
0 & \text{otherwise}.
\end{cases}
\]
Then we have for our estimator of \( p(x) \),

\[
\hat{p}_N(x) = \frac{1}{N} \sum_{i=1}^{M} \sum_{j=1}^{N} g_i(x_j) g_1(x).
\]  

(14)

From the central limit theorem one has

\[
E[\hat{p}_1] = \bar{p}_1 = \int_{\mathcal{R}_1} p(x) \, dx
\]

(15)

and

\[
p_N(\hat{p}_1) = \frac{1}{\sqrt{2\pi} \sigma_i} e^{-\frac{(\hat{p}_1 - \bar{p}_1)^2}{2 \sigma_i^2}}
\]

(16)

where \( \sigma_i = \sigma_0 / \sqrt{N} \),

when \( \bar{n}_i \equiv N \bar{p}_i \) is large. A more careful analysis shows that for any \( \bar{n}_i \), the \( n_i = N \bar{p}_i \) are distributed according to a multinomial distribution

\[
p_N(\hat{n}_1, \hat{n}_2, \ldots, \hat{n}_M) = N! \prod_{i=1}^{M} \frac{n_i}{\hat{n}_i!}
\]

if the total number of events, \( N \), is considered fixed. 4) Note from Eq. 17

\[
E[\hat{n}_i] = \bar{n}_i
\]

(18)

so that the estimator is unbiased. The variance of \( \hat{n}_i \) is

\[
V[\hat{n}_i] = N \bar{p}_1 (1 - \bar{p}_1)
\]

(19a)

so that

\[
V[\hat{p}_1] = \frac{\bar{p}_1 (1 - \bar{p}_1)}{N}
\]

(19b)

Equation 19 shows that \( \hat{p}_1 \) is a consistent estimator of \( \bar{p}_1 \). For \( \bar{p}_1 \ll 1 \) (large number of bins for example) Eq. 19a can be approximated by

\[
V[\hat{n}_i] \approx N \bar{p}_1 \equiv \bar{n}_i
\]

(20)

Since \( \bar{n}_i \) is usually not known it seems reasonable to make the further approximation

\[
\bar{n}_i \approx \hat{n}_i
\]

(21)
in Eq. 20, in which case

\[ V[\hat{n}_1] \approx \hat{n}_1 \]

or

\[ \sigma[\hat{n}_1] \equiv \sqrt{V[\hat{n}_1]} = \sqrt{\hat{n}_1} . \] (22)

These approximations, then, yield the common "rule of thumb" result that the statistical uncertainty in the number of counts in a histogram bin is equal to the square root of the number of counts.

It can easily be shown that Eq. 17 can be approximated by Eq. 16 for large \( \bar{n}_1 \), with \( \sigma_{\hat{n}_1}^2 \) given by Eq. 19b.

Although \( \hat{p}_1 \) is a consistent estimator of \( \bar{p}_1 \), \( \hat{p}_N(\chi) \) from Eq. 14 is not a consistent estimator of \( p(\chi) \) (unless by some chance \( p(\chi) \) is exactly piece-wise constant over the chosen bins). As the total number of counts tends to infinity, the average variance (mean squared error) of the density estimation

\[ V_N = E \left[ \int_R [p(\chi) - \hat{p}_N(\chi)]^2 \, d\chi \right] \] (23)
does not approach zero.

There are other shortcomings of the histogram approach. The choice of the binned intervals, \( r_i \), and their number, \( M \), is arbitrary. There are no general guidelines for optimum binning except looking at the result and rebinning. Also, if constant bin sizes are used many bins may have too few counts while only a few of the others may contain nearly all of the counts rendering useful density estimation impossible. Histogramming also fails to use to advantage any continuity properties of \( p(\chi) \). Since the estimates in neighboring bins are independent of each other there will be sharp discontinuities in \( \hat{p}_N(\chi) \), Eq. 14, at the bin boundaries.

These discontinuities (usually termed "statistical fluctuations" by physicists) are normally the result of the variance of the estimation in each separate bin and do not represent actual structure in \( p(\chi) \). Most probability densities \( p(\chi) \), are reasonably continuous, and using this information can considerably reduce the variance, \( V_N \), (Eq. 23) of the density estimation.

4.1.2 The Orthogonal Function Approach

After histogramming, the most common density estimators used by high energy physicists are orthogonal functions. Let \( \{ \psi_i(\chi) \}_{i=1}^M \) be a set of
orthogonal functions defined on the real line

\[ \int_{\mathbb{R}} \phi_i(x) \phi_j(x) \, dx = \delta_{ij} \]

and we wish to estimate \( p(x) \) from the data points \( \{x_j\}_{j=1}^N \) with an estimator of the form

\[ \hat{p}_N(x) = \sum_{i=1}^M \hat{c}_i(N) \phi_i(x) . \] (24)

If the actual probability density function, \( p(x) \), were known then it is easy to show that the variance of the density estimation, \( V_N \), (Eq. 23) is minimal for

\[ \hat{c}_i^* = \int_{\mathbb{R}} \phi_i(x) p(x) \, dx = E[\psi_i] . \] (25)

For non-parametric estimation \( p(x) \) is not known so we estimate the integral from the data sample

\[ \hat{c}_i(N) = \frac{1}{N} \sum_{j=1}^N \psi_i(x_j) . \] (26)

From the central limit theorem one has (for large \( N \))

\[ p_N\left[ \hat{c}_i(N) \right] = \frac{1}{\sqrt{2\pi} \sigma_N} \, e^{-1/2} \frac{(\hat{c}_i(N) - \hat{c}_i^*)^2}{(\sigma_N)^2} \] (27)

where

\[ \sigma_N^2 = V(\psi)/N = E[(\psi - E[\psi])^2]/N . \]

Thus, \( E[\hat{c}_i(N)] = \hat{c}_i^* \) so that the estimate is unbiased and \( \lim_{N \to \infty} \sigma_N^2 = 0 \) so that it is consistent.

Combining Eqs. 24 and 26 we have for our density estimate

\[ \hat{p}_N(x) = \frac{1}{N} \sum_{i=1}^M \sum_{j=1}^N \phi_i(x_j) \phi_i(x) . \] (28)

The average variance of the estimate, \( V_N \), Eq. 23, is

\[ V_N = E\left[ \int_{\mathbb{R}} (p - \hat{p}_N)^2 \, dx \right] = E\left[ \int_{\mathbb{R}} (p - \hat{p}_N)^2 \, dx \right] + E\left[ \int_{\mathbb{R}} (p - \hat{p}_N)^2 \, dx \right] \] (29)
where
\[ \hat{p}(x) = \sum_{i=1}^{M} \hat{c}_i^* \psi_i(x). \]  

(30)

The first term on the right hand side of Eq. 29 is a constant independent of the data so that
\[ V_N = \int_R \hat{p}^2(x) \, dx - \int_R \hat{p}_N^2(x) \, dx + E \left[ \int_R \left( \hat{p}(x) - \hat{p}_N(x) \right)^2 \, dx \right] \]
or
\[ V_N = \int_R p^2(x) \, dx - \int_R \hat{p}_N^2(x) \, dx + \sum_{i=1}^{M} (\sigma_{(i)}^2). \]  

(31)

Equation 31 shows that the variance of the estimate is composed of a constant systematic part and a statistical part that approaches zero as \( N \) becomes infinite. Thus, like the histogramming approach, the orthogonal function density estimator is inconsistent (unless by some chance \( p(x) = \hat{p}(x) \) for all \( x \) — i.e., either \( M = \infty \), or for finite \( M \), \( p(x) \) can exactly be expressed by Eq. 30).

It is no accident that the histogramming and orthogonal function estimators share this property of inconsistency. Inspecting Eq. 14, one sees that it is just a special case of Eq. 28 where the orthogonal functions are the indicator functions \( g_i(x) \). Note that
\[ \int_R g_i(x) g_j(x) \, dx = \delta_{ij} \]
and
\[ \hat{C}_i^{(N)} = \frac{1}{N} \sum_{j=1}^{N} g_i(x_j) = \frac{n_i}{N}. \]

The general orthogonal function approach suffers from generalized analogs of most of the problems discussed for histogramming. The problem of specific bin choice and number of bins becomes the problem of number and specific choice of the orthogonal functions, \( \{ \psi_i(x) \}_{i=1}^{M} \). Also, it may happen that \( \hat{p}_N(x) \) is negative for some value of \( x \) rendering it inadmissible as a probability density function (although it still may be quite useful).

4.1.3 The Rosenblatt Estimator

We will now begin to consider some consistent estimators of univariate probability density. The first is the Rosenblatt or "naive pdf" (probability density
function) estimator. This estimator is defined as
\[
\hat{p}_N(x) = \frac{1}{2h} \left[ \hat{P}_N(x+h) - \hat{P}_N(x-h) \right].
\] (32)

Here \( \hat{P}_N(x) \) is the empirical cumulative distribution of the data points defined as
\[
\hat{P}_N(x) = \begin{cases} 
0 & x < x_1 \\
\frac{i}{N} & x_1 \leq x < x_{i+1} \\
1 & x \geq x_N
\end{cases} \quad (32a)
\]

where the points are labeled in increasing order of \( x \). From the central limit theorem one has
\[
\lim_{N \to \infty} \hat{P}_N(x) = P(x) \equiv \int_{-\infty}^{x} p(x) \, dx.
\] (33)

The quantity \( P(x) \), defined in Eq. 33, is called the cumulative distribution of \( p(x) \). From its definition (Eq. 32), one sees that this estimate, \( \hat{P}_N(x) \), is just the fraction of counts that lie in a window of width \( 2h \), centered at \( x \), divided by the window width. If we define an indicator function
\[
\psi(x;x') = \begin{cases} 
1, & \text{if } x - h \leq x' < x + h \\
0, & \text{otherwise}
\end{cases}
\] (34)

then the probability density estimate can be written as
\[
\hat{p}_N(x) = \frac{1}{2hN} \sum_{i=1}^{N} \psi(x;x_i).
\] (35)

Rosenblatt shows that for all \( N \) (not just for \( N \to \infty \))
\[
E[\hat{p}_N(x)] = \frac{1}{2h} \left[ P(x+h) - P(x-h) \right].
\] (36)

Expanding this in a Taylor series
\[
E[\hat{p}_N(x)] = p(x) + \frac{h^2}{6} p''(x) + 0(h^4).
\] (37)

This result shows that the estimate is biased with the bias approaching zero, quadratically, as the window size \( h \) approaches zero.

Rosenblatt also calculates the variance of the estimate as
\[
E \left[ (\hat{p}_N(x) - p(x))^2 \right] = \frac{p(x)}{2hN} + \frac{h^4}{36} \left[ p''(x) \right]^2 + 0 \left( \frac{1}{hn} + h^4 \right).
\] (38)
This estimator can be made consistent so long as \( h \) tends to zero, while the product \( (hN) \) approaches infinity. Parameterizing the window size as
\[
h = CN^\alpha
\]  
and choosing \( \alpha \) so as to minimize the dominant terms in Eq. 38, one obtains \( \alpha = 1/5 \) as the value that causes the variance to decrease most rapidly with increasing \( N \). A careful analysis shows that the constant should be
\[
C = \left[ 9 p(x)/2 |p''(x)|^2 \right]^{1/5}. \tag{40}
\]

The bias of this estimator (Eq. 37) is easy to understand. For finite window size the estimator \( \hat{p}_N(x) \) (Eq. 32) is an unbiased estimator of the average of the probability density within the window
\[
\bar{p}(x) = \frac{1}{2h} \int_{x-h}^{x+h} p(x')dx'. \tag{41}
\]
If \( p(x') \) is nonlinear within the window region, then this average will be different than the value of the probability density at the center of the window, \( p(x) \). As the window size approaches zero, or as the probability density approaches linearity, this effect will disappear, as reflected by Eq. 37.

The expression for the variance (Eq. 38) shows that like histogramming, the variance of this estimate is proportional to the value of the probability density (standard deviation proportional to the square root of the probability density). Unlike histogramming, however, this probability density estimate is not piecewise constant over fixed intervals (bins) and does not suffer from the sharp discontinuities that histogramming produces at the boundaries of these intervals ("statistical fluctuations"). This estimator does, of course, suffer from statistical uncertainty as reflected by its variance (Eq. 38). However, the Rosenblatt estimator produces a relatively smooth probability density estimate which (at least in the limit of large sample size) can be shown to be more accurate than histogramming (see below for finite sample comparisons).

4.1.4 Parzen Estimators

The Rosenblatt estimator is a special case of a general class of density estimators known as Parzen estimators or Parzen windows. 6) Let \( K(y) \) be a bounded absolutely integrable function such that
\[
\int_{\mathbb{R}} K(y) \, dy = 1 \quad \text{and} \quad \lim_{|y| \to \infty} |yK(y)| = 0. \tag{42}
\]
Then the Parzen window estimators are defined as

\[
P_N(x) = \frac{1}{h(N)} \sum_{i=1}^{N} K\left(\frac{x-x_i}{h(N)}\right).
\]  

(43)

The function \(K(y)\) is called the kernel or window function. The notation \(h(N)\) is used to explicitly indicate that the scale parameter for the kernel function depends upon the sample size, \(N\). For the Rosenblatt estimator one has

\[
K\left[\frac{x-x_i}{h(N)}\right] = \frac{\psi(x;x_i)}{2N}
\]  

(44)

where \(\psi(x;x_i)\) is defined in Eq. 34. Other possible kernels are: a) the double exponential function \(e^{-|y|}\); b) the standard normal (Gaussian) function; c) the Cauchy function \(1/(1+y^2)\); and d) \(\sin^2 y/y^2\). Using procedures analogous to those for the Rosenblatt estimator, one can show that these estimators are biased, with the bias tending to zero quadratically as the scale parameter \(h(N)\) approaches zero. Also, the variance of the estimate tends to zero as \(1/Nh(N)\) for increasing sample size, \(N\). Thus, these estimators are consistent provided that \(h(N) \to 0\) while \(Nh(N) \to \infty\).

4.1.5 \(k\)-th Nearest Neighbor Density Estimation

A disadvantage of the density estimators so far discussed is that there are few general guidelines for choosing the scale parameter (bin width for histogramming, window size, \(h(N)\), for Rosenblatt and Parzen estimators). For small variance (high statistical precision) the scale parameter should be as large as possible. For maximum sensitivity to \(p(x)\), rapid convergence as well as minimal bias (high systematic precision), the scale parameter should be as small as possible. The choice for a scale parameter is usually then a compromise between these two competing effects. Ideally, the scale parameter should depend upon the data. That is, on the basis of a density estimate the scale parameter can be changed and the density re-estimated. Although quite reasonable, this procedure invalidates the analyses that give rise to the statistical results stated above concerning the bias, consistency and variance of these estimators, since the analyses all assume that \(h(N)\) is a deterministic function independent of the data. Thus, the statistical properties of such a procedure are largely unknown. Even further, the scale parameter should probably change for different values of the variable, \(x\). In denser regions, one can take advantage of the large number of counts to increase systematic precision by using smaller
values for the scale parameter. In the sparser regions the statistical precision is relatively low so that larger values of scale parameter are in order.

The k-th nearest neighbor estimator \(^7\) allows the scale parameter \(h(N)\) to adapt to the data. Density is measured as counts per distance interval (univariate volume). For the estimators discussed so far, the interval was predetermined (by the scale parameter) and the probability content was estimated by the fraction of counts that fall in the interval. With the k-th nearest neighbor estimator, the probability content is predetermined and the interval size required to contain the probability is estimated. The estimation statistic is distance instead of number of counts. Specifically, let \(k(N)\) be a predetermined integer \((\leq N)\) and let \(h(N)\) be the distance from \(x\) to the k-th closest data point to \(x\). Thus, \(h(N)\) is a random variable depending on the data. The number of counts within an interval of width, \(2h(N)\) centered at \(x\), is \(k(N)\) by definition so that the probability density function estimate at \(x\) is

\[
\hat{p}_N(x) = \frac{k(N)}{2Nh(N)}.
\]  

(45)

It is clear that this estimator overcomes many of the disadvantages of the fixed interval estimators discussed above. The interval width, \(h(N)\), becomes narrow in regions of high counting density and wider in sparser regions, tending to stabilize the variance of the estimates.

The k-th nearest neighbor estimator is biased by the same mechanism as the Rosenblatt estimator. To second order

\[
E[\hat{p}_N(x)] = p(x) + \frac{1}{24} \left[ \frac{k(N)}{N} \right]^2 \frac{p''(x)}{p^2(x)}
\]  

(46)

so that (like the Rosenblatt estimator) the bias is proportional to the nonlinearity of the probability density function and approaches zero quadratically as \(k(N)/N\) approaches zero. Fukunaga and Hostetler\(^8\) show that the variance for this estimator is

\[
V_N(x) = \frac{p^2(x)}{k(N)} + \left[ \frac{p''(x)}{24p^2(x)} \left( \frac{k(N)}{N} \right)^2 \right]^2
\]  

(47)

(again to second order). From these equations we see that this estimator is consistent provided \(k(N)\) is chosen such that

\[
\lim_{N \to \infty} \frac{k(N)}{N} = \infty
\]  

\[
\lim_{N \to \infty} \frac{k(N)}{N} = 0
\]  

(48)
These conditions were shown by Loftsgaarden and Quesenberry\textsuperscript{9}) to be required for consistency very early and, in fact, the k-th nearest neighbor estimator is sometimes referred to as the method of Loftsgaarden and Quesenberry.

Minimizing Eq. 47 with respect to k(N), one obtains

\[
k_0(N) = \left( \frac{144 p'(x)}{|p''(x)|^2 N} \right)^{1/5} N^{4/5}
\]

so that

\[
\frac{k_0(N)}{N} = \left( \frac{144 p'(x)}{|p''(x)|^2 N} \right)^{1/5} \cdot N
\]

(49a)

(49b)

We see that the optimum number of nearest neighbors depends upon the underlying distribution. The smaller \(|p''(x)|\), the smoother the density function, and the number of nearest neighbors can be increased for greater statistical precision. For very nonlinear functions, where \(|p''(x)|\) is large, the bias dominates the precision of the estimate and a smaller number of neighbors should be chosen to reduce it.

The first term in Eq. 47 is the variance of the estimate about its mean, i.e.,

\[
E[\hat{\rho}_N^2(x)] - E^2[\hat{\rho}_N(x)] = \frac{p'(x)}{k(n)}
\]

or

\[
\sigma[\hat{\rho}_N(x)] \approx \hat{\rho}_N(x)/\sqrt{k(N)}
\]

where \(p(x)\) is approximated by \(\hat{\rho}_N(x)\). Thus, the statistical uncertainty of this density estimator is proportional to the density rather than the square root of the density, as is the case for the fixed interval estimators discussed above.

The coefficient of variation of the statistical uncertainty

\[
C = \frac{\sigma[\hat{\rho}_N(x)]}{\hat{\rho}_N(x)} = \frac{1}{\sqrt{k(N)}}
\]

is constant for the k-th nearest neighbor estimator. Thus, the fractional statistical precision in the estimate of \(p(x)\) is uniform for all \(x\), which overcomes one of the difficulties mentioned above for the fixed interval estimators.
4.1.6 Discussion

Several techniques for nonparametric one-dimensional density estimation have been presented in the previous sections and their properties discussed. It was shown that in the limit of very large data samples the Rosenblatt, Parzen, and near neighbor estimators are more accurate than the histogramming and orthogonal function approach. The near neighbor technique was shown to have the additional advantage of adaptability to the data.

In order to gain insight into the relative performance of these estimators for sample sizes and distributions commonly encountered with particle physics data, several Monte Carlo experiments were performed. A random sample of 710 data points was drawn from the probability density function

\[ p(x) = 0.5 \begin{pmatrix} \frac{2}{1+2(x-0.5)^2} + \frac{20}{1+20(x-0.2)^2} \end{pmatrix} \]  

(51)

in the interval \(0 \leq x \leq 1\). From these data points, an estimate of the probability density function, \(\hat{p}_{710}(x)\), was obtained using the histogramming approach, the Rosenblatt estimator, the Parzen estimator with the standard normal density function as a kernel, and the near neighbor technique. These estimates were then compared to the true probability density, \(p(x)\), of Eq. 51 by

\[ V_{710} = \int_0^1 [p(x) - \hat{p}_{710}(x)]^2 dx . \]  

(52)

This process was repeated nine times with different random sample points drawn from \(p(x)\) (Eq. 51), and the expected value \(E[V_{710}]\) was estimated as the average \(V_{710}\) from these nine trials. Figure 1 (a-d) shows the results. Here the \(\hat{p}_{710}(x)\), as estimated by each method, is plotted for the most "typical" data sample of the nine trials. This most "typical" trial was taken to be the one with the four values of \(V_{710}\) (from the four methods) that were closest to the averages over the nine trials. These averages were:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Average (V_{710})</th>
<th>(V_{710}) for &quot;typical&quot; Trial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>h = .012</td>
<td>.064</td>
<td>.064</td>
</tr>
<tr>
<td>Rosenblatt</td>
<td>h = .045</td>
<td>.038</td>
<td>.037</td>
</tr>
<tr>
<td>Parzen (K=Normal)</td>
<td>h = .030</td>
<td>.045</td>
<td>.045</td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>k = 120</td>
<td>.023</td>
<td>.023</td>
</tr>
</tbody>
</table>
For comparison, the true density, \( p(x) \), Eq. 51, is superimposed in each figure as a continuous line over the density estimate, \( \hat{p}_{710}(x) \).

The parameter values used in each method were those that gave the best results (minimum \( V_{710} \)) for the particular method. The histogramming method was most sensitive to particular parameter value while the k-th nearest neighbor technique was least sensitive.

The results shown in Table 1 indicate that, for this example, the consistent estimators do provide more accurate density estimation than histogramming. The variance of the most accurate estimator, the k-th nearest neighbor, is nearly three times less than that for the histogram.

Although only a single example, Figure 1 illustrates the various different properties of these estimators. The bias of the Rosenblatt and Parzen estimators is especially apparent in the center and at the right shoulder of the peak. It is in these two regions where \( p(x) \) is most nonlinear. As predicted by Eq. 37, those regions where the second derivative is large and negative (center of peak), \( \hat{p}_{710} \) underestimates \( p(x) \), whereas when it is large and positive (right shoulder), \( p_{710} \) overestimates \( p(x) \). The k-th nearest neighbor estimator is also biased from this same mechanism. However, as predicted by Eq. 46, the bias will be small in the peaked region since the bias term is proportional to \( p''(x)/p^2(x) \) and \( p(x) \) is very large in this region. The bias is larger in the right shoulder region where \( p(x) \) is not large enough to overcome the affect of \( p''(x) \).

The k-th nearest neighbor estimate is seen from Figure 1d to become poor near the boundaries. This is a general property of this estimator. When straightforwardly applied, the near neighbor estimator will always underestimate the density whenever the interval containing the k neighbors is adjacent to a boundary edge. In this case, the actual interval size is \( h(N) + B \) where B is the distance from the evaluation point to the boundary. Since this is less than \( 2h(N) \), which appears in Eq. 45, \( \hat{p}_N(x) \) will have a strong negative bias. One could try to remedy this by using the actual interval size, \( h(N) + B \), in place of \( 2h(N) \) whenever the interval contains a boundary edge. This, however, also causes the estimate to be biased with the bias being proportional to, and having the opposite sign of, \( p'(x) \) in the interval.

A good boundary strategy (and the one used in Figure 1d) is to revert to a variable interval Rosenblatt estimate. That is, whenever the k-th nearest neighborhood contains a boundary a distance \( B < h(N) \) from the evaluation point, \( x \), then the number of points, \( n_B \), in the smaller interval of width 2B centered
at \( x \), is determined, and the density is estimated as

\[
\hat{p}_B = \frac{n_B}{2NB}.
\] (53)

As seen in Figure 1d, this strategy removes most of the bias. The variance of the estimate, however, becomes relatively large for those points very close to the boundary edge.

The Parzen estimate (Figure 1c) is seen to be the smoothest, while the histogram (Figure 1a) is least smooth. The relative accuracy of the estimates, as reflected from visual inspection of Figure 1, appears to correspond to the relative values of \( E[V_{710}] \) given in Table 1. That is, the \( k \)-th nearest neighbor method gives the best density estimation, followed by (in order) the Rosenblatt, Parzen and Histogram methods.

Although the consistent estimators (especially the nearest neighbor) are generally more accurate than the histogramming approach, they are also computationally considerably more expensive. A histogram can be made with a single pass over the data sample so that the number of computations is simply proportional to the sample size \( N \). Also, the whole sample need not reside in memory at one time. The most computationally efficient method for computing the Rosenblatt, Parzen and nearest neighbor estimates is to first sort the data points. This requires computation proportional to \( N \log_2 N \). After sorting, these estimates can be calculated with computation simply proportional to \( N \). Thus, these estimators require computation proportional to \( N \log_2 N \). Also, all of the data points must be simultaneously in memory for the sorting.

The histogram's computational advantage is probably largely responsible for its popularity. Another reason is historical familiarity. Physicists usually learn by experience how to intuitively interpret histogram results accurately, although they seldom study the statistical foundations and approximations that lead to their techniques. Similar techniques and intuition can be learned just as well for the other density estimators. The most common objection to the consistent estimators, discussed above, is that the resulting estimation is relatively smooth and does not exhibit the sharp discontinuities ("statistical fluctuations") that are present in histograms. Statistically, this smoothness property is an asset, not a liability. It is this relative smoothness that makes these estimators more accurate than the histogram and renders them consistent.
estimators. (The nearest neighbor estimator gains additional accuracy, of course, from its adaptability.)

Since histogramming is the least expensive and most popular density estimator, the following sections discuss some techniques for making the histogram a more effective tool for density estimation and data presentation.

4.2 Smoothing Counted Data

For all of the density estimators discussed in the previous sections, the variance of the estimate (statistical uncertainty) came from two sources. The first source was a systematic one. This systematic uncertainty gives rise to the inconsistency of the histogramming and orthogonal function approaches and the biases of the Rosenblatt, Parzen and k-th nearest neighbor estimators. The second source is purely statistical in nature and arises from the sampling fluctuations inherent in the random nature of the data. Associated with each of these estimators is a scale parameter (number of histogram bins, number and type of orthogonal functions, window size, h, for Rosenblatt and Parzen, and number of nearest neighbors, k). As the value of this scale parameter is varied, the amount of variance contributed from the two sources has opposite behavior. Those values that give small systematic variance usually give large statistical variance and vice versa. Thus, the choice of parameter value is a compromise between these two effects and there is usually an optimum parameter value for each specific problem where the sum from two sources is minimal. The scale can also be influenced by considerations outside the data. For example, if each data point has associated with it a measurement uncertainty, then it will make little sense to make the scale parameter much smaller than this uncertainty.

If it is possible to reduce the statistical variance by some external means, then the scale parameter can be adjusted to further reduce the systematic uncertainty, resulting in a much more accurate density estimation. This is the purpose of smoothing. Smoothing makes the assumption that the true probability density, \( p(x) \), is reasonably continuous and does not change value dramatically for small changes in \( x \). Thus, any such rapid changes in the estimate, \( \hat{p}_N(x) \), must be caused by the statistical fluctuations in the estimation procedure. By taking overlapping averages of successive estimates, one hopes to dampen these fluctuations while preserving the true shape of \( p(x) \). In the language of Fourier transforms, the assumption is that the Fourier transform of \( \hat{p}_N(x) \), is composed of high frequency components resulting from the statistical fluctuations, and lower frequency components resulting from the true probability density \( p(x) \).
The problem of smoothing is to filter out the high frequency components leaving the lower frequencies which are representative of the true probability density. [In fact, one smoothing algorithm simply Fourier transforms $\hat{P}_N(x)$, attenuates the high frequencies, and then transforms back to the original coordinate representation.]

Smoothing techniques have been extensively studied and applied in many fields to all kinds of data (not just counted data), and there are many smoothers described in the literature. Each of these smoothers has special properties and applications for which it is most effective. Only one type of smoother will be discussed here; the nonlinear, robust smoothers suggested by Tukey. 10) Although these smoothers were not specifically designed for smoothing counted data (they are probably more robust than is necessary), our experience indicates they seem to work quite well for that purpose. They are also especially easy to understand and implement.

These smoothing algorithms have three components; running medians, running means, and quadratic interpolation. Consider a sequence of observed values $\{y_i\}_{i=1}^n$, and it is desired to produce from them another sequence of values $\{z_i\}_{i=1}^n$, which will be the smoothed representation of the original set. The first ingredient of the smoothing process is running medians of three. That is

$$z_i = \text{median} (y_{i-1}, y_i, y_{i+1}) .$$

(54)

For the end points, we take

$$z_1 = \text{median} (3z_2 - 2z_3, y_1, z_2)$$

$$z_n = \text{median} (z_{n-1}, y_n, 3z_{n-1} - 2z_{n-2}) .$$

(55)

Running medians of three yield the following results:

1) monotonic sequences are unchanged.

2) points that are larger or smaller than both their adjacent points will be moved inward (i.e., set equal to the closest adjacent point).

Next, running medians of five are applied to the results of the running medians of three. That is,

$$z_i = \text{median} (z_{i-2}, z_{i-1}, z_i, z_{i+1}, z_{i+2}) .$$

(56)
For this situation, there are two special cases, the end points and next to end points. The next to end points are evaluated as medians of three:

\[ z_2 = \text{median} \left( z_1, z_2, z_3 \right) \]

\[ z_{n-1} = \text{median} \left( z_{n-2}, z_{n-1}, z_n \right) . \]

The end points are treated as medians of one; that is, simply copied:

\[ z_1 = z_1 \]

\[ z_n = z_n . \]

Running medians of five yield the following results:

1) monotonically rising or falling sequences are unchanged.
2) flat tops or bottoms of length three or greater are unchanged.
3) tops and bottoms (compared to two adjacent values) of length less than three are moved inward.

The final step in running medians is to apply another running medians of three to the results of the running medians of five step. In this case, however, the end points are simply copied. This part of the smoothing procedure (running medians) is called "353" for running medians of three, followed by running medians of five, followed by running medians of three. The 353 running medians procedure goes a long way toward smoothing the data. It, however, still has two shortcomings. First, rising and falling monotonic sequences are unaffected. This is not always good. A sequence can be monotonic and still not be considered smooth. As an example, consider the sequence

\[ 1 3 4 7 66 72 74 . \]

A second shortcoming of the 353 procedure is that it clips or flattens peaks and valleys to leave flats three values long. This gives the smoothed result an unnatural appearance of having a discontinuous derivative.

This latter deficiency can be remedied by quadratic interpolation. One looks for three adjacent equal values surrounded by values on each side that are either both lower or higher than the flat value. For each such occurrence, a quadratic fit is made through the two points adjacent to the flat, and the point in the flat next to the adjacent point with the value farthest from the three flat
value. The other two points in the flat are then given values corresponding to this quadratic.

The monotonic discontinuity problem can be dealt with by Hanning or running means (averages)

$$z_i = \frac{1}{4} z_{i-1} + \frac{1}{2} z_i + \frac{1}{4} z_{i+1},$$  \hspace{1cm} (59)

with simple copy

$$z_i = z_i \quad \text{and} \quad z_n = z_n$$  \hspace{1cm} (60)

for the end points.

This smoother is referred to as

353QH

where 353 represents the running median block, "Q" stands for the quadratic interpolation step for the three-flats, and "H" refers to the Hanning or running averages step. This smoother (like most) follows straight lines rather well but tends to over-smooth (cut-off) real peaks and valleys or any region with large second derivatives. This defect can be greatly reduced by "twicing". Consider the residuals after the smooth, defined as

$$r_i = y_i - z_i$$  \hspace{1cm} (61)

where \(\{y_i\}_{i=1}^{n}\) is the original data sequence and the set \(\{z_i\}_{i=1}^{n}\) is the resulting sequence from the 353QH smoothing procedure. The sequence \(\{z_i\}_{i=1}^{n}\) is referred to as the "smooth", whereas the sequence \(\{r_i\}_{i=1}^{n}\) is referred to as the "rough". Twicing consists of smoothing the rough (using 353QH procedure) and adding the result to the previous smooth. That is,

$$z = \text{smooth (y) + smooth (r)}$$

or

$$z = \text{smooth (y) + smooth \{y - \text{smooth(y)}\}}.$$  \hspace{1cm} (62)

The complete procedure, including twicing, is labeled

353QH, twice.

One can imagine many variations on this basic smoothing procedure. Beaton and Tukey\(^\text{11}\) suggest

3G53QH, (more than twice)
as a useful alternative. Here, "G" is a conditional Hanning; if the signs of
three adjacent values alternate, the middle value is replaced by the Hanning all
three; otherwise, the value is unchanged. Other (simpler) algorithms that have
been successfully employed are 53H, twice and 95H, twice.

As indicated above, one might consider "thricing" or even more repeated
applications of twicing. In fact, one could envision a variable number of re-
peated applications of residual smoothing and adding until the remaining resi-
duals meet some terminating condition.

As mentioned above, this particular type of smoother is not specifically
designed for counted data. It was designed for more general situations where
the fluxuations can be much more severe than those arising from purely sta-
tistical mechanisms, and where there is no information on the expected sizes of
the fluxuations. In particular, it treats all similar sized fluxuations on an equal
footing. For counted data, this is not desirable. For example, in histograms
one knows that the statistical fluxuations are proportional to the square root of
the number of counts, so that the smoother should allow larger residuals in high
count regions and smaller residuals for smaller number of counts.

For histograms, this problem can be overcome by transforming the density
estimate to its stable variance representation. That is

\[ y_i = f[p_N(x_i)] \]  \hspace{1cm} (63) \]

where the function \( f(u) \) is chosen such that the expected statistical variance of \( y_i \)
is constant. From Eq. 22, we see that

\[ f(u) = \sqrt{u} \]  \hspace{1cm} (64) \]

Thus, if we input to the smoother

\[ y_i = \sqrt{p_N(x_i)} \]

statistical theory tells us that the statistical fluxuations are expected to be the
same for all the \( y_i \)'s and they should be so treated by the smoother. To obtain
the smoothed density estimate, we square the smoothed output from the
smoother

\[ \text{smooth } [p_N(x_i)] = z_i^2 \]  \hspace{1cm} (65) \]
For the other density estimators the solution is not straightforward. Since for these estimators the estimates overlap, the relative fluctuations have a more complicated dependence.

The stable variance representations for the Rosenblatt and Parzen estimators are (from Eq. 38)

\[ f(u) = \sqrt{u} \]  

and for the k-th nearest neighbor estimator (from Eq. 47)

\[ f(u) = \log(u) \]  

The statistical fluctuations for these estimators are probably more constant in their stable variance representations than in their original representations. However, these estimators already provide a reasonably smooth probability density estimate so that less would be gained in applying smoothers to their results.

Figure 2a shows a histogram of the data of Figure 1, using 170 instead of 40 bins. Comparing Figures 1a and 2a, one sees that the 170 bin representation is clearly less accurate. The increase in statistical variance for the 170 bins overwhelms the decrease in systematic variance. Figure 2b superimposes over the 170 bin histogram, the result of applying the smoothing algorithm discussed above. The smoothed representation is clearly much more continuous than the unsmoothed histogram. Figure 2c plots the square root residuals between the smooth and the original data. These residuals are the differences between the square root of the original histogram and the square root of the smooth. As discussed above, these residuals are expected to have constant size. A value of ±0.5 for a root residual corresponds to one standard deviation (values of ±1 correspond to two standard deviations, etc.). Inspection of Figure 2c indicates that the smooth is indeed a reasonable representation of the original data.

Figure 2d compares this smooth to the true probability density function, Eq. 51, from which the data were generated. The correspondence is seen to be quite good, especially in the region of the peak. To obtain a quantitative comparison, the average variance, \( E[V_{710}] \), (Eq. 52) was computed for this density estimate (smooth of 170 bin histogram) using the same nine trials as for the four other density estimators. The result was

\[ E[V_{710}] = 0.035 \]

Comparing this result to those presented in Table 1, we see that this estimation procedure is more accurate than all but the k-th nearest neighbor technique.
Comparison of Figure 2d to Figures 1b–d shows why. Although the smooth has a little more statistical variance than the others, it has very little of the bias in the regions of high second derivative, \( p''(x) \). This is mainly due to the twicing component of the smoother. These results also indicate that the main ingredient of the Rosenblatt and Parzen estimators that makes them more accurate than histogramming is their relative smoothness. This is also true for the \( k \)-th nearest neighbor estimate but it has the additional ingredient of adaptability to the data, yielding a further reduction in the variance.

The main advantage of the smoothed histogram estimate is its computational economy. Since it operates directly on the histogram and requires no additional information from the data, its computational requirements are very nearly the same as for the histogram. The increase in computation required for the smoothing operation is very small and is independent of the data sample size. Thus, smoothing gives the best of both worlds: the computational economy of histogramming and accuracy comparable to the consistent estimators.

There is one disadvantage to this approach. Namely, there are no formulas for the variance of the estimate analogous to Eqs. 22, 38 and 47 for the other estimators. Thus, one has to essentially guess at the statistical uncertainty of this estimate. A very crude upper limit is, of course, provided by the variance of the histogram estimate before the smoothing. Comparing Figures 2a and 2d, we see that typically the statistical uncertainty in the smooth is a small fraction of that for the unsmoothed histogram.

An important aspect of smoothing is inspection of the residuals, as in Figure 2c. If these residuals tend to be large or have a systematic trend, then one has less confidence in the smoothed result. If this is the case, one could again smooth the residuals and add it to the data smooth (i.e., thricing). This process can be repeated until there is no change in the resulting smooth (i.e., the smooth of the remaining residuals has a constant value of zero).

4.3 Parametric Estimation

As described earlier, in parametric (model dependent) analysis the data probability density function, \( p(x) \), is assumed to be a member of a parameter-ized family of distributions

\[
p(x) = p(\vec{a}; x)
\]

where \( \vec{a} \) is the set of parameters that specify the particular member of the parameterized family. The problem of density estimation becomes that of
estimating the parameter values from the data distribution, i.e.,

$$\hat{p}_N(x) = p(\hat{\mathbf{a}}_N, x)$$

(68)

where $\hat{\mathbf{a}}_N$ are the estimated values of the parameters $\mathbf{a}$. As described in an earlier section, one constructs a set of statistics

$$\mathbf{Y} = \mathbf{\Phi} (x_1, x_2, \ldots, x_N)$$

which are random variables with joint probability density function $p_N(\mathbf{a}, \mathbf{Y}_N)$ that (for a good estimator) is sharply peaked near $\mathbf{Y}_N = \mathbf{a}$.

The particular set of values $\mathbf{Y} = \hat{\mathbf{a}}$, resulting from a given set of $x$'s (experiment), is called an estimate of $\mathbf{a}$. Statistics used for estimation are called estimators.

There are a wide variety of estimators useful for one-dimensional data. The technique described in Eqs. 5-10 is called the method of moments. The most highly promoted estimator is called maximum likelihood. For this estimator one forms the likelihood function

$$L_N(\mathbf{\hat{a}}, x_1 \ldots x_N) = \prod_{i=1}^{N} p(\mathbf{\hat{a}}, x_i)$$

(69)

and chooses as an estimate for $\mathbf{a}$, the set of values $\mathbf{\hat{a}}$, that maximize $L_N(\mathbf{\hat{a}}, x_1 \ldots x_N)$ with respect to $\mathbf{\hat{a}}$. This can be expressed as

$$\hat{\mathbf{a}} = \max_{\mathbf{\hat{a}}} \left[ L_N(\mathbf{\hat{a}}, x_1 \ldots x_N) \right],$$

(70)

or as the solution to the set of simultaneous equations

$$\frac{\partial L_N}{\partial a_i} (\mathbf{\hat{a}}, x_1 \ldots x_N) = 0 .$$

(71)

Usually, in practice, one uses as the estimator

$$w_N(\mathbf{\hat{a}}; x_1 \ldots x_N) = \log L_N(\mathbf{\hat{a}}, x_1 \ldots x_N)$$

$$= \sum_{i=1}^{N} \log p(\mathbf{\hat{a}}, x_i)$$

$$= N \mathbf{E}_x \left[ \log p(\mathbf{\hat{a}}, x) \right] .$$

(72)
Since the logarithm is a monotonic function of its argument, the estimates will be the same. There are numerous plausible arguments for why the maximum likelihood estimator should be good but the essential results are:

1) the likelihood estimator is consistent.

2) the likelihood estimator is asymptotically efficient. That is, as \( N \to \infty \) the likelihood estimate has minimum possible variance. 3)

3) Asymptotically (again as \( N \to \infty \)), one has

\[
p_N(\tilde{a}, \hat{a}) = L(\tilde{a}; a) = \frac{1}{\sqrt{2\pi}^{m/2} |\Sigma|} \exp\left(-\frac{1}{2} (\hat{a} - \tilde{a}) \Sigma^{-1} (\hat{a} - \tilde{a})\right)
\]

i.e., \( \tilde{a} \) has a normal density distribution centered at \( \tilde{a} \), where

\[
\lim_{N \to \infty} \text{trace}(\Sigma^{-1}) = 0.
\]

It must be emphasized that the maximal efficiency of the likelihood estimator is purely an asymptotic property, and for finite \( N \), there may well be other estimators that are more efficient for particular problems. It should also be kept in mind that the likelihood estimator is unbiased for most problems. (Since the estimator is consistent, the bias must approach zero as the sample size increases toward infinity.)

For large sample size, \( N \), the empirical likelihood function can be used to estimate the variance of the estimate as well as the mean. Assuming the sample size is large enough so that Eq. 73 is a good approximation \( p_N(\tilde{a}; \hat{a}) \), and that \( \hat{a} \) is a good approximation to \( \tilde{a} \), one can estimate \( \Sigma \) as

\[
\Sigma^{-1} = \left[ \frac{\partial L}{\partial \tilde{a}} \right]_{\tilde{a} = \hat{a}, Y = \tilde{a}}^{\hat{a}}.
\]

This equation is commonly used in particle physics to determine the variance of likelihood estimations. It should be noted that this procedure is an approximation on two counts. First, and most important that the empirical likelihood function obtained for a particular experiment, \( L_N(\tilde{a}; x_1 \ldots x_N) \), is a good approximation to the true likelihood function \( L_N(\tilde{a}; x_1 \ldots x_N) \) (i.e., the one that would be obtained by averaging over all possible experimental results), and second, that the likelihood function has the multivariate normal shape given by Eq. 73.
Besides the method of moments and the maximum likelihood method, one can construct estimators by first performing a nonparametric density estimation, \( \hat{p}_N(x) \), (as discussed in the previous sections) and then forming a dissimilarity measure between the nonparametric estimate and the parametric representation

\[
d(\vec{a}) = \int_R D[\hat{p}_N(x), \ p(\vec{a}, x)] \, dx .
\]

(75)

Some examples of dissimilarity functions are:

\[
D[\hat{p}_N(x), \ p(\vec{a}, x)] = |\hat{p}_N(x) - p(\vec{a}; x)|^\ell
\]

(76a)

or

\[
D[\hat{p}_N(x), \ p(\vec{a}, x)] = \left(\frac{|\hat{p}_N(x) - p(\vec{a}; x)|^\ell}{\sqrt{V_N[\hat{p}_N(x)]}}\right)
\]

(76b)

where \( \ell \) is greater than zero. The quantity \( V_N[\hat{p}_N(x)] \) is the variance of the density estimate at the point \( x \). The estimate for the parameters, \( \vec{a} \), is taken to be those values, \( \hat{\vec{a}} \), that minimize the dissimilarity measure between the nonparametric density estimate and the parameterization, i.e.,

\[
\hat{\vec{a}} = \min_{\vec{a}}^{-1}[d(\vec{a})] ,
\]

(77a)

or is the solution to the set of equations

\[
\frac{\partial d(\vec{a})}{\partial \vec{a}} = 0 .
\]

(77b)

As an example, if one chooses histogramming for the nonparametric density estimator, \( \hat{p}_N(x) \), and Eq. 76b with \( \ell = 2 \) as the dissimilarity measure, then from Eqs. 75, 14, 15 and 20 one has

\[
d(\vec{a}) = \sum_{i=1}^{M} \frac{[\hat{n}_i - \bar{n}_1(\vec{a})]^2}{\bar{n}_1(\vec{a})}
\]

(78)
where $M$ is the number of bins or channels, $r_i$, and

$$
\hat{n}_1 = N \int_{r_i} \hat{p}_N(x) \, dx
$$

and

$$
\overline{n}_1(a) = N \int_{r_i} p(a; x) \, dx.
$$

(79)

This, of course, is the familiar least squares estimator used extensively in particle physics data analysis. Another dissimilarity measure often used with the histogramming density estimator is

$$
d(a) = -\log(N!) \sum_{i=1}^{M} N \hat{p}_i \log \hat{p}_i(a) - \log [(N\hat{p}_i)!].
$$

(80)

This dissimilarity measure is often referred to as the log binned likelihood method. However, it is important to distinguish it from the actual maximum likelihood method of Eqs. 69-72, which does not require a preliminary non-parametric density estimate.

Other nonparametric density estimators, as well as other dissimilarity measures, may be used. Since the consistent estimators (Rosenblatt, Parzen, $k$-th nearest neighbor) tend to be more accurate than histogramming, parametric estimation using them will also tend to be more accurate (have smaller variance). The best value for the power $l$ (or more generally the choice for a dissimilarity measure) depends on the particular problem at hand. It can be shown that if $\hat{p}_N(x)$ has a normal distribution centered at $p(a, x)$ with variance $V_N[p(x)]$, then Eq. 76b with $l=2$ (i.e., least squares) is optimum. For other distributions of $\hat{p}_N(x)$ other dissimilarity measures are best. For example, if $\hat{p}_N(x)$ has a square window function distribution centered at $p(a, x)$ with width $w(x)$, then $l=\infty$ would be optimum, i.e.,

$$
d(a) = \max_x \left[ \frac{|p_N(x) - p(a, x)|}{w(x)} \right].
$$

(81)

Generally, the smaller the tails of the probability distribution of $\hat{p}_N(x)$ about $p(a, x)$, the larger the optimum value of $l$ becomes. On the other hand, the higher the value for $l$ the less robust the estimate becomes. As mentioned
earlier, least squares ($\ell=2$) is already a very non-robust estimator so that high values of $\ell$ (including $\ell=2$) should be used with great care.

Estimators formed by constructing dissimilarity measures between nonparametric density estimates and the parameterized density, are usually much less efficient than the more direct methods of moments and maximum likelihood. This is because of the two-step nature of the estimation. First, the nonparametric density estimate must be made, and then this nonparametric estimation is used as input for a parametric estimation. Both stages involve statistical uncertainty. Since nonparametric procedures generally tend to have low efficiency, the first stage tends to contribute most heavily to the statistical uncertainty of the total estimate. Also, for nonparametric density estimation one must choose the value of the scale parameter and there are generally no good guidelines for this. Finally, there is the additional statistical uncertainty introduced by the second (parametric) stage of the estimation procedure, as well as further arbitrary parameters associated with the choice for a particular dissimilarity measure.

The advantage of this two-stage procedure is that it provides more information. Namely, the actual value of $d(\hat{\alpha})$ at the solution can be used as a measure of the goodness-of-fit of $p(\hat{\alpha},x)$ to the data. Goodness-of-fit testing is discussed below under hypothesis testing. The direct parametric estimators that do not involve a preliminary nonparametric density estimate cannot be used for goodness-of-fit testing. However, this is no real disadvantage since one can use them for the estimation procedure and then do a subsequent goodness-of-fit test.

Another advantage of the two-stage procedure is computational economy. Usually these statistics (especially when histogramming is used) involve considerably less computation than the direct parametric estimators.

5. A MINI INTRODUCTION TO HYPOTHESIS TESTING

The purpose of hypothesis testing is less ambitious than density estimation. For the latter, the attempt is to infer from the random data sample the actual probability density function from which it was drawn. For hypothesis testing, one wishes to use the random sample to simply confirm or reject a preconceived notion (theory) concerning a property of $p(x)$, or to distinguish between two or several possible properties.
Like estimation, hypothesis testing divides into the two subclasses, parametric and nonparametric. In parametric, \( p(x) \) is assumed to be a member of a parameterized family of density distributions \( p(\tilde{a}; x) \). However, instead of trying to estimate the most likely values for \( \tilde{a} \) as in estimation, the purpose is to accept or reject the proposition that \( \tilde{a} \) has a preconceived value, or to distinguish between several alternate preconceived values. In nonparametric hypothesis testing, no parameterized family is assumed for \( p(x) \) and the hypotheses concern general properties of \( p(x) \) that are formulated independently of its specific functional form.

One of the most common hypotheses to be tested is that \( p(\bar{x}) \) is a particular function of \( \bar{x} \), \( p(x) = f(x) \). This preconceived notion is to be tested against the notion that \( p(x) \neq f(x) \). That is, the hypothesis \( p(x) = f(x) \) is to be tested against all possible alternate hypotheses. This type of hypothesis testing is called goodness-of-fit testing and is discussed in the next section.

This section deals with using the data points \( \{x_1^i\}_{i=1}^N \) to test a specific hypothesis, \( H_0 \), (referred to as the null hypothesis) against a specific alternate hypothesis \( H_1 \). As in estimation, one constructs a statistic

\[
Y = \phi(x_1, x_2, \ldots, x_N)
\]

from the data points. This statistic is a random variable with probability density \( p_N^{(0)}(Y) \) if the null hypothesis, \( H_0 \), is true and \( p_N^{(1)}(Y) \) if the alternate hypothesis, \( H_1 \), is true. The design goal is to choose a statistic such that \( p_N^{(0)}(Y) \) is as different as possible from \( p_N^{(1)}(Y) \) for the given two hypotheses. Specifically, the overlap

\[
\int_R p_N^{(0)}(Y) p_N^{(1)}(Y) \, dY
\]

should be as small as possible. Here \( R \) is the range of all possible values of \( Y \). Statistics used for hypothesis testing are called test statistics. Clearly a value of \( Y \), for which \( p_N^{(0)}(Y) \) is large while \( p_N^{(1)}(Y) \) is small, is evidence for the truth of \( H_0 \) and vice versa. In hypothesis testing, one divides the test statistic space, \( R \), into two regions: a region of rejection, \( r \), and a region of acceptance, \( R-r \), such that \( H_0 \) will be regarded as false (\( H_1 \) true) if the value of \( Y \) is in \( r \), and \( H_0 \) will be regarded as true (\( H_1 \) false) if the value of \( Y \) falls in the region \( R-r \). The quantity

\[
\alpha_N = \int_r p_N^{(0)}(Y) \, dY
\]
is called the level of significance or size of the test. It is the probability that the null hypothesis, \( H_0 \), will be declared false when it is, in fact, true. Rejection of \( H_0 \) when it is true is called loss or error of the first kind. The quantity

\[
\beta_N = \int_{R-r} \mathbb{P}_N^{(1)}(Y) \, dY
\]  

(85)

is the probability that \( H_0 \) will be declared true when it is, in fact, false (\( H_1 \) is true). This is called contamination or error of the second kind. The quantity

\[
1 - \beta_N = \int_{r} \mathbb{P}_N^{(1)}(Y) \, dY
\]  

(86)

is the probability that \( H_1 \) will be declared true when it is, in fact, true, and is called the power of the test. Clearly the rejection region, \( r \), should be chosen so that for a given size, \( \alpha_N \), the contamination, \( \beta_N \), is as small as possible (power \( 1 - \beta_N \), as large as possible). The experimenter usually decides what loss, \( \alpha_N \), he can tolerate and then chooses a test statistic and rejection region so as to maximize the power, \( 1 - \beta_N \), of the test. Clearly, to be able to do hypothesis testing, the probability density functions \( \mathbb{P}_N^{(0)}(Y) \) and \( \mathbb{P}_N^{(1)}(Y) \) must be known or calculable for the chosen test statistic.

Consider the following very simple example

\[ H_0: \quad p(x) \text{ is a normal distribution with mean } \mu = 0 \]

\[ H_1: \quad p(x) \text{ is a normal distribution with mean } \mu = \mu_1 \text{ (where } \mu_1 > 0) \]

A good test statistic for this problem is

\[
t = \frac{\frac{1}{N} \sum_{i=1}^{N} x_i}{\left[ \frac{1}{N} \sum_{i=1}^{N} x_i^2 - \left( \frac{1}{N} \sum_{i=1}^{N} x_i \right)^2 \right]^{1/2}}
\]  

(87)

which is known as a \( t \)-statistic. The probability density function \( \mathbb{P}_N^{(0)}(t) \) if the null hypothesis, \( H_0 \), is true can be shown to be a Students–t distribution with \( (N-1) \) degrees of freedom

\[
\mathbb{P}_N^{(0)}(t) = \frac{\Gamma(N/2)}{\sqrt{N\pi} \Gamma\left(\frac{N-1}{2}\right)} \left( 1 + \frac{t^2}{N(N-1)} \right)^{-N/2}
\]
while for the alternate hypothesis, \( p_N^{(1)}(t) \) is a similar Student's-t distribution centered at \( \mu_1 \). For large \( N \) (\( \geq 100 \)), the Student's-t distribution tends toward the standard normal distribution so that
\[
p_N^{(0)}(t) = \frac{1}{\sqrt{2\pi N}} e^{-t^2/2N} \quad (N > 100)
\]
\[
p_N^{(1)}(t) = \frac{1}{\sqrt{2\pi N}} e^{-(t-\mu_1)^2/2N}
\]

If the experimenter is willing to tolerate a loss of \( \alpha \), then the best rejection region is defined as
\[
r_N(\alpha) \leq t < \infty
\]
where \( r_N(\alpha) \) is the solution to
\[
\alpha = \int_{r_N(\alpha)}^{\infty} \frac{1}{\sqrt{2\pi N}} e^{-t^2/2N} \, dt \quad (88a)
\]
or
\[
r_N(\alpha) = \Phi^{-1}(1 - \alpha)/\sqrt{N} \quad (88b)
\]
where \( \Phi^{-1}(\alpha) \) is the inverse of the standard normal error function. The power of the test is
\[
1 - \beta_N = \int_{r_N(\alpha)}^{\infty} \frac{1}{\sqrt{2\pi N}} e^{-(t-\mu_1)^2/2N} \, dt \quad (89a)
\]
or
\[
1 - \beta_N = \Phi [ \sqrt{N} (\mu_1 - r_N(\alpha)) ] \quad (89b)
\]
where \( \Phi(\alpha) \) is the standard normal error function.

Like estimators, test statistics are rated by several qualities: consistency, efficiency, bias and robustness.
1. **Consistency**

A test is said to be consistent if the power, \( 1 - \beta_N \), approaches unity as \( N \) approaches infinity,

\[
\lim_{N \to \infty} (1 - \beta_N) = 1. \tag{90}
\]

That is, the ability to distinguish between the two hypotheses becomes better with additional data for very large samples. Note that from Eq. 89 we see that the t-statistic (Eq. 87) is consistent.

2. **Bias**

A test is said to be biased if the probability of accepting the null hypothesis is greater when it is false than when it is true. Conversely, a test is said to be unbiased if the probability of accepting the null hypothesis, \( H_0 \), is greatest when it is true. From Eq. 89 we see that the t-statistic (Eq. 87) is unbiased. As for estimators, a test can be both biased and consistent since bias is a property of test statistics for finite \( N \), while consistency refers to their properties for infinite sample size.

3. **Efficiency**

The efficiency of a test refers to its power for given hypotheses and level of significance. A test is said to be efficient or most powerful if it has the largest power possible for a given size, \( \alpha \), and given hypotheses, \( H_0 \) and \( H_1 \). A test that is most powerful for all alternate hypotheses under consideration is called a uniformly most powerful test. The efficiency of a test is the ratio of its power to the most powerful test in the given situation.

4. **Robustness**

Robustness for test statistics has similar meaning as for estimators. Namely, the effect on the power of the test when the underlying density distribution of the data deviates from a priori assumptions. Tests that suffer great loss of power when the density distribution \( p(x) \), is different than that hypothesized are non-robust, while those that maintain reasonable power over a wide range of density distributions are robust. Clearly, robust estimators are required for nonparametric applications where \( p(x) \) is not known.

The t-statistic (Eq. 87) is uniformly most powerful for all alternate hypotheses \( \mu = \mu_1 \) (i.e., for all \( \mu_1 \)) provided that the distribution of the data points, \( p(x) \), is normal. However, if the data points are not normally distributed then this test can become very inefficient. For example, if \( p(x) \) is a Cauchy
distribution
\[ p(x) = \frac{1}{1 + x^2} \] (91)

then this test has zero power. Thus, the t-statistic of Eq. 87 is not robust. It is possible, however, to formulate robust analogs of this t-statistic that have reasonable efficiency for all \( p(x) \).

5.1 Goodness-of-fit Testing

Goodness-of-fit testing is probably the most common type of hypothesis testing in high energy particle physics. Here the null hypothesis, \( H_0 \), is that the probability density function of the data is a specified one. That is,

\[ H_0 : p(x) = f(x) \]

where \( f(x) \) is explicitly given. Here a specific alternate hypothesis is not given. Or alternatively, one can consider the alternate hypothesis to be

\[ H_1 : p(x) = \{ g(x) \} \]

where \( \{ g(x) \} \) is the set of all possible alternatives to \( f(x) \). Either way the alternate hypothesis is not explicitly specified so that it is impossible to calculate a contamination or power of the test.

As in regular hypothesis testing, goodness-of-fit testing starts with choosing a test statistic,

\[ Y = \Phi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N) \]

The test statistic space, however, is not divided into an acceptance and rejection region since there is no specific alternate hypothesis to accept in favor of the null hypothesis. It should be noted that since the alternate hypothesis is the set of all possible alternatives to \( H_0 \), there is surely one out of the infinity of alternatives that will always fit better than \( H_0 \). A trivial example is

\[ f(x) = \frac{1}{N} \sum_{i=1}^{N} \delta(x - \vec{x}_i) \] (92)

where \( \{ \vec{x}_i \}_{i=1}^{N} \) are the actual data points.
In goodness-of-fit testing one calculates the probability under \( H_0 \), that the test statistic would have a value less probable than the value observed from the data. This is known as the level of significance for the test. For example, if the test statistic distribution under \( H_0 \), \( p_N^{(0)}(Y) \), decreases for large increasing \( Y \), then

\[
\alpha_N(T) = \int_T^{\infty} p_N^{(0)}(Y) \, dY
\]

would be this level of significance for a given value of \( T \). This quantity is usually referred to as the "confidence level" in particle physics. In most other scientific fields it is known as the "P-value".

Clearly, the probability density of the test statistic, \( p_N^{(0)}(Y) \), under the null hypothesis must be known or be calculable from the hypothesized data probability density \( p(x) = f(x) \). For some tests the probability distribution, \( p_N^{(0)}(Y) \), of the test statistic is independent of the data distribution and depends only on the number of data points, \( N \), and the truth of the null hypothesis. These are called distribution free tests. Most of these distribution free tests are distribution free only in the limit of infinite sample size. For this case, the distribution of various types of averages will be normal from the central limit theorem independent of the underlying distribution of the data. However, the actual sample size, \( N \), required for the asymptotic approximation to be valid does depend upon the underlying probability density distribution of the data.

Goodness-of-fit tests are constructed by formulating a dissimilarity measure between a nonparametric density estimation, \( \hat{p}_N(x) \), of the data and the hypothesized functional form \( p(x) = f(x) \),

\[
d(x_1, x_2, \ldots, x_N) = \int_R D(\hat{p}_N(x; x_1, \ldots, x_N), f(x)) \, dx . \tag{93}
\]

Two of the most common dissimilarity measures are given by Eqs. 76a and 76b. This procedure is identical to the procedure described earlier for formulating parametric estimators from nonparametric density estimates. Here, however, the objective is to determine the goodness-of-fit from the value of the dissimilarity, \( d \), rather than trying to find the values of parameters that minimize it. It is clear that any goodness-of-fit statistic can also be used as an estimator by simply adjusting parameters of \( f(x) \) to achieve the best fit (i.e., minimum dissimilarity, \( d \)). However, as discussed in the earlier sections, these are seldom
the best estimators for a given problem. On the other hand, there are no
goodness-of-fit analogs to the method of moments and maximum likelihood esti-
mators. That is, the value of the likelihood function at its maximum gives no
information as to goodness-of-fit.

Goodness-of-fit statistics of the form given by Eq. 93 tend to be distribution
free for large sample size because the distribution of \( \hat{p}_N(x) \) about \( p(x) \) is deter-
mined mainly by the central limit theorem (law of large numbers) since the
\( \hat{p}_N(x) \) are local averages.

The most common goodness-of-fit test statistic used in particle physics is
the Pearson's \( \chi^2 \) test, whose test statistic is given by Eq. 78, with

\[
\bar{n}_1 = N \int_{\hat{n}_1} f(x) \, dx .
\]  

As discussed earlier, this statistic uses histogramming as the nonparametric
estimator and a scaled Euclidean distance type measure (Eq. 76b with \( \ell = 2 \)) for
a dissimilarity measure. For large \( \hat{n}_1 = N\hat{n}_1 \), the central limit theorem requires
that \( \hat{n}_1 \) be normally distributed about its center \( \bar{n}_1 \) (under \( H_0 \)) with variance \( \bar{n}_1 \).
Thus, each term in the sum is a random variable with a standard normal dis-
tribution. It can be shown that a random variable which is the sum of squares of
M normally distributed random variables has the probability density distribution

\[
p_M^{(0)}(\chi^2) = \frac{1}{2\Gamma(M/2)} \left( \frac{\chi^2}{2} \right)^{M/2-1} e^{-\chi^2/2} .
\]  

This probability density distribution is known as a chi-square distribution with
M degrees-of-freedom. For a given value of \( \chi^2 \), determined from an experi-
ment, the significance level or p-value is simply given by

\[
\alpha_M(\chi^2) = \int_{\chi^2}^{\infty} p_M^{(0)}(\chi^2) \, d(\chi^2) .
\]  

It is important to emphasize that the \( \chi^2 \) test is very non-robust. It is clear
from Eq. 78 that those terms in the sum, for which \( \bar{n}_1 \) is very small, will domi-
nate. Thus, for these terms a very small departure from the assumptions that
lead to Eq. 95 will give rise to large departure in the results. Specifically, if
the expected number of counts \( \bar{n}_1 = N\bar{n}_1 \) is small, then the central limit theorem
cannot be applied and the residuals
\[ r_i = \frac{\hat{n}_i - \bar{n}_i}{\sqrt{\bar{n}_i}} \]  \hspace{1cm} (97)

will not have the standard normal distribution. For this case, the probability
distribution for the test statistic, \( p^{(0)}_M (x^2) \), deviates considerably from the \( x^2 \)
distribution (Eq. 95).

The precise distribution for \( \hat{n}_i = N \bar{n}_i \) is given by Eq. 17. This distribution
is reasonably well approximated by a normal for \( \bar{n}_i \geq 5 \). Therefore, if all of
the bins have at least five expected counts, Eq. 95 can be accepted as a good
approximation. If this is not the case, several remedies are possible. Tukey\(^{13} \)
suggests replacing the observed number of counts, \( \hat{n}_i \), by
\[ \hat{s}_i = 2 + 4 (\hat{n}_i) \]  \hspace{1cm} (98a)

and the expected number of counts, \( \bar{n}_i \), by
\[ \bar{s}_i = 1 + 4 (\bar{n}_i) . \]  \hspace{1cm} (98b)

These quantities are called "started counts". The motivation for using these
started counts, is that if the "raw counts", \( \hat{n}_i \), have the distribution given by
Eq. 17 then the \( \hat{s}_i \) will be much more nearly normally distributed than the raw
counts. The reason for using a smaller start for the expected number of counts,
\( \bar{n}_i \), is due to the asymmetry of the distribution of \( \hat{n}_i \) (Eq. 17) about \( \bar{n}_i \). This dis-
btribution is skewed towards lower number of counts. For example, if one count
is expected (\( \bar{n}_i = 1 \)) then zero counts (\( \hat{n}_i = 0 \)) will be observed twice as often as
two counts (\( \hat{n}_i = 2 \)). (In fact, zero counts will be observed as often as
one count!) Giving the expected number of counts a smaller start helps compensate
for this skewness so that \( \hat{s}_i \) is more nearly symmetrically distributed about \( \bar{s}_i \) as required by a normal distribution. Using started counts (\( \hat{s}_i \) and \( \bar{s}_i \)) instead
of raw counts (\( \hat{n}_i \) and \( \bar{n}_i \)) allows Eq. 95 to remain a good approximation for
smaller sample sizes.

Another method for dealing with small sample size is to use the log binned
likelihood dissimilarity measure given by Eq. 80. It can be shown that -2\( d \) has
(like the \( x^2 \) test statistic) a \( x^2 \) distribution with \( M-1 \) degrees of freedom for
infinite sample size. However, it is generally felt that this property remains a
good approximation for smaller sample size than with the \( x^2 \) test statistic.
Other goodness-of-fit tests can be constructed that use analogs of the other density estimators. The two most common are the Smirnov-Cramer-Von Mises test\(^\text{14}\) and the Kolmogorov test.\(^\text{15}\) These both use the Rosenblatt type estimator for the nonparametric density estimation. However, instead of estimating the probability density, \(\hat{p}_N(x)\), one estimates the cumulative density function

\[
\hat{P}_N(x) = \int_{-\infty}^{x} \hat{p}_N(x) \, dx
\]

using Eq. 32a. This has the advantage that no scale parameter, \(h(N)\), need be specified.

The Smirnov-Cramer-Von Mises test uses a Euclidean type dissimilarity measure (Eq. 76a, \(\ell=2\)). That is

\[
Y = d(x_1, x_2, \ldots, x_N) = \sqrt{\int_{-\infty}^{\infty} \left[ \hat{P}_N(x_1, x_2, \ldots, x_N) - F(x) \right]^2 \, dF(x)}
\]

where \(F(x)\) is the cumulative distribution function of \(f(x)\)

\[
F(x) = \int_{-\infty}^{x} f(x) \, dx.
\]

For the Kolmogorov test, a dissimilarity measure analogous to Eq. 81 is used (i.e., Eq. 76a, \(\ell=\infty\)). That is

\[
Y = d(x_1, x_2, \ldots, x_N) = \max_{-\infty < x < \infty} \left| \hat{P}_N(x_1, x_2, \ldots, x_N) - F(x) \right|
\]

The probability density function of the test statistic, \(\hat{P}_N^{(0)}(Y)\), has been calculated for these tests and their level of significance, \(\alpha(Y)\), as a function of test statistic value, \(Y\), are tabulated in standard statistical tables.

These estimators can also be used for nonparametric goodness-of-fit tests. That is, instead of comparing the experimental point set to a specific functional form, it is compared to another experimental point set. Consider two different point sets \(x_i \, i=1, \ldots, N\) and \(y_i \, i=1, \ldots, M\) drawn from unknown probability density functions \(p(x)\) and \(q(y)\), respectively. The null hypothesis is

\[
H_0: p(x) = q(y) \quad \text{for all } x \text{ and } y.
\]

The test is nonparametric because no information is assumed to be known about either \(p(x)\) or \(q(y)\). The hypothesis is only that they are the same. Here one
constructs a test statistic by forming a dissimilarity measure between the probability density estimates for the two point sets. For example,

\[ d(x_1 x_2 \ldots x_N; y_1 y_2 \ldots y_M) = \int_{-\infty}^{\infty} \left[ \hat{P}_N(z; x_1 x_2 \ldots x_N) - \hat{P}_M(z; y_1 y_2 \ldots y_M) \right]^2 d\hat{P}(z) \]  

(104)

where

\[ \hat{P}(z) = \frac{N\hat{P}_N(z) + M\hat{P}_M(z)}{N + M} \]

for the analog of the Smirnov-Cramer-Von Mises test, and

\[ d(x_1 x_2 \ldots x_N; y_1 y_2 \ldots y_M) = \max_{-\infty < z < \infty} |\hat{P}_N(z; x_1 x_2 \ldots x_N) - \hat{P}_M(z; y_1 y_2 \ldots y_M)| \]  

(105)

for the Kolmogorov.

As is the case for their parametric counterparts, these tests are distribution free (independent, under \( H_0 \), to whatever \( p(x) = q(y) \) might be), and are tabulated in statistical tables.

5.2 Visual Representation of Goodness-of-Fit

Quite often a single number representing the significance level for a goodness-of-fit test is not enough, especially if the significance is marginal or small. The experimenter usually would like to know those values of the measured variable where the correspondence between the theory and data is relatively good and, similarly, those regions that most contribute to making the fit bad. The goodness-of-fit test statistics themselves can often be used for this purpose. For example, if a \( \chi^2 \) test statistic is used, one can look for those terms in the sum that are relatively large (or small). For the Smirnov-Cramer-Von Mises and Kolmogorov tests (Eqs. 101 and 103), one can look for values of the integration variable, \( x \), that result in large or small values of

\[ \frac{d}{dx} [\hat{P}_N(x; x_1 x_2 \ldots x_N) - F(x)] . \]  

(106)

Since one-dimensional density is easily represented in graphical form (plotted as density vs. coordinate value), a common procedure is to simultaneously plot on the same graph the nonparametric density estimate \( \hat{p}_N(x) \) and the hypothesized functional form, \( f(x) \). The experimenter can then visually evaluate the goodness-of-fit. This technique suffers from some drawbacks. First, the variance of the nonparametric estimate, \( \text{V}[\hat{p}_N(x)] \) (statistical uncertainty), is
usually a nonconstant function of the coordinate, \( x \). In order to successfully evaluate local goodness-of-fit for some value of \( x \), one must known the value of \( V[\hat{p}_N(x)] \). Also, if this variance varies widely over the range of \( x \), then a quick recognition of those regions that represent significant departures is difficult. One way to alleviate this is to plot, in addition to \( \hat{p}_N(x) \) and \( f(x) \), two more quantities

\[
\hat{\sigma}_+ (x) = \hat{p}_N(x) + \sqrt{V[\hat{p}_N(x)]} \\
\hat{\sigma}_- (x) = \hat{p}_N(x) - \sqrt{V[\hat{p}_N(x)]}
\]

or alternatively,

\[
\sigma_+ (x) = f(x) + \sqrt{V[\hat{p}_N(x)]} \\
\sigma_- (x) = f(x) - \sqrt{V[\hat{p}_N(x)]}
\]  \hspace{1cm} (107a) \hspace{1cm} (107b)

In the case of Eq. 107a, these are usually represented as "error bars" centered on the corresponding density estimates. This procedure overcomes the handicap at the expense of making the graph considerably more cluttered and unreadable.

When the explicit functional form of \( V[\hat{p}_N(x)] \) is known, a better solution is to transform the ordinate of the graph, \( \hat{p}_N(x) \), to the stable variance representation. That is, both \( \hat{p}_N(x) \) and \( f(x) \) are transformed

\[
\hat{p}_N^*(x) = T[\hat{p}_N(x)] \\
f^*(x) = T[f(x)]
\]  \hspace{1cm} (108)

where the transformation function, \( T[u] \), is chosen such that the statistical variance is a constant. For the histogram, Rosenblatt and Parzen estimators

\[
T[u] = \sqrt{u}
\]  \hspace{1cm} (109a)

whereas for the kth nearest neighbor estimator

\[
T[u] = \log u
\]  \hspace{1cm} (109b)

In particular, a histogram in which the square root of the number of counts (rather than the number of counts) is plotted is called a rootogram. Rootograms have the advantage that the variance of the estimate (expected size of statistical fluctuations) is a constant independent of \( x \). Thus, a difference between
\[ \hat{p}_N^*(x) = \sqrt{\hat{p}_N(x)} \text{ and } \hat{f}(x) = \sqrt{\hat{f}(x)} \] is a direct indication of the lack of correspondence between them, and these regions can be identified quickly at a glance.

Another problem with this general technique is that humans are much more attuned to recognizing and evaluating departures from straight lines (especially horizontal ones) than highly curved lines. Humans easily and quickly identify and properly evaluate the significance of deviations from horizontal lines but have considerably more difficulty when the line has a steep slope or is highly curved.

This difficulty is also easily overcome by using a stable variance representation. This is because the expected departures of the residuals, \( r(x) = p_N^*(x) - \hat{f}(x) \), from zero are constant independent of \( p(x) \), \( \hat{f}(x) \) or \( x \). Thus, a plot of \( r(x) \) vs. \( x \) can be evaluated independently of any other information. These residuals can be investigated for systematic departures from a horizontal line at \( r = 0 \). In the case of histograms of counts (or better "started" counts) where the fluxuations are reasonably well approximated by a normal distribution, one standard deviation corresponds to a square root residual of \( r_1 = \pm 0.5 \), two standard deviations correspond to \( r_1 = \pm 1.0 \) and so on, independent of the actual number of counts in the bin. Figure 2c is an example of such a plot of residuals. Using this approach of independently plotting the residuals allows one to see at a glance whether the fit is good, or to spot those regions where it is bad.

Tukey\(^{16}\) recommends plotting the densities and residuals on a single plot. This procedure is illustrated in stages in Figure 3. Figure 3a shows an example of the traditional representation with a histogram plotted, along with the hypothesised density function superimposed. Figure 3b shows the same plot, but where the histogram bars, instead of being aligned with the horizontal axis, are aligned with the function. The histogram residuals then appear as departures from the horizontal axis. Figure 3c shows a traditional rootogram of the same data, while Figure 3d shows the rootogram aligned with the curve. The expected constant size residuals then appear as departures from the horizontal axis.

Also, the rootogram in Figure 3d is inverted so that the residuals play a more prominent role. This is called a hanging rootogram. With the hanging rootogram, the positive residuals appear as positive departures above the horizontal axis while the negative residuals appear below the axis. Finally, in Figure 3e the residuals are emphasized further by being shaded — differently for positive and negative — and the rootogram part is further suppressed. Also, horizontal lines representing \( \pm 2 \) and \( \pm 3 \) standard deviations are included.
The hanging rootogram allows the researcher to see at a glance both the general shape of the hypothesized function and the residuals from that function. The constant expected size residuals are contrasted against a horizontal line so that those regions of maximum departure, as well as the importance of the departure, can be recognized easily.
6. **MULTIVARIATE DATA ANALYSIS**

Many experiments in high energy particle physics are multivariate (sometimes referred to as multidimensional) in nature. That is, for each event several attributes or quantities are simultaneously measured. A particle reaction, resulting in n-particles in the final state, has $3n - 4$ independent measurables (not including spin information) associated with it. Most experiments measure several of these quantities and some can measure the complete set. Analyzing and interpreting data for these experiments is a problem in multivariate data analysis.

The concept of probability density is easily generalized for the multivariate case. Let $\mathbf{x}_1 = \left( x_1^{(1)}, x_1^{(2)}, \ldots, x_1^{(d)} \right)$ be a set of attributes or quantities measured for each event. If the value of each quantity is plotted along a Cartesian axis, then the set of simultaneous values can be represented as a point in a Cartesian space of dimensionality, $d$. The entire experiment can then be regarded as a collection or swarm of such points in this $d$-dimensional space. Since each point contains all the information for the corresponding event, this point swarm contains all of the information of the experiment. As in the univariate case, the purpose of data analysis is to use this point swarm to make inferences concerning the joint probability density $p(\mathbf{x})$, defined in Eq. 1. For the multivariate case, $r_1$, is a small volume in the $d$-dimensional space. Letting this volume approach zero while $n_1$ and $N$ approach infinity, one defines the notion of the value of $p(\mathbf{x})$ at a point $\mathbf{x}$. Loosely speaking, $p(\mathbf{x}_0)$ is the probability that $x^{(1)}, x^{(2)}, \ldots, x^{(d)}$ all simultaneously have the values $x^{(1)} = x_0^{(1)}$, $x^{(2)} = x_0^{(2)}$, $\ldots$, $x^{(d)} = x_0^{(d)}$. As an example, in exclusive experiments where $d = 3n - 4$, the spin averaged Lorentz invariant amplitude squared

$$\frac{1}{\sigma} |M\left(x^{(1)}, x^{(2)}, \ldots, x^{(3n-4)}\right)|^2 = p(\mathbf{x})$$

(110)

is the joint probability density function in the Lorentz invariant phase space. Here, $\sigma$ is the total cross section for the reaction. This joint probability density function contains all the information attainable concerning the momentum (nonspin) aspects of the experiment.

Statistical techniques for multivariate data analysis are much less well developed than univariate techniques. When they exist, multivariate techniques are usually straightforward extensions of the corresponding univariate techniques. For parametric density estimation, the method of moments and maximum likelihood are easily extended. One simply replaces the parametrized
univariate probability density, $p(\vec{a};x)$, by the parametrized joint probability density, $p(\vec{a};x)$. The general asymptotic ($N \to \infty$) statistical properties of these estimators is the same for the multivariate case as for the univariate case. However, the value of the sample size, where the asymptotic properties become good approximations, is usually much larger in the multivariate case.

Although easy to generalize conceptually, the computational complexity of both the moments and likelihood methods increases dramatically for high dimensionality. This is because of the general problems associated with the numerical evaluation of definite integrals in high dimensionality. Most joint probability density functions $p(\vec{a};\vec{x})$ that appear in high energy physics applications cannot be analytically integrated over the allowed range of the variables, $R$. Even for the simplest case of

$$p(\vec{a};\vec{x}) = \text{constant},$$

the integral

$$\int_{\vec{R}} d\vec{x}$$

cannot be explicitly evaluated when $R$ is the region of phase space defined by momentum and energy conservation. From Eq. 5 we see that the moments method explicitly requires, in general, the evaluation of a multidimensional integral.

Although the likelihood method (Eqs. 70 or 71) does not explicitly require multidimensional integrals, it implicitly requires them through the normalization condition (Eq. 2). The likelihood function (Eq. 69) requires that $p(\vec{a};\vec{x})$ be a proper probability density function, i.e.,

$$\int_{\vec{R}} p(\vec{a};\vec{x}) \, d\vec{x} = 1 \tag{2}$$

where $R$ is the region of the allowed values for the variables, $\vec{x}$. Usually the Lorentz invariant amplitudes that arise in high energy physics are not so normalized, requiring the evaluation

$$p(\vec{a};\vec{x}) = \left| \frac{M(\vec{a};\vec{x})}{\int_{\vec{R}} |M(\vec{a};\vec{x})|^2 \, d\vec{x}} \right|^2 \tag{111}$$

before the likelihood method can be applied. The value of the integral generally depends upon the values of the parameters. Thus, if an iterative scheme is used to solve Eq. 70, then the integral must be re-evaluated for each step in the iteration procedure. This can be extremely costly, computationally, especially if Monte Carlo techniques are required for the integrations.
Nonparametric multivariate density estimation is very difficult. This is due to the extreme sparseness of the multidimensional data space. Even for the very largest experiments being contemplated, the average counting density in the d-dimensional space of the measurement point swarm is very small. To make the situation even worse, even though the average density is very small there are usually one or several very small regions or surfaces of complex shape where the density becomes quite large.

For these reasons, straightforward extensions of the univariate density estimators do not usually work. For example, consider the multivariate analog of histogramming. Even if only ten bins or channels per dimension are chosen (a very coarse binning), there would be \((10)^d\) cells in the d-dimensional space. For \(d=10\) and a large experiment \((N \approx 10^6)\) the average counting rate would be \(.0001\) per cell, with a very tiny number of the cells containing all of the events.

Although avoiding cells, the Rosenblatt and Parzen estimators do not fare much better. This is because of the huge density variation in the measurement space. Thus, choosing a scale parameter, \(h(N)\), that is adequate for the sparse regions is much too big in the dense regions and vice versa. For example, consider a \(p(\vec{x})\) that contains two components of equal probability content (number of events)

\[
p(\vec{x}) = p_1(\vec{x}) + p_2(\vec{x})
\]

where the scale (extent) of the first is \(10\%\) of the other in each of the dimensions. Then for \(d=10\) the extent of the first component (1/2 of the events) is \(10^{-10}\) times the extent of the other in the 10-dimensional measurement space. A scale parameter value that was useful where \(p_2(\vec{x})\) dominates is \(10^{10}\) times too large for those regions where \(p_1(\vec{x})\) dominates and, conversely, if the scale parameter value was chosen to be accurate in the \(p_1(\vec{x})\) region, it would not work at all in the other regions of the space (there would never be any counts within the window).

The only density estimator that has a chance of being useful for multivariate applications is the kth nearest neighbor. Because of its property of adapting to the data, it does not suffer from the aforementioned difficulties. However, there are additional problems that severely limit its usefulness as a density estimator. These problems can be understood by inspecting Eq. 46. The multivariate analog
of this equation\(^8\) for the bias is (to second order)

\[
b_N(x) = E\left[\hat{p}_N(x)\right] - p(x) = \frac{\Gamma^{2/d} \left(\frac{d+2}{2}\right)}{2\pi^{(d+2)/2} N^{2/d} p(x)} \text{tr}\left[p''_{ij}(x)\right]
\]

(112)

where \(\text{tr}\left[p''_{ij}(x)\right]\) is the trace of the Hessian (second derivative \(\frac{\partial^2 p(x)}{\partial x_i \partial x_j}\)) matrix evaluated at \(x\). As discussed above, typical joint probability density functions in high dimensions are characterized by very small average density \((p(x)\) very small) and large variability \((\text{tr}\left[p''_{ij}(x)\right]\) big). Thus, the bias of the estimate can be expected to be very large.

This large bias can be understood intuitively as follows. Consider a point located in a sparse region of the measurement space. As one expands a spherical volume centered at the point, data points will be encountered very slowly. Large increases in volume result in small accumulation of data points until the sphere borders a dense region where vast numbers of data points will be included for small changes in volume. Thus, the kth nearest neighbors of the sparse points will tend to include considerable contamination from dense regions, resulting in a large overestimate of the density at the point. As the sample size becomes infinite \((N \to \infty)\) while \(k(N)/N \to 0\), the kth nearest neighbors to a point will all be very close, eliminating this effect and sending the bias to zero, rendering the estimate consistent (as seen from Eq. 112). However, the sample sizes required for these asymptotic results to be useful are truly astronomical.

Because of its large bias, the kth nearest neighbor technique is not useful for direct density estimation. However, it can be successfully used for multivariate techniques where absolute density estimation is not required. Some of these techniques are discussed below.

Unlike univariate data, multivariate data is difficult to present (except for the special case of \(d=2\)). This is due to the inability of humans to perceive in more than three dimensions. Methods using interactive computer graphics to aid humans to perceive and manipulate multidimensional data is described in Refs. 17 and 18.

Because of the great difficulties in directly dealing with multivariate data, nonparametric techniques have, in the past, sought to reduce d-dimensional data to one, two or three dimensions where human perception can be employed and where nonparametric density estimation is practical. The most common tool for this dimensionality reduction is projection. With projection, one integrates over all but one or a few of the variables leaving a density function of lower
dimensionality. For example, with one dimensional projection

\[ p(x_1) = \int_R p(x_1, x_2, \ldots, x_d) \, dx_2 \, dx_3 \ldots \, dx_d \]

or more generally

\[ p(y) = \frac{d}{dy} \left[ \int_R p(\mathbf{x}) \, d\mathbf{x} \right] \]

where \( y = T(\mathbf{x}) \). Here \( T(\mathbf{x}) \) is some arbitrary function of the measurement variables. For two dimensions

\[ p(y_1, y_2) = \frac{d^2}{dy_1 \, dy_2} \left[ \int_R p(\mathbf{x}) \, d\mathbf{x} \right] \]

where \( y_1 = T_1(\mathbf{x}) \) and \( y_2 = T_2(\mathbf{x}) \).

Operationally, \( p(y) \) (one-dimensional projection) can be estimated by evaluating \( y_1 \) for each event, \( y_1 = T(\bar{x}_1) \), and performing a univariate density estimation on the resulting set of values, \( \{y_1\}_{i=1}^N \). That is,

\[ \hat{p}_N(y; y_1, y_2, \ldots, y_N) = \hat{p}_N[y; T(\bar{x}_1), T(\bar{x}_2), \ldots, T(\bar{x}_N)] . \]

By judicious choices for various transformation functions, \( T(\mathbf{x}) \), one hopes to infer some of the salient features of the multivariate density, \( p(\mathbf{x}) \). Similarly, for two-dimensional projections, one estimates

\[ \hat{p}_N(y_1, y_2) = \hat{p}_N[y_1, y_2; T_1(\bar{x}_1), \ldots, T_1(\bar{x}_N), T_2(\bar{x}_1), \ldots, T_2(\bar{x}_N)] , \]

or simply maps the points \( \{(y_1, y_2)\}_{i=1}^N \) onto the two-dimensional plane (scatter plot). All of the one-dimensional density estimators discussed earlier can usually be extended to two dimensions without encountering the difficulties of high dimensionality (\( d > 3 \)).

As an aid to this process, masking (making cuts) is sometimes used in conjunction with projection. With masking, only a preselected subregion of the total measurement space is chosen for projection

\[ p_T(y) = \frac{d}{dy} \left[ \int_{\mathbf{r} \in R} p(\mathbf{x}) \, d\mathbf{x} \right] \]

or equivalently

\[ p_T(y) = \frac{d}{dy} \left[ \int_R p(\mathbf{x}) B(\mathbf{x}) \, d\mathbf{x} \right] \]

where \( B(\mathbf{x}) \) is the masking function.
where

\[ B(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in r \\ 0 & \text{otherwise} \end{cases} \]  \quad (117)

Operationally, \( B(\vec{x}_i) \) is evaluated for each data point, \( \vec{x}_i \), and those points for which \( B(\vec{x}_i) = 0 \) are excluded from the density estimate, \( \hat{p}_N(y) \). By clever choices for both transformation functions, \( T(\vec{x}) \), and masking functions, \( B(\vec{x}) \), one can often learn a great deal about the underlying multivariate joint probability density function, \( p(\vec{x}) \), describing the data.

There are several fundamental drawbacks to the projection approach. Most important is the tremendous loss of information inherent in the projection process. By integrating over all of the measurement variables but one (or two), a great deal of the information contained in the data is lost. This makes complex interrelationships between the variables very difficult to discover using projection and masking only.

Another problem is reflections due to the non-rectangular shape of the boundaries of the measurement space. Momentum and energy conservation impose complicated boundaries on the measurement variables. These boundaries cause two problems. First, their shape appears in the projection along with any structure coming from the actual density function (dynamics) of the data. Even for constant multivariate data density, projected densities have nonconstant distributions due to the shape of these boundaries. From the projections alone, it is not possible to tell whether an effect comes from the data density (dynamics) or from the boundaries (kinematics).

Another more serious problem caused by the non-rectangular nature of the boundaries is the reflection of actual dynamical effects in the multivariate density. This is illustrated in Fig. 4 with the classical example of projecting a two-dimensional Dalitz plot onto two one-dimensional projections. Here the density is made up of a constant background plus an enhancement in the vertical coordinate. Projecting the data onto the vertical coordinate, the enhancement appears and one makes the correct inference concerning its nature. Projecting the data onto the horizontal axis also reveals an enhancement in this coordinate. Inspecting the data in its full dimensionality (\( d = 2 \) in this case) reveals that these two enhancements result from the same cause and are not independent. However, with only the projected data this inference could not be made and the
experimenter would incorrectly interpret the horizontal coordinate enhancement as an additional independent effect in the data.

More generally, the non-rectangular nature of the boundaries causes spurious interrelationships to appear between measurement variables when they are projected onto lower dimensional manifolds that are not contained in the data. The preceding example shows that the problem can be serious when two-dimensional data are projected onto one dimension. The problem is even more serious when 7- or 13-dimensional data are projected onto one- or two-dimensional manifolds.

Another limitation of projections that is unique to high energy physics has to do with the symmetry properties of the probability density function required by identical particles. Probability density functions that describe particle physics interactions must be invariant to interchange of identical particles. The nature of this problem can be illustrated by again referring to an example of two-dimensional data projected onto one-dimensional manifolds. Consider a two-dimensional density, \( p(x, y) \), that is required to be invariant to the interchange of \( x \) and \( y \). If during the measurement process \( x \) and \( y \) do not happen to be treated equally (which is usually the case), then the measured density will not have the proper symmetry. There are two ways to remedy this situation. One is to symmetrize the data by using each data point twice. That is, the data point is entered as measured, and then entered again with its values of \( x \) and \( y \) permuted. Another technique is to completely asymmetricize the data. That is, the measured coordinates are always entered in order, say \( x \) always larger than \( y \). This is equivalent to folding the density about the line \( x = y \) into the lower diagonal part of the plane.

The information content of the two procedures is equivalent (even though the symmetrization procedure introduces twice as many points). That is, the data density \( p(x, y) \) is the same for both cases; only the boundaries have been changed. For those techniques that deal directly with the density, the asymmetrication procedure is preferred since it requires half the computation. However, the procedures are not equivalent if projection is used. This is due to the change in the boundaries. This is illustrated in Fig. 5 with a uniform density. In Fig. 5a, where symmetrization is used, the projections have uniform densities and the correct inference is made about the bivariate density. If asymmetrication is used, however, the projected distributions are no longer uniform but have a linear shape. Without knowing the true nature of the data in the full
dimensionality, one might incorrectly infer that this nonuniformity was a property of the data density.

In high multiplicity final states, where there tend to be several identical particles, this effect can be especially severe. Projection techniques require that symmetrization be used causing the computation to increase by \(2^m\) where \(m\) is the multiplicity of identical particles. Fully multidimensional techniques that deal directly with the data density can use asymmetrization, avoiding this increased cost.

6.1 Mapping

With projection, the choice of transformation functions, \(y = T(\vec{x})\), is usually dictated by the intuition of the researcher or suggested by some theoretical model. In addition, some techniques have been developed that use the data points themselves to suggest useful transformations. These techniques use the data point swarm in the full dimensionality to suggest those transformations or mappings to lower dimensionality that best reveal the salient features of the full dimensional data.

These mapping techniques are divided into linear and nonlinear dimension reducers. The linear methods consider only linear mapping functions

\[
y = \vec{a} \cdot \vec{x}
\]  

(118)

where the vector \(\vec{a} = \vec{a}\langle x_1, x_2, \ldots, x_N \rangle\) is determined from the full dimensional data set by some criteria.

The nonlinear methods do not define a specific transformation at all. Instead, they directly associate with each point in the full dimensional space, a point in the lower dimensional space. The points in the lower dimensional space are moved around with respect to each other until their relative positions have some relationship to the relative positions of the actual data points in the full dimensional space. This relationship is usually based on some criterion involving the mutual interpoint distances from each point to all of the others.

There are a wide variety of both linear and nonlinear mapping algorithms discussed in the literature. This section will discuss two linear methods and one nonlinear method. Most of the other methods are modifications and generalizations of those discussed here.

6.1.1 Principal components (linear factor analysis)

The most widely used mapping algorithm is principal components. The assumption with this method is that those coordinate projections that exhibit the
largest data spread are most likely to reveal interesting structure. One considers the class of transformations

$$y = \hat{a} \cdot \vec{x}$$

(119)

where $|\hat{a}| = 1$, and varies the direction, $\hat{a}$, seeking to maximize

$$V_{\hat{a}}[y] = \frac{1}{N} \sum_{i=1}^{N} (\hat{a} \cdot \vec{x}_i)^2 \cdot \left( \frac{1}{N} \sum_{i=1}^{N} \hat{a} \cdot \vec{x}_i \right)^2$$

(120)

with respect to $\hat{a}$,

$$\hat{a}^* = \max_{\hat{a}}^{-1} \left[ V_{\hat{a}}[y] \right] .$$

(121)

One can then map the data onto the solution projection via

$$y_i = \hat{a}^* \cdot \vec{x}_i .$$

(122)

If the mapping subspace is to be two-dimensional, then one can consider $\hat{a}^*$ as the transformation for the first dimension, i.e.,

$$y_i^{(1)} = \hat{a}^* \cdot \vec{x}_i$$

(123)

and consider another similar transformation for the second coordinate; that is,

$$y_i^{(2)} = \hat{b}^* \cdot \vec{x}_i$$

(124)

where $\hat{b}^*$ is the solution

$$\hat{b}^* = \max_{\hat{b}}^{-1} \left[ V_{\hat{b}}[y] \right]$$

subject to constraint

$$\hat{a}^* \cdot \hat{b}^* = 0 .$$

That is, $\hat{b}^*$ is the direction in which the data has the largest spread orthogonal to $\hat{a}^*$. In a similar manner, one could consider a third direction, $\hat{c}^*$, such that

$$\hat{a}^* \cdot \hat{c}^* = \hat{b}^* \cdot \hat{c}^* = 0$$

and

$$\hat{c}^* = \max_{\hat{c}}^{-1} \left[ V_{\hat{c}}[y] \right] ,$$

(125)

for a three-dimensional map.

Since both the criteria and transformations are linear, the directions $\hat{a}^*, \hat{b}^*, \hat{c}^*$, etc., can be explicitly solved by linear methods. First, one forms
the sample covariance matrix

\[ V_{ij} = \frac{1}{N} \sum_{k=1}^{N} x_k^{(i)} x_k^{(j)} - \left( \frac{1}{N} \sum_{k=1}^{N} x_k^{(i)} \right) \left( \frac{1}{N} \sum_{k=1}^{N} x_k^{(j)} \right) \]  \hspace{1cm} (126)

from the data. This matrix is real, symmetric, and non-negative. Thus, its eigenvectors are mutually orthogonal and its eigenvalues are all non-negative. The eigenvector associated with the largest eigenvalue corresponds to the direction \( \hat{a}^* \); the eigenvector associated with the second largest eigenvalue corresponds to \( \hat{b}^* \) and the third largest to \( \hat{c}^* \), and so on. The eigenvalues themselves are the sample variances of the data as projected onto the eigenvectors.

If the data sample is drawn from a normal distribution (Eq. 3), then the sample covariance matrix is an estimate of the true covariance matrix and rotation to the principal components as axes will cause the probability density function to completely factor

\[ p(\vec{y}) = \prod_{i=1}^{d} p_i(y_i) \]  \hspace{1cm} (127)

That is, each of the \( y_i \) are totally independent of all the others.

The principal components method is not particularly useful as a mapping technique for exploratory data analysis in particle physics since densities are seldom normal and those directions with the largest spread are seldom those with the most structure. However, it is computationally very inexpensive and usually worth trying.

6.1.2 Projection pursuit

Projection pursuit\(^{19}\) is also a linear mapping algorithm (Eq. 118) but the criteria for choosing the optimum projection, \( \hat{a}^* \), is nonlinear. Here, one directly seeks those projection axes upon which the data exhibit maximum structure. Projection pursuit maximizes a projection index of the form

\[ I(\hat{a}) = s(\hat{a}) d(\hat{a}) \]  \hspace{1cm} (128)

The first term, \( s(\hat{a}) \), measures the spread of the data, as projected onto the direction \( \hat{a} \), as with principal components. For \( s(\hat{a}) \) one takes the trimmed standard deviation from the mean

\[ s(\hat{a}) = \left[ \sum_{i=pN}^{(1-p)N} \frac{(x_i - \hat{\bar{x}}_a)^2}{(1-2p)N} \right]^{1/2} \]  \hspace{1cm} (129a)
where

$$\bar{x}_a = \frac{(1-p)N \sum_{i=pN} x_i \cdot \hat{a}}{(1-2p)} \quad (129b)$$

Here \(\bar{x}_i\) are the data points ordered on their projected values \((\bar{x}_i \cdot \hat{a})\) and \(p\) is some small fraction regarded as outliers in the projection, and thus deleted. This makes the estimate robust against extreme outliers.

The term \(d(\hat{a})\) is an average nearness function of the form

$$d(\hat{a}) = \sum_{i=1}^{N} \sum_{j=1}^{N} f(r_{ij}) \cdot I(R-r_{ij}) \quad (130)$$

where

$$r_{ij} = |(\bar{x}_i - \bar{x}_j) \cdot \hat{a}|$$

and \(I(\eta)\) is a step function

$$I(\eta) = \begin{cases} 1 & \text{if } \eta > 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, only those projected distances for which \(r_{ij} < R\) contribute to the sum. The function \(f(r)\) should be monotonically decreasing for increasing \(r\) in the region \(r < R\), reducing to zero at \(r = R\).

Projections onto two dimensions are characterized by two orthogonal directions \(\hat{a}\) and \(\hat{b}\) \((\hat{a} \cdot \hat{b} = 0)\). For this case, Eq. 129 generalizes to

$$s(\hat{a}, \hat{b}) = s(\hat{a}) \cdot s(\hat{b})$$

and \(r_{ij}\) becomes

$$r_{ij} = \left(\frac{(\bar{x}_i - \bar{x}_j) \cdot \hat{a}}{2} + \left(\frac{(\bar{x}_i - \bar{x}_j) \cdot \hat{b}}{2}\right)^2\right)^{1/2} \quad (131)$$

in Eq. 130.

The algorithm is insensitive to the explicit function form of \(f(r)\) and shows dependence only on

$$\bar{r} = \int_{0}^{R} rf(r) \, dr / \int_{0}^{R} f(r) \, dr \quad \text{(one dimension)}$$

or

$$\bar{r} = \int_{0}^{R} rf(r) \, rdr / \int_{0}^{R} f(r) \, rdr \quad \text{(two dimensions)} \quad (132)$$

The value of \(\bar{r}\) establishes the scale of density variation to which the algorithm is sensitive and thus defines the size of the structure being sought.
The projection index, \( I(\hat{a}) \) (or \( I(\hat{a}, \hat{b}) \) in two dimensions) measures the degree of structure present in the data in the particular projection. The strategy of projection pursuit is to find those projections that maximize \( I(\hat{a}) \), i.e.,

\[
\hat{a}^* = \max_{\hat{a}}^{-1}[I(\hat{a})] \quad \text{(one dimension)}
\]

\[
(\hat{a}^*, \hat{b}^*) = \max_{(\hat{a}, \hat{b})}^{-1}[I(\hat{a}, \hat{b})] \quad \text{(two dimensions)}
\]

Since the mapping criteria are not linear, as with principal components, numerical hill climbing methods are required to seek the maxima. The projection index is reasonably well behaved, however, so that only a few iterations of the maximizer are usually necessary to find a solution. Also, quite often several solutions exist providing several possible highly structured projections for inspection by the researcher.

Since projection pursuit directly seeks projections with high structure, it is potentially more useful than principal components. Figure 6a shows a projection of some particle physics data on the largest principal axis. Figure 6b shows the same data projected onto the projection pursuit solution obtained at the first local maximum of the projection index, uphill from the principal components solution. Figure 6c shows the same data projected onto the plane of the two largest principal components, while Figure 6d shows the corresponding projection pursuit solution.

Although the principal axis projections indicate possible structure within the data set, the projection pursuit solutions are clearly more revealing. This is indicated by the substantial increase in the projection index (p-index), and verified by visual inspection.

Because projection pursuit is a linear mapping algorithm, it suffers from well known limitations of linear mapping. The algorithm will have difficulty in detecting clustering about highly curved surfaces in the full dimensionality. In particular, it cannot detect nested spherical clustering. It can, however, detect nested cylindrical clustering where the cylinders have parallel generators.

6.1.3 Nonlinear mapping algorithms

With nonlinear mapping, there is no specific transformation function, \( y = T(\bar{x}) \), defined. Each projected point \( \bar{y}_i \) (usually two-dimensional) is associated with a particular data point, \( \bar{x}_i \), in the full dimensionality. The positions of the \( \bar{y}_i \) in the two-dimensional space are altered until they match as closely as possible some property of the \( \bar{x}_i \) in the full dimensional space.
Consider the nonlinear mapping algorithm of Sammon,\textsuperscript{20} as an example. Here, the property to be matched is the mutual interpoint distances. Let $D_{ij}$ be the distance between two points in the projection

$$D_{ij} = |\vec{y}_i - \vec{y}_j|$$

and $d_{ij}$ be the interpoint distance in the full dimensional data space

$$d_{ij} = |\vec{x}_i - \vec{x}_j|.$$ \hspace{1cm} (134a, 134b)

A mapping error function of the form

$$E(\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq i+1}^{N} \left[ \frac{D_{ij} - d_{ij}}{d_{ij}} \right]^2$$

is then minimized with respect to the positions of the projected points $\{\vec{y}_i\}_{i=1}^{N}$. Thus, the number of variables in the minimization is twice the number of data points. The solution positions $\{\vec{y}_i^*\}_{i=1}^{N}$ give the best two-dimensional representation of the full dimensional data with respect to the interpoint distances. The value of the mapping error at the solution gives an indication of how well the two-dimensional mapping represents the full dimensional data. A very small mapping error indicates that the data lies on a two-dimensional manifold imbedded in the full dimensional space.

There is clearly no limitation on the dimensionality of the projection subspace. A one or three-dimensional projection subspace would work just as well. Also, the values of the interpoint distances are not the only possible mapping function. Shepard and Carrol,\textsuperscript{21} for example, suggest the monotonicity of the interpoint distances as a criterion. That is, finding the mapping that best preserves the order relationship between the interpoint distances of the data points.

The principal limitation of these nonlinear mapping techniques is the computational resources they require. Memory proportional to $N^2$ is needed. Each evaluation of the mapping function requires a number of operations proportional to $N^2$. The number of variables in the search for the minimum is $D \cdot N$, where $D$ is the dimensionality of the projection subspace. For these reasons, the nonlinear mapping techniques cannot be applied conveniently to sample sizes larger than a few hundred. Other limitations of the nonlinear mapping algorithms are that the mapping cannot be summarized by a few parameters. This makes interpretation of the resulting map difficult. Also, the mapping only exists for the data set used in the analysis so that additional data cannot be identically mapped. These nonlinear mappings, however, are the most effective tool for
picturing the interrelationships between the data points. Data points that are found to have a particular relationship can be identified and isolated. Traditional analysis can then be used to determine the physical causes of the interrelationship.

6.2 A semi-parametric technique for model fitting (prism plot analysis)

In one very special case, multivariate data analysis reduces to univariate analysis. This is when the joint probability density function is completely factorable, that is

$$ p(\vec{x}) = \prod_{i=1}^{d} p_i(x_i) \quad . $$

(136)

In this case, each of the measurement variables is completely independent of the others and the d-dimensional problem reduces to d one-dimensional problems. One can then analyze each of these one-dimensional problems using standard univariate techniques.

An extension of this concept, to the case where the joint probability density function can be represented as a sum of terms, each of which is completely factorable (but not necessarily by the same variables) was originally proposed by Brau, Dao, Hondous, Pless and Singer\(^{22}\) (although not in this formalism) and later modified by Condon and Cowell\(^{23}\) and Van Hove. \(^{24}\) With this technique, the joint probability density function is assumed to have the form

$$ p(\vec{x}) = \sum_{m=1}^{M} \alpha_m f_m(\vec{x}) \quad (137a) $$

where

$$ f_m(\vec{x}) = \prod_{\ell=1}^{L} f_{\ell m}(y_{\ell m}) \quad (137b) $$

and

$$ y_{\ell m} = T_{\ell m}(\vec{x}) \quad . \quad (137c) $$

It is further assumed that the overlap between the functions

$$ \int_{R} f_m(\vec{x}) f_n(\vec{x}) \, d\vec{x} \quad (m \neq n) \quad (138) $$

is not large.

The purpose is to estimate the explicit parameters $\alpha_m$ and perhaps other parameters that may be associated with the functions $f_{\ell m}(y)$. The procedure
begins by making a first guess as to the values of the parameters. A weight for each data point corresponding to each term in Eq. 137a is constructed

$$w_m(x_i) = \frac{\alpha_m f_m(x_i)}{\sum_{n=1}^{M} \alpha_n f_n(x_i)} \tag{139}$$

Note that \(\sum_{m=1}^{M} w_m(x_i) = 1\) for each event.

For the case where the values of the parameters are the correct ones, this weight is the probability that the event is associated with the corresponding term in the sum. That is, if each term, \(f_m(x)\), in the sum is regarded as an independent density function, then \(w_m(x_i)\) is the probability that \(x_i\) was drawn from this density rather than any of the others. In addition, if a projection is made of any kinematic quantity, \(y = T(x)\), and the univariate density, \(\hat{p}_N(y)\) is estimated with each event weighted by \(w_m(x_i)\), then result will be the same as if

$$p_m(x) = \frac{f_m(x)}{\int_R f_m(x) dx} \tag{140}$$

were the true joint probability density rather than \(p(x)\) (Eq. 137). Also, the sum

$$N_m = \sum_{i=1}^{N} w_m(x_i) \tag{141}$$

will be the number of events drawn from \(p_m(x)\). This procedure allows the isolation of the contribution from each term in \(p(x)\) (Eq. 137a) to the plot of any kinematic variable.

If the first guess for the values of the various parameters is not quite correct, then the events weighted with \(w_m\) will contain contributions from all of the terms. Thus, if the experimenter has a good idea of the nature of the contribution to various kinematic variables from each of the \(f_m(x)\), he can adjust the parameters, recalculate the weights, and again plot the weighted distributions. This iterative procedure can continue until all of the distributions show consistency.

This iterative procedure is not confined to simply adjusting parameters. At any point in the iterative procedure, one can introduce new terms to the sum (Eq. 137a) or completely change the form of the parameterization of a term. Thus, as well as adjusting parameters, one can build the model iteratively. This procedure is, therefore, semi-parametric in the sense that the model need
not be completely specified in advance, and can, to some degree, be developed along the way. But at all times it must conform to the functional form of Eq. 137 (as must the true joint probability density of the data).

This technique is usually applied to resonance production in three and four particle final states where the joint probability density functions (transition matrix elements squared) tend to satisfy the restrictions of Eqs. 137 and 138. Also, the contributions to various kinematical variables from each term (resonance) is usually well known. The product of functions (Eq. 137b) usually contains a Breit-Wigner term in the appropriate invariant mass, as well as production and decay angular distributions. That is

\[ f_m(x) = BW(\mu_m) \ P(\theta_p) \ D(\Omega_d) \]

where \( \mu \) is the invariant mass, \( \theta_p \) the production angle, and \( \Omega_d \) the decay angle of the resonant particles.

As a first guess for the \( \alpha_m \), one takes the relative heights of each resonant peak in their corresponding mass plots. The first guess for the angular distributions is usually flat. Projections of each mass plot, as well as each angular distribution, are made weighting the events with the corresponding \( w_m(x_1) \). Those weighted distributions are then inspected for consistency. For example, if an \( \alpha_m \) is too small, there will be peaks in the corresponding \( \mu_m \) plots when the events are weighted by \( w_n(x_1) \), \( n \neq m \). Conversely, if \( \alpha_m \) is too large, there will be holes at the mass of the resonance when using these other weights. If the weighted angular distributions are not flat, then specific distributions can be incorporated into the model for the next iteration. After the model has been adjusted, the weights are again calculated and the weighted distributions again plotted. These distributions are inspected for further refinements of the model, and so on until the experimenter is satisfied with all of the weighted distributions. At that point, the model and fitting procedure are complete.

The various distributions weighted with each \( w_m(x_1) \) are the same as if one had a pure sample of resonance events from each channel. The \( N_m \) (Eq. 141) are estimates for the number of events produced by each resonance channel. The resulting values of the parameters are estimates for their true values.

When this procedure can be applied, it has several strong advantages. First, nowhere in the procedure are normalization integrals required. As noted earlier, a severe computational disadvantage with the maximum likelihood
technique was the requirement to evaluate complex multidimensional integrals for normalization. These integrals are usually evaluated with time consuming Monte Carlo methods. This technique completely avoids these Monte Carlo calculations.

Another advantage is the strong interactive coupling of the experimenter to the fitting procedure. At every stage in the analysis, the researcher is directly involved in adjusting the model and evaluating the results. Thus, his intuition can be utilized to help solve the problem.

There are also disadvantages to the procedure. First, is its limited range of applicability. It can only be used when the model and the data conform to the restrictions imposed by Eqs. 137-138. Like any parametric technique, its validity depends upon the truth of the a priori assumptions concerning the true joint probability density of the data. If the true data distribution does not conform to these assumptions, there is no way to discover this with the procedure. It is the factorization assumption (Eq. 137b) that allows one to use one-dimensional projections to build a multidimensional joint probability density. There is no way to tell if the factorization hypothesis is true by looking at the projections.

There are other less important statistical limitations to the procedure. First, since it is not deterministic (involves human interaction) there are no results concerning its statistical properties as an estimator. Most important, the procedure may not be consistent. Since the weight, \( w_m \left( \vec{x}_1 \right) \) (Eq. 139), involves all of the \( f_n (\vec{x}) \) terms, not just \( f_m (\vec{x}) \), a deficiency in one (or several) of the other terms can cause the projections weighted with \( w_m (\vec{x}_1) \) to be incorrect, even if \( f_m (\vec{x}) \) is correct. Thus, in principle, if a weighted projection is incorrect, there is no way to tell whether this is caused by a deficiency in the weighting channel or one of the others. It is conceivable that the experimenter might make the wrong decision and modify \( f_m (\vec{x}) \) to improve the projection when the problem was with \( f_n (\vec{x}) \), \( n \neq m \). In fact, one might be able to iterate to a consistent picture (all projections look correct) that is completely incorrect. The ability to arrive at a unique (and correct) solution depends strongly on the experimenter's skill and intuition. Thus, this procedure, like most parametric procedures, must be used with great care. Keeping in mind its restrictions, this technique is, however, very useful and powerful when it can be applied.
6.3 Generalized Nonparametric multivariate techniques

This section discusses some general nonparametric methods for dealing with multidimensional data. All of these methods use some variation on the kth nearest neighbor technique, but never for direct density estimation. As pointed out earlier, direct density estimation is very difficult in high dimensional spaces. Another problem with nonparametric multivariate techniques is that the test statistics are seldom distribution free. That is, the probability density function of the test statistic \( p_N(Y) \) is seldom independent of the underlying joint probability density distribution, \( p(\vec{X}) \), of the data. For the techniques discussed below, the test statistic distributions are nearly distribution free in that the probability density functions, \( p_N(Y) \), change very little for large differences in the underlying density distributions, \( p(\vec{X}) \), of the data. In addition, procedures are described for estimating \( p_N(Y) \) directly from the data so that for any given application the significance level of the test can be determined. In this way the tests are distribution free.

6.3.1 A nonparametric procedure for comparing multivariate point sets

Consider two samples of \( N_1 \) and \( N_2 \) observations taken on vector random variables \( \vec{X} \) and \( \vec{Y} \) with unknown joint probability density functions \( p(\vec{X}) \) and \( q(\vec{Y}) \). We wish to test the null hypothesis, \( H_0 \), that \( p(\vec{X}) = q(\vec{Y}) \) for all \( \vec{X} \) and \( \vec{Y} \).\(^{26}\)

This is the multivariate analog of the nonparametric goodness-of-fit test discussed earlier.

It is often useful in high energy physics to compare two experimental point sets to determine to what extent they are similar or different. At the most straightforward level two experiments can be compared for their compatibility. Since the test makes the comparison in the full dimensionality of the data measurement space, all of the information contained in the experiments is used.

Usually one wishes to determine if changing a property of the data has any effect or consequences on the resulting joint probability density function of the experimental measurables. This property may be external or it may be one of the measurement variables themselves. For example, an experimenter may wish to test for the presence of experimental biases in his apparatus by comparing his data to similar data taken with some of the magnet currents reversed. The null hypothesis is that there are no biases, that is, the two data sets should be the same in every way. In another application, the external property could be the spin of the beam or target. Here one would like to know if there is any property of the data that is different for different signs of the spin. Data taken
with spin up is compared to data taken with spin down. A frequent application
is energy dependence of multiparticle production. Here the data from similar
experiments, taken at several beam energies, are compared to see if there is
any energy dependence in the Lorentz invariant amplitude for the reaction.
Sometimes the varying property of the data is one of the measurement variables.
For example, in electroproduction one could test the data for a dependence on
the mass, $q^2$, of the exchanged photon.

The test described below not only gives a measure (confidence level or p-
value) for the compatibility of the two point distributions, but also gives infor-
mation as to those regions of the multidimensional space where the corre-
spondence between the point sets is good and where it is bad. This information
can give considerable insight as to the dynamical mechanism causing the point
sets to disagree.

A common application of this algorithm is in multivariate goodness-of-fit.
As discussed above, there are no general multivariate goodness-of-fit statistics.
If a Monte Carlo procedure can be used to generate data points distributed from
a density, $p(\mathbf{x})$, corresponding to some model, then this algorithm can be used
to compare the Monte Carlo data to the actual data. In this way, one can obtain
a confidence level for the model describing the data (in the full dimensionality of
the data space) as well as determining those regions of the data space where the
model gives a good description and those regions where it is poor.

This technique can also be used to design experiments. Monte Carlo data
from two different models can be compared. If the comparison results in a good
 correspondence, then there is no way the proposed experiment can distinguish
between the two models. If the correspondence is not good, then the region of
the multidimensional measurement space where the two models most disagree
can be identified. The values of the measurables corresponding to that region
can then be used to determine how to best set up the experiment to have maxi-
mum discrimination ability between the two models.

The algorithm for testing the null hypothesis, $H_0$, that two multivariate
point samples (classes) were drawn from the same unknown joint probability
density function, proceeds in the following manner. The two samples of size
$N_1$ and $N_2$ respectively, are combined into a single sample of size $N=N_1+N_2$
with each point tagged as to the class from which it originates. The closest $k$
points to each point are examined and the number, $k_1$, originating from class
one (or the corresponding number, $k_2=k-k_1$, originating from class two), is
determined. Thus, associated with each point in this combined sample is a measure of the composition of the points closest to it. The observed frequency distribution of \( k_1 \), \( n(k_1) \), for all the sample points, is recorded. This frequency distribution is then compared to that expected under the null hypothesis.

There are a variety of ways of testing whether the observed distribution for \( k_1 \) conforms to that expected under \( H_0 \). One technique involves comparing the frequency distribution of \( k_1 \), \( n_1(k_1) \), evaluated in the neighborhoods centered at class one points to the frequency distribution of \( k_1 \), \( n_2(k_1) \), evaluated in the neighborhoods centered at class two points. Under \( H_0 \), these two distributions are expected to be the same. A useful general procedure is to compare the detailed distributions of \( n_1(k_1) \) and \( n_2(k_1) \) to their expected distribution, \( n_0(k_1) \), under \( H_0 \). The problem of comparing two multivariate point distributions is, in this way, reduced to a univariate goodness-of-fit test.

If each of the \( N \) \( k \)-neighborhoods were mutually exclusive, then the relative frequency of the possible values of \( k_1 \) would (under the null hypothesis) conform to a binomial distribution over \( k_1=0, 1, 2, \ldots, k \) with probability \( p=N_1/N \); that is, \( n_0(k_1) \) would be a binomial distribution with \( k \)-degrees of freedom. These neighborhoods cannot be mutually exclusive, however, since there are \( N \) neighborhoods—each containing \( k \) points—with only \( N \) total sample points. Thus, there is no reason to expect the distribution of \( k_1 \) values to be compatible with such a binomial distribution. The precise distribution in the general case is difficult to derive, but Monte Carlo calculations for a wide variety of cases indicate very little discrepancy between the true distribution and a binomial. Thus, a difference between the two multivariate samples can be measured by comparing the distribution observed for the \( k_1 \) values with the corresponding binomial distribution. Any of the univariate goodness-of-fit tests described earlier may be used for this purpose. The test statistic, \( Y \), for comparing the two multivariate point distributions is just this univariate goodness-of-fit statistic for comparing \( n_1(k_1) \) and \( n_2(k_1) \) to the binomial distribution \( B_{N_1/N}^k(k_1) \).

Although this procedure reduces the multivariate problem to a univariate goodness-of-fit test, its test statistic distribution is not the same as when the univariate test is applied to a standard univariate problem. This is due to the lack of independence of the values of \( k_1 \) in the univariate distribution. Because the neighborhoods are not mutually exclusive but overlap considerably, the values of \( k_1 \) for neighboring events are highly correlated. These correlations cause the probability density distribution of the test statistic, \( p_N^0(Y) \), to deviate
substantially from that when the univariate sample points are all independent. This deviation usually takes the form of increased variance of \( p_N^{(0)}(Y) \). That is, the expected value of the test statistic, \( Y \), is the same as that for independent data, but the variance about that mean tends to be much larger.

As discussed in the section on univariate goodness-of-fit testing, the distribution of the test statistic under the null hypothesis, \( p_N^{(0)}(Y) \), must be known or calculatable in order for the test to be useful. Specifically, one must be able to calculate the significance level, \( \alpha(Y) \), for the experimentally obtained value of the test statistic

\[
\alpha(Y) = \int_Y^\infty p_N^{(0)}(Y') \, dY'.
\] (142)

For this test, it is possible to use a permutation procedure to estimate the significance level of the test, directly from the experimental data. This permutation test proceeds as follows: the two samples are combined and the points randomly re-assigned to the two sample classes in the original proportion, \( N_1/N_2 \); the comparison algorithm is applied to the two newly defined samples and the value of the test statistic is obtained. Repeated application of this random permutation procedure yields a series of test statistic values. The fraction of these values that are larger than the value, \( Y \), obtained for the unpermuted case is an estimate of the significance level, \( \alpha(Y) \), for the test.

The statistical properties of this test are discussed in detail elsewhere\(^{26}\) and only the results are presented here. The test is consistent. This follows from the fact that the kth nearest neighbor technique is a consistent density estimator. The test is unbiased. Even though the kth nearest neighbor density estimator is extremely biased in high dimensionality, the bias is identical under the null hypothesis for the two samples being compared, so that the comparison is unaffected and the test is unbiased. The test is extremely robust. This is because the multivariate aspect of the test uses only order statistics and no arithmetic statistics. The test has very high efficiency. This is a somewhat surprising result. For example, this nonparametric test was found to be almost as efficient as the parametric normal theory (likelihood ratio) test on normal data for differences in location (mean), and four times more efficient for differences in scale (standard deviation), with small to moderate sample sizes.

Although the permutation procedure can always be used to estimate the significance level for the test in any application, it is seldom necessary. This
is because the test statistic distribution, $p_N^{(0)}(Y)$, is remarkably independent of the underlying data density $p(\bar{X})$. Only, when the experimenter has reason to believe that this data is especially pathological, need he apply the full permutation procedure. The performance of the test is also reasonably independent of the chosen number of near neighbors, $k$, so long as it is not too small ($k \geq 10$). The test statistic distribution shows its strongest dependence on the dimensionality, $d$, of the data space for very low dimensionality. This is because the overlapping of the neighborhoods is greater for lower dimensionality. However, for higher dimensionality, this effect diminishes so that $p_N^{(0)}(Y)$ is roughly independent of the dimensionality of the data space for $d \geq 4$.

As mentioned above, it is often useful to be able to identify those regions of the multidimensional measurement space where the two point samples most disagree (or agree) in their relative densities. This algorithm assigns such an estimate to each data point in the combined sample. Those events for which $n(k_1)$ is near zero or $k$ are located in regions where the two distributions most disagree, while those points for which $n(k_1)$ is near $(N_1/N)k$ (its expected value under $H_0$) are located in regions where the agreement is best. Thus, those points for which the discrepancy is very large (or small) can be identified and isolated, and their properties can be studied independently of the rest of the sample.

Figure 7 illustrates an application of this algorithm in the study of the energy dependence of the Lorentz invariant amplitude for multiple pion production in $pp$ collisions. Here, 203 events of the reaction $pp \to pp\pi^+\pi^+\pi^-\pi^-$ at 12 GeV/c are compared to 196 events at 28 GeV/c. For this application, the test statistic was formed in the following manner; the two $k_1^{-}$-distributions, $n_1(k_1)$ and $n_2(k_1)$, were summed to a single distribution $n(k_1) = n_1(k_1) + n_2(k_1)$, and the resulting sum compared to that predicted by the null hypothesis, $n_0(k_1)$. This comparison was done using a $\chi^2$ test statistic

$$Y = \sum_{k_1=0}^{k} \left[ \frac{n(k_1) - n_0(k_1)}{n_0(k_1)} \right]^2.$$  \hspace{1cm} (143)

Figure 7 shows frequency histograms of the number of 12 GeV/c events (class 1) in a 20 event neighborhood ($k=20$) about every event in the combined sample for various coordinate combinations. Figure 7a compares the two samples in the six-dimensional subspace of scaled momentum components parallel to the incident beam. The frequency histogram, for this case, deviates considerably
from the binomial distribution (open circles) expected for the null hypothesis, indicating that these two samples differ considerably in their multidimensional shapes. Figure 7b shows the results of comparing these two samples in the 12-dimensional momentum subspace transverse to the incident beam direction. Figures 7c and 7d show the comparison in the two six-dimensional cylindrical coordinate subspaces of transverse momentum. In contrast to the scaled longitudinal momenta, the frequency histograms for these cases do not deviate significantly from the expected binomial distribution. The comparison is slightly better in the azimuthal angle subspace than the subspace of the transverse momenta squared, however, neither deviates strongly from the expected binomial distribution.

These results show that the 12-dimensional shape of the differential cross-section transverse to the beam direction, \( \frac{d^{12}\sigma}{d(P_T)} \), either independent of, or at most, varies slowly with energy for this reaction in this energy range. By contrast, the shape of the six-dimensional differential cross section parallel to the incident beam, \( \frac{d^{6}\sigma}{d\chi} \), is changing considerably; thus, the energy dependence of the dynamics manifests itself mostly, if not completely, in the longitudinal variables.

Inspection of Fig. 7a shows that the main source of disagreement in the longitudinal variables is an excess of events for high values of \( k_1 \) (15 \( \leq k_1 \leq 20 \)). This means that there is a region of the six-dimensional scaled longitudinal phase space with a strong excess of 12 GeV/c events. This can be interpreted as a dynamical production mechanism that has a substantial cross section at 12 GeV/c which becomes very small at 28 GeV/c. These events, for which 15 \( \leq k_1 \leq 20 \), can be isolated and studied separately using traditional techniques to identify the nature of this production mechanism.

Figure 8 illustrates the use of this algorithm for multidimensional goodness-of-fit testing. Here the same data is compared in the six-dimensional subspace of azimuthal angles, to models that predict no azimuthal angle dependence in the Lorentz invariant amplitude. For this purpose, 970 Monte Carlo events of the reaction pp \( \rightarrow \) pp (4\( \pi \)) at 23 GeV/c were generated according to peripheral phase space. These Monte Carlo events were compared to the data, at 23 GeV/c, in the six-dimensional azimuthal angle subspace. Figure 8 shows the results of the comparison. The frequency of data events within a 20-event neighborhood of each point in the combined sample is clearly compatible with the corresponding binomial distribution. Thus, to the statistical accuracy of this test, the
shape of $d^6 \sigma/d\phi^6$ is compatible with that predicted solely by momentum conservation. In particular, jets in the transverse plane would require the data to approximately lie on a lower dimensional manifold in this six-dimensional space and would be easily detectable.

6.3.2 Multivariate tests for independence

An important property of multivariate data is the degree to which its joint probability density function, $p(\mathbf{x})$, factors into a product of density functions, $p_j(\mathbf{x}_j)$, each defined over exclusive orthogonal subspaces of the full dimensional space. That is,

$$p(\mathbf{x}) = \prod_{j=1}^{q} p_j(\mathbf{x}_j)$$  (144)

where $\mathbf{x} = \{\mathbf{x}_j\}_{j=1}^{q}$ and $2 \leq q \leq d$. That is, $\dim(\mathbf{x}_j) < d$ and $d = \sum_{j=1}^{q} \dim(\mathbf{x}_j)$. If the joint probability density does factor in this manner, then the vectors defined over each of the $q$-different subspaces are said to be stochastically independent. That is, the distribution of points in each subspace is totally independent of the distributions in the other subspaces.

A special case of this factorization was discussed earlier, namely where $q = d$ and thus,

$$p(\mathbf{x}) = \prod_{j=1}^{d} p_j(x_j).$$  (145)

In this case, the vector components $x_j$ are said to be totally independent. One can also speak of the pairwise independence of pairs of coordinates. A pair of coordinates $(x_i, x_j)$ is said to be pairwise independent if their marginal two-dimensional joint probability density

$$p(x_i, x_j) = \int_{\mathbb{R}} p(\mathbf{x}) \prod_{k \neq i, j} dx_k$$  (146a)

factors

$$p(x_i, x_j) = p_i(x_i) p_j(x_j).$$  (146b)

It is important to keep in mind the distinction between these three types of independence. It is clear that total independence (Eq. 145) implies both pairwise independence (Eq. 146) for all pairs, and stochastic independence (Eq. 144), but pairwise independence implies neither total nor stochastic independence.

Even if all of the coordinates $(x_{1 i}, x_{1 j}, \ldots, x_{1 j})$ are pairwise independent, this does not imply that the joint probability density function, $p(\mathbf{x})$, is either totally or stochastically independent. Stochastic independence implies pairwise independence.
for those pairs where each coordinate comes from a different stochastically independent subspace. However, the converse is not true. Even if all pairs of coordinates between two subspaces are pairwise independent, this does not imply that the two subspaces are stochastically independent.

It is also important to keep in mind the distinction between a pairwise dependence or relationship between two coordinates and a correlation between them. In Statistics, a correlation is defined to be a linear dependence or relationship as measured by the linear correlation coefficient

$$C_{ij} = \frac{V_{ij}}{\sqrt{V_{ii} V_{jj}}}$$

(147)

where

$$V_{ij} = \frac{1}{N} \sum_{k=1}^{N} x_{k}^{(i)} x_{k}^{(j)} - \left[ \frac{1}{N} \sum_{k=1}^{N} x_{k}^{(i)} \right] \left[ \frac{1}{N} \sum_{k=1}^{N} x_{k}^{(j)} \right]$$

is the sample covariance matrix. The correlation coefficient can have values between ±1. A value of zero implies no correlation or linear relationship. Positive values imply a positive slope to the linear relationship while negative values imply the opposite slope.

A pair of coordinates (x₁, x₂) can have a pairwise dependence and be uncorrelated ($C_{ij} = 0$). Consider a pair of coordinates such that the data points all lie on the circumference of a circle, i.e., $x_1^2 + x_2^2 = a^2$ where $a$ is a constant. This pair of coordinates is clearly related and show a pairwise dependence; however, since this relationship is purely quadratic, they are uncorrelated as can be easily verified by calculating their correlation coefficient (Eq. 147). Therefore, lack of correlation is necessary but not sufficient to insure pairwise independence.

For the special case where $p(\vec{x})$ is a multivariate normal distribution (Eq. 3), noncorrelation implies pairwise independence and pairwise independence implies total independence. Thus, a necessary and sufficient condition for total independence is that all of the pairwise correlation coefficients (Eq. 147) be consistent with zero; or, put another way, that the sample covariance matrix be diagonal. For two subspaces to be stochastically independent, it is necessary and sufficient that the correlation coefficients for all pairs, where one coordinate comes from one subspace and the other coordinate from the other subspace, be consistent with zero. These results are due to the fact that only linear relationships are possible with multivariate normal distributions.
The correlation coefficient (Eq. 147) is the most widely used statistic for measuring pairwise dependence. In fact, the term correlation is often used interchangeably with dependence. It should be kept in mind, however, that the correlation coefficient only measures linear dependence which is seldom the total dependence. In high energy physics especially, very few multivariate distributions are normal so that a small correlation does not necessarily mean a small dependence. A general test for independence must be able to detect nonlinear relationships between coordinate pairs, and since pairwise independence does not imply stochastic independence, it must be able to detect stochastic dependence directly. In the next section, we discuss generalized tests for pairwise independence and in the following section an even more generalized test for stochastic dependence.

6.3.2.1 The mutual information measure for pairwise dependence

The mutual information measure\(^{28}\) uses the entropy of a probability density function. The entropy, \(H\), of a probability density function is defined as

\[
H = -\int_{\mathbb{R}} p(x) \log[p(x)] \, dx
\]

\[= -E_x[\log p(x)] .\] (148)

The entropy measures the spread of the distribution. For example, for a univariate normal distribution

\[
p(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2}(x^2/\sigma^2)},\] (149)

the entropy is

\[
H(\sigma) = \frac{1}{2} \left[ 1 + \log(\sqrt{2\pi}\sigma) \right] .\] (150)

This definition is related to the intuitive notion of entropy. A distribution that is very narrow tends to constrain the random variable to local regions where \(p(x)\) is large, tending to restrain the range of values taken by \(x\). A very broad distribution allows the random variable to take on many more different values, causing a "more random" distribution.

The mutual information between two coordinates \((x_1, x_j)\) is defined as

\[
M_{ij} = H_i + H_j - H_{ij} .\]
Here

\[ H_1 = - \int_\mathbb{R} p(x_i) \log[p(x_i)] dx_i \]  \hspace{1cm} (151)

and

\[ H_{ij} = - \int_\mathbb{R} p(x_i, x_j) \log[p(x_i, x_j)] dx_i dx_j \]

where \( p(x_i, x_j) \) is the marginal bivariate joint probability density defined in Eq. 146a and \( p(x_i) \) is the marginal univariate probability density

\[ p(x_i) = \int_\mathbb{R} p(\mathbf{x}) \prod_{k \neq i} dx_k \]  \hspace{1cm} (152a)

or alternatively

\[ p(x_i) = \int_\mathbb{R} p(x_i, x_j) dx_j \]  \hspace{1cm} (152b)

In all cases, \( \mathbb{R} \) symbolizes the allowed range of values for the random variables.

The mutual information can take on values in the range

\[ 0 \leq M_{ij} \leq \text{minimum} \left[ H_1, H_j \right] \]  \hspace{1cm} (153)

A normalized mutual information can be defined as

\[ m_{ij} = \frac{M_{ij}}{\text{minimum} \left[ H_1, H_j \right]} \]  \hspace{1cm} (154)

so that \( 0 \leq m_{ij} \leq 1 \). A small value of \( m_{ij} \) indicates small pairwise dependence between \( x_i \) and \( x_j \), while a value near its maximum indicates a large pairwise dependence. Unlike the correlation coefficient, however, the mutual information measure is sensitive to all types of relationships and not just linear ones. A value of \( m_{ij} \) consistent with zero indicates pairwise independence between the coordinates \( x_i \) and \( x_j \).

For nonparametric applications, the marginal probability densities \( p(x_i) \), \( p(x_j) \), and \( p(x_i, x_j) \) must be estimated from the data sample. Any of the techniques described earlier may be used for that purpose. Since these are one and two-dimensional densities, these estimates do not encounter the difficulties present with higher dimensional density estimates.

6.3.2.2 An algorithm for the direct measure of stochastic independence

In this section we discuss methods for directly testing for stochastic independence. For simplicity of discussion, we will consider the special case of \( q=2 \) in Eq. 144. Generalizations for arbitrary values of \( q \) are straightforward.
We wish to test the null hypothesis, $H_0$, that the unknown joint probability density distribution of the data can be factored into two independent probability density functions, each defined over orthogonal subspaces of the full dimensional space. That is,

$$H_0: p(\vec{x}) = p(x_1, x_2, \ldots, x_d) = p_A(x_1, x_2, \ldots, x_M) p_B(x_{M+1}, x_{M+2}, \ldots, x_d) \quad (1 \leq M < d).$$

(155)

If a set of measurables $(x_1, x_2, \ldots, x_d)$ can be found for which such a factorization occurs, there are two important consequences. First, the nature of the particular set of measurables can give considerable insight into the dynamics of the production process. Many theories of multiparticle production either make predictions concerning the factorability of the Lorentz invariant amplitude, or need to make assumptions concerning such factorability properties in order to calculate predicted experimental results. This algorithm allows one to test directly for such factorization properties.

A second important consequence is that if the subspaces are stochastically independent then the $d$-dimensional problem can be separated into an $M$-dimensional problem and an independent $(d-M)$-dimensional problem. Thus, the dimensionality has been reduced with no loss of information. Since the problems in data analysis increase dramatically with increasing dimensionality, this is always a great advantage.

The algorithm compares the interrelationships between the points in one subspace to the interrelationships in the other subspace. Specifically, the identities of the $k$ closest neighbors to each data point are found and listed separately in each of the two subspaces. For each point, these two lists are compared for coincidences. Namely, the number of data points, $k_c$, that the two lists have in common are counted. The number of such coincidences between the two subspaces is evaluated for each event. This number, $k_c$, can have values from zero to $k$.

If the two subspaces are stochastically independent (null hypothesis), then those coincidences that do occur will be totally accidental in nature. The probability distribution of the number of such accidentals can be shown to be a binomial distribution, namely,

$$P_N(k_c) = \binom{k}{k_c}$$

(156)
with expected value

$$E[k_c] = k^2/N \quad (157)$$

A test statistic can then be formed by performing a univariate goodness-of-fit test between the experimental distribution of the $k_c$ values, $n(k_c)$, obtained from all of the data points, and this binomial distribution. A departure of the experimental distribution, $n(k_c)$, from the binomial distribution indicates stochastic dependence between the two subspaces.

The binomial distribution result for the accidental rate is invariant to how the list of points associated with each sample point in the two subspaces was prepared. For example, one could form a list in each subspace of the $k$ points farthest away from the sample point. Alternatively, we could compare the list of the farthest away in one subspace to the closest in the other subspace. Under the null hypothesis, the distribution of coincidences should conform to the binomial distribution (Eq. 156) for all of these cases. The power of the test to discriminate against various classes of alternate hypotheses can be improved by forming a test statistic from a combination of our goodness-of-fit tests, shown in Table 2.

<table>
<thead>
<tr>
<th>Case</th>
<th>Subspace A</th>
<th>vs.</th>
<th>Subspace B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>k-closest</td>
<td></td>
<td>k-closest</td>
</tr>
<tr>
<td>2)</td>
<td>k-closest</td>
<td></td>
<td>k-farthest</td>
</tr>
<tr>
<td>3)</td>
<td>k-farthest</td>
<td></td>
<td>k-closest</td>
</tr>
<tr>
<td>4)</td>
<td>k-farthest</td>
<td></td>
<td>k-farthest</td>
</tr>
</tbody>
</table>

It is easy to see that for purely linear relationships (correlations) the experimental value of $k_c$ will be larger, smaller, smaller, and larger, respectively, than $E[k_c]$ (Eq. 157) for these four cases.

Any of the goodness-of-fit tests described earlier can be used for comparing each of the four $n(k_c)$ distributions to the corresponding binomial distribution (Eq. 156). Another test statistic that has proven useful is

$$Y = \sqrt{N} \sum_{i=1}^{4} \frac{(k_c^{(i)} - E[k_c])^2}{\sqrt{\hat{V}_1(k_c) + V(k_c)}} \quad (158a)$$
where
\[ \hat{V}_1(k_{c_j}) = \frac{1}{N} \sum_{j=0}^{k} \left( k_{c_j}^{(i)} - \frac{k^{(i)}}{N} \right)^2 p_N^{(i)}(k_{c_j}) \]  
(158b)
is the experimental sample variance, and \( V(k_{c_j}) \) is the variance of the predicted binomial distribution under \( H_0 \).

\[ V(k_{c_j}) = \frac{k^2}{N} \left( 1 - \frac{k}{N} \right) \]  
(158c)
The sum in Eq. 158a is over the four distributions corresponding to the cases listed in Table 2.

As was the case for comparing two multivariate points sets, there exists for this test a permutation procedure for estimating the probability density function of the test statistic, \( p_N^{(0)}(Y) \), under the null hypothesis, directly from the data. For this permutation, the identities of the data points in one of the subspaces are randomly re-assigned. The identities of the data points in the other subspace may, but need not, be given a different random re-assignment. The test for stochastic independence is applied to the re-assigned samples and a value for the test statistic obtained. Repeated application of this permutation procedure yields a series of test statistic values that closely approximate the null probability density function for the test statistic \( p_N^{(0)}(Y) \). In particular, the number of permuted test statistic values greater than the value obtained from the experimental (unpermuted) test is an estimate of the significance level for the test.

The statistical properties of this test are quite similar to those for the test that compares multivariate point sets. This is not surprising since basic to both is the kth nearest neighbor technique. Specifically, the test is consistent, unbiased, and very robust. The test is somewhat less efficient than tests using correlation coefficients when there are only linear dependencies (correlations) involved. It is also slightly less efficient than the mutual information tests when there are only pairwise dependencies in the data. It is, of course, much more efficient than either when the dependence between subspaces is not linear and is more complicated than simply pairwise dependencies. Most important, unlike the simpler tests, this test provides a necessary and sufficient test for stochastic independence.
6.4 A multivariate goodness-of-fit test

This section describes an algorithm for general multivariate goodness-of-fit testing. That is, given a mathematical model, $f(x)$, defined over the multidimensional data space, this algorithm tests the hypothesis that the true underlying data probability density distribution, $p(x)$, is compatible with $f(x)$,

$$H_0: p(x) = f(x).$$

In addition, this test does not require that $f(x)$ be normalized so that the calculation of

$$\int_{\mathbb{R}} f(x) \, dx$$

is not necessary, avoiding the computational problems of multidimensional integration. Since any goodness-of-fit test can also be used for estimation, this procedure can be used to estimate the parameters of multidimensional models. Unlike the maximum likelihood and moments methods, where the computational expense is often dominated by multidimensional integrations, this procedure is computationally very fast. However, like most goodness-of-fit tests that are used for estimation, this test has generally lower efficiency than the direct parametric estimators.

Most maximum likelihood estimates involve iterating from some starting values for the parameters to a set of solution values that maximize the likelihood function. All of these iterative schemes converge much faster to a solution, the closer the parameter starting values are to the solution values. Because it is computationally very fast, the solution values from this algorithm can be used as the starting point for a likelihood maximizer. Since this starting point will generally be very close to the maximum likelihood solution, considerable computation will be saved. Also, if at its solution this algorithm indicates a very poor goodness-of-fit, the experimenter may wish to avoid the likelihood estimate altogether since this lack-of-fit indicates that the parametric assumptions upon which the maximum likelihood technique is based are not valid (i.e., the model doesn't fit).

As discussed earlier, univariate goodness-of-fit tests are constructed by forming a dissimilarity measure between the density as predicted by the model, $f(x)$, and a nonparametric estimate of the density, $\hat{p}_N(x)$, from the data. Such a procedure is not possible in the general multivariate case because of the difficulty (discussed earlier) in nonparametric multidimensional density
estimation. This algorithm, like the multivariate algorithms discussed above, achieves its success by avoiding direct density estimation.

The procedure begins by finding the \( k \) nearest neighbors to each point in the data sample. Let \( S_k(\mathbf{x}) \) be the region in the \( d \)-dimensional space containing the \( k \) nearest neighbors to a point at \( \mathbf{x} \), and let \( V_k(\mathbf{x}) \) be the volume of this region. Consider the quantity

\[
v_k(\mathbf{x}) = \int_{S_k(\mathbf{x})} \frac{1}{f(\mathbf{x})} \ p(\mathbf{x}) \ d\mathbf{x}
\]

where \( f(\mathbf{x}) \) is the model and \( p(\mathbf{x}) \) is the true probability density function of the data. Under the null hypothesis, \( H_0 \), \( p(\mathbf{x})/f(\mathbf{x}) \) is a constant, \( C \), so that

\[
v_k^{(0)}(\mathbf{x}) = \int_{S_k(\mathbf{x})} C \ d\mathbf{x} = CV_k(\mathbf{x})
\]

The integral in Eq. 159 can be estimated by

\[
\hat{v}_k = \frac{1}{k+1} \sum_{i=0}^{k} \frac{1}{f(\mathbf{x}_i)}
\]

where the summation is over the data point located at \( \mathbf{x} \) (i=0) and the \( k \)-nearest neighbors to \( \mathbf{x} \) (i=1, k). Under \( H_0 \), the quantity \( \hat{v}_k \) should be proportional to \( V_k \) and the univariate probability density of \( \hat{v}_k \), \( p_N(\hat{v}_k) \), should be compatible with the univariate probability density of \( V_k \), \( p_N(V_k) \).

For a model independent estimate of \( V_k(\mathbf{x}) \), \( \hat{V}_k \), one can take the smallest spherical volume centered at \( \mathbf{x} \) containing the \( k \) nearest points to \( \mathbf{x} \). The two univariate probability densities \( p_N(\hat{v}_k) \) and \( p_N(\hat{V}_k) \) can then be compared using a standard univariate goodness-of-fit test. Thus, a multivariate goodness-of-fit test has again been reduced to a univariate goodness-of-fit test.

When the null hypothesis is not true \( f(\mathbf{x}) \neq p(\mathbf{x}) \), then \( v_k(\mathbf{x}) \) (Eq. 159) will not be proportional to the volume \( V_k(\mathbf{x}) \) and its variation will take on a different shape, giving rise to a different univariate probability density for \( \hat{v}_k \), \( p_N(\hat{v}_k) \). This will result in a bad correspondence between \( p_N(\hat{v}_k) \) and \( p_N(\hat{V}_k) \) in the univariate goodness-of-fit test.

The test therefore consists of finding the \( k \) nearest neighbors to each data point in the full dimensionality. The volume, \( \hat{V}_k \), of a \( d \)-dimensional sphere whose radius is the distance from the point to its \( k \)th closest neighbor is calculated. The quantity \( \hat{v}_k \) (Eq. 161) is also calculated for the data point and its
k closest neighbors. The test statistic for this multivariate goodness-of-fit test is then taken to be a univariate goodness-of-fit test statistic between the distribution of \( \hat{V}_k \) and \( \hat{v}_k \) over all of the data points.

The fact that the shape of the volume for \( \hat{V}_k \) is taken to be spherical is mainly for calculational convenience and is not essential. Any volume may be used that contains the k closest points and no others. The spherical volume works well except when the data space has boundaries that are important. That is, the data density is high near a boundary. In this case, the spherical shape will severely bias the volume estimate towards values that are much too large since a considerable fraction of the sphere will lie outside the allowed region for the data. A solution would be to use only that volume of the sphere that lay inside the allowed data space. This, however, requires a detailed knowledge of the shape of the boundaries and, except in the simplest cases, considerable computation.

Quite often, one would like to do a goodness-of-fit test or estimate parameters without being required to supply information concerning the details of the boundaries of the data space. Note that both the maximum likelihood and moments estimators require such information since these boundaries form the region of integration for the multidimensional normalization integrals.

If the density of data points tends to be small near the boundaries, then the algorithm described above, using spherical volumes, is adequate since the effect of the bias near the boundaries will not be severe. However, if this is not the case, a different shape for the volume of the k nearest neighbors is required.

A volume shape that always just contains the k nearest neighbors and never exceeds the data boundaries, no matter what their shape (so long as its convex), can be conveniently calculated for the special case where \( k=d \). In this case, the \( d+1 \) points (data point plus its \( d \) closest neighbors) can be considered to be the vertices of a \( d \)-dimensional simplex. A simplex is the simplest geometrical solid for a given dimensionality (i.e., a triangle for \( d=2 \), a tetrahedron for \( d=3 \), etc.). This simplex is, in fact, the smallest nonconcave volume that
contains these data points and no others. The volume of such a simplex is

\[
\tilde{V}_k = \text{determinant} \begin{bmatrix}
    x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d)} & 1 \\
    x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d)} & 1 \\
    \vdots & \vdots & \cdots & \vdots & \vdots \\
    x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d)} & 1 \\
    x_{d+1}^{(1)} & x_{d+1}^{(2)} & \cdots & x_{d+1}^{(d)} & 1 \\
\end{bmatrix}
\]

(162)

where \(x_i^{(j)}\) is the \(j\)th coordinate of the \(i\)th vertex point.

The simplex volume shape has several disadvantages. First, the number of nearest neighbors is constrained to be equal to the dimensionality. This is a disadvantage for very low dimensionality. However, in low dimensionality the boundary effects are considerably less severe than for high dimensionality, reducing the bias from spherical volumes. Second, the variance of the volume estimates is larger with the simplex than with the spherical volume. Also, the distribution of the test statistic, \(p_N^{(0)}(Y)\), becomes more dependent on the underlying data density, \(p(x)\), and in some cases becomes badly biased when using the simplex volume. For these reasons, spherical volumes should always be used unless boundary effects are important.

When used as a goodness-of-fit test, this algorithm is somewhat less efficient than comparing Monte Carlo events generated from the model to the data events, using the procedure for comparing multivariate point sets. Also, this algorithm does not provide as much information concerning those regions of the multidimensional space where the fit is good and where it is bad. These limitations are due to the fact that this algorithm does not require \(f(x)\) to be normalized. The loss of this information, as well as information concerning the boundaries of the data space, causes the reduction in efficiency. It also results in a great increase in computational economy. In order to generate Monte Carlo events from a model, the data space boundaries and the normalization must be either explicitly or implicitly determined. Also, such Monte Carlo's are usually computationally very expensive. This algorithm trades a loss in statistical efficiency for a great gain in computational efficiency.

As for estimation, this algorithm can form the first step in a two-step procedure for multivariate goodness-of-fit. First, this computationally fast
procedure is applied. If the result is a poor goodness-of-fit, then no further processing is necessary. If this test shows a marginal or good goodness-of-fit, then the more expensive procedure of generating Monte Carlo events from the model and comparing point sets can be applied.

Another limitation with this test associated with the lack of normalization information is that, in general, the test is not consistent. The procedure tests for goodness-of-fit of the model to the data only in regions where there are data points. It is possible that the model fits the data well in regions where there are data, but predicts large data densities in regions where there is no actual data. The test is completely insensitive to this case. The converse is not true. The test is very sensitive to the case where there is data in regions where the model predicts small or zero densities. Although it is unlikely that the model will fit the data well where the data points exist, and still not be correct, it should be kept in mind that this situation is possible. This situation is easily detected when the Monte Carlo points are compared to the data points.

Another disadvantage with this test, as compared to the Monte Carlo generation method, is that there is no analog of the permutation procedure for estimating the null distribution of the test statistic, \( p_N^{(0)}(Y) \), directly from the data. This null distribution is reasonably (but not completely) distribution free. It is usually sufficient to determine the test statistic distribution with a Monte Carlo procedure once and for all using a model that allows quick and easy Monte Carlo generation, and is at least a crude approximation to those models being tested. Since the test statistic is nearly distribution free, the null distribution obtained in this manner will serve as a good approximation for most applications.

Both this algorithm and the one that compares point distributions leave to the researcher's discretion the choice of the coordinate variables and metric, and the number of nearest neighbors, \( k \). These algorithms are reasonably insensitive to the choice of \( k \), provided that it is not too small. In order for the tests to be consistent, \( k \) should be a function of the total sample size such that

\[
\lim_{N \to \infty} k(N) = \infty, \quad \text{and} \quad \lim_{N \to \infty} \frac{k(N)}{N} = 0.
\]

Experimentation has shown that the choice of \( k \) is not important so long as \( k > 10-20 \). Clearly, \( k \) should be small compared to the total sample size, \( N \).

These algorithms are somewhat more sensitive to choice of measurement variables and metric. Unfortunately, there are no good guidelines for their choice. For infinite sample size these algorithms are clearly invariant to
changes in coordinate variables and metric since these changes simply alter the shape of the volume element containing the evaluation point. Since these volumes are infinitesimally small, their shape doesn't matter.

For finite sample sizes, however, the shape does matter. Changes in the volume shapes that result in changes of the identities of the nearest neighbors will have an effect on the performance of the algorithms. Fukunaga and Hostetler\textsuperscript{8)} show that for those data distributions that can be made spherically symmetric by a linear transformation, the optimum metric is the inverse covariance matrix of the underlying distribution, $p(\mathbf{x})$. If this covariance matrix is estimated by the data sample covariance matrix, then this is equivalent to scaling each of the coordinates so that they have equal variance along the principal axes of the data.

If one has no \textit{a priori} information concerning the data, then this is probably the best procedure. Another reasonable procedure is to simply scale the data to have equal variance along the \textit{original} measurement coordinates. On the other hand, different experimental measurement accuracy or different characteristic length of density variation can dictate unequal scales among the various coordinates. Changing the scale of a coordinate changes its relative importance in determining the goodness-of-fit. Thus, if the researcher has information as to which coordinates are most important, they should be given larger scales.

The number and specific choices of coordinate variables also affect the performance of these algorithms. Increasing the number of coordinates only improves the performance when those variables contain information concerning the hypothesis under test. In fact, coordinates that do not contain such information (noise coordinates) dilute the power of the tests. This is because these dimensions add statistical variance to the volume estimates without providing information helpful to the estimation. Even a coordinate that does contain some additional information may not help because the increase in statistical variance that it introduces hurts more than the information increase helps. The precision of these tests can be increased greatly if the researcher's knowledge and intuition lead him to a judicious choice of coordinate variables.

For goodness-of-fit testing, the model itself can be used to help choose optimum coordinate variables. Clearly those coordinates that enter directly into the model are the ones that will tend to have the most bearing on the problem. However, there may be strong dependencies in the data, not predicted by the model. In this case, choosing only coordinate variables that appear directly
in the model will dilute the power of the goodness-of-fit to discriminate against
the model.

There are other considerations that affect the best choice for coordinate
variables. For the algorithm discussed in this section only, the nature of the
data space boundaries is very important. For some variables, the data densi-
ties approach zero at the boundaries or there are no boundaries at all. For
example, since angular variables are simply periodic, they have no boundaries
or unallowed regions. Also, in multiparticle production, the limited transverse
momenta plus energy conservation usually prevent large data population near
kinematic boundaries. On the other hand, t-channel variables such as four-
momentum transfers squared usually have the highest densities near some of
their boundaries.

Choosing good coordinate variables and a good metric usually requires a
compromise among all of these considerations. Intelligent choices based on
experience and intuition can substantially improve the performance of the
algorithms, provided that these are correct. The researcher always has the
option, of course, of applying the algorithms with many different choices and,
therefore, empirically determining which are the best.

ACKNOWLEDGMENTS

Helpful discussions with William H. Rogers, Sam Steppel and John W.
Tukey are gratefully acknowledged. I would also like to acknowledge L. Van
Hove whose comment 32 "Speaking generally, one can only regret that some
people are still content to analyze data at a level of superficiality which practi-
cally guarantees in advance that no really new and instructive conclusions will
emerge," formed the motivation for a substantial portion of this work.

FOOTNOTES AND REFERENCES

1) See Reference 2 for an excellent and relatively complete discussion of the
foundations of the statistical techniques used in high energy particle
physics.

2) W. T. Eadie, D. Drijard, F. E. James, M. Roos and B. Sadoulet, Statistical
Methods in Experimental Physics, North-Holland, Amsterdam-London,
1971.

3) The smallest possible variance for an estimate is called the minimum vari-
ance bound and is related to the information content of the estimator by
the Cramer-Rao inequality (see Reference 2, Section 7.4, pp 130).
4) If the total number of counts, $N$, is itself considered a random variable, then the distribution of counts in a histogram bin is Poisson

$$p(n_1, n_2, \ldots, n_M) = \prod_{i=1}^{M} \frac{e^{-\bar{n}_i} \bar{n}_i^{n_i}}{n_i!}$$

with $N = \sum_{i=1}^{M} \hat{n}_i$, for which $E[\hat{n}_i] = \bar{n}_i$ and $V[\hat{n}_i] = \bar{n}_i$.


12) See Reference 2, Chapter 7 for derivations of these results.


14) Reference 2, pp 268-269.

15) Reference 2, pp 269-271.


25) Substituting $\alpha_m = N_m / I_m$ with

$$I_m = \int_R f_m(\bar{x}) d\bar{x}$$

into Eqs. 139 and 141, one has

$$N_m = \frac{N_m}{I_m} \sum_{i=1}^{N} \frac{f_m(\bar{x}_i)}{\sum_{n=1}^{M} N_n f_n(\bar{x}_i)/I_n} \quad (m=1,N) .$$

Condon and Cowell (Ref. 23) show that the values of $|N_m|_m=1$ that solve this set of simultaneous equations are equivalent to the maximum likelihood solutions with Eq. 137a as the model.


29) The minimum value for the mutual information estimate expected for the null hypothesis depends upon the sample size and approaches zero asymptotically.


FIG. 1a  Histogram density estimate.

FIG. 1b  Rosenblatt density estimate.
FIG. 1c  Parzen (normal kernel) density estimate.

FIG. 1d  k-th nearest neighbor density estimate.
FIG. 2a  Histogram density estimate.

FIG. 2b  Histogram density estimate (smooth superimposed).
FIG. 2c  Residuals between rootogram and smooth of rootogram.

FIG. 2d  Comparison of histogram smooth to the data density.
FIG. 3a  Traditional histogram representation.

FIG. 3b  Histogram aligned with comparison curve.
FIG. 3c  Standard rootogram representation.

FIG. 3d  Hanging rootogram.
FIG. 3e  Hanging rootogram with residuals emphasized.
FIG. 4  Example of reflections caused by non-rectangular boundaries.

FIG. 5
$pp \rightarrow pp\pi^+\pi^+\pi^-\pi^-$

(12 GeV/c) vs (28 GeV/c)

203 Events vs 196 Events

![Graphs showing distributions of $x = p_L/p_L(\text{max})$ for different quantities.](image)

$\chi^2 = 57980$

$\chi^2 = 46$

$\chi^2 = 48$

$\chi^2 = 31$

FIG. 7
$pp ightarrow pp\pi^+\pi^+\pi^-\pi^-$

$\frac{d^6\sigma}{d\phi^6} (18-28 \text{ GeV/c})$ vs "Constant"

970 Events

$\phi = \tan^{-1} \left( \frac{p_y}{p_x} \right)$

$\chi^2 = 25$

Fig. 8
ERASME - Automatic Processing of Bubble Chamber Photographs

W. Jank
CERN, Geneva

Contents

1. Introduction
   1.1 Bubble Chambers
   1.2 Event Processing Chain

2. General Design of ERASME

3. Hardware
   3.1 Scan and Measure (S/M) Unit
      3.1.1 Optical and mechanical part
      3.1.2 Analogue electronics (CRT)
      3.1.3 Digital electronics (control)
      3.1.4 Stage and film transport system
      3.1.5 Operator interface (scan table, displays, keyboards, track ball)
   3.2 Computer
      3.2.1 Main computer (DEC System 10)
      3.2.2 Control computer (PDP-11)
      3.2.3 Micro-programmed special processor (ESOP)
      3.2.4 Special interface DEC System 10 - PDP-11

4. Software
   4.1 General
   4.2 System organization
      4.2.1 Current implementation
      4.2.2 Future plans
   4.3 Application software
      4.3.1 Calibration
      4.3.2 Pattern recognition and data reduction
      4.3.3 Geometrical reconstruction

5. Status
1. Introduction

ERASME is a system for the automatic processing of photographs of bubble chamber events (ref. 1). It represents a new approach in that it fuses into a single facility the usually independent steps such as scanning and premeasuring, measuring, spatial reconstruction and rescuing.

1.1 Bubble Chambers

Before discussing the system in detail I would like to describe briefly the use of the bubble chamber in high energy physics.

The bubble chamber is an instrument which visualizes the trajectories of fast charged particles; its operation relies on the formation of tiny (0.1 - 1 mm) bubbles along the path of the particles in a superheated liquid. These bubbles provide a "track" which can be recorded in stereoscopic photographs, and show with high accuracy where the particle went. The chamber is immersed in the field of a large electromagnet to provide a means of measuring momentum from track curvature.

The use of a bubble chamber makes it possible to observe interaction of incident particles with the nuclei of the liquid (free protons in the case of the hydrogen bubble chamber). A beam of 10 - 20 particles enters the chamber, and some of these particles interact with the protons of the liquid; for the study of these "events" one needs to determine the curvature and angles in space of all the tracks at the point of interaction (fig. 1,2). Bubble chambers can take one or two photographs per accelerator pulse; as an example, the CERN 2m HBC takes 4-5 million stereopictures (3 views) per year and therefore experiments with several hundred thousand events are not uncommon with this chamber.

1.2 Event Processing Chain

Typical bubble chamber film measuring systems distinguish four consecutive phases during the processing of the film. In the first one, called the scanning phase, an operator inspects the film for interesting events; he then selects those interactions which are of interest. This selection is normally made on the basis of some rather simple topological criteria and very little help, if any, is given by a computer. Attempts to automate this phase were only successful for very simple experiments, but in general the human eye and brain have a much more efficient pattern recognition capability and are much more economic.

In the second phase, called the measuring phase, the selected events are measured by giving precisely some 10 to 20 points along each track image belonging to the event. From this one obtains the necessary information to parameterize the tracks. There are many different systems for doing these measurements, ranging from rather simple ones, which are hand operated, to fully automatic machines, which make very precise measurements at high speed.
Most of these automatic systems use a light beam which is projected onto the film and can be moved very precisely in a controlled way. The light which passes through the film is detected by a photo multiplier tube. Thus a signal is obtained when the light beam crosses the image of a bubble which can be used to trigger the read out of counters giving the bubble position. Therefore, one can convert the interesting events contained in the picture into a stream of digitizings, which can be processed by a computer. The distinction between background (noise, other tracks, etc.) and the interesting information (tracks belonging to the event) is made at several levels. At the lowest level, much of the background noise having different characteristics from tracks is removed by discrimination in hardware. Further reduction of the data has to be done by a computer or special purpose hardware, preferably in real time to minimize off-line data handling. The complex control of all the different units of one machine can again be done by either a computer or hardware. A computer is more flexible but a very fast response is needed in order to control the machine and process the forthcoming data in real time.

The way this light beam is generated gives rise to many different machines, where either spots or line elements are generated and moved by mechanical or electronic means using high intensity lamps, lasers or cathode ray tubes (CRT) as the light source.

In the third phase, the spatial reconstruction of the event is performed in order to parameterise the tracks in a spatial coordinate system and check that the measurements done in the different stereoscopic views are consistent when projected back into space. This involves the massive usage of fitting procedures and least squares techniques as well as large matrix calculation, so fast floating point arithmetic and a large memory to store the data are necessary on the computer used for these reconstruction programs.

All the programs involved up to this stage make checks for consistency while processing the measurements. Nevertheless, a certain number of events normally remain, in which one or several tracks were not measured or reconstructed properly. In order to save these events, one normally tries to rescue them in a forth phase by either passing the film through the system again or trying to patch up the event on an interactive graphics system.

After that, the measurement of an event is normally finished and data describing it completely are written onto a file for further processing. This processing includes the kinematic analysis of the interaction and the statistical evaluation of the whole data of an experiment.
2. General Design of ERASME

Any bubble chamber film measuring system has to deal with the problem of completely different computing requirements during the phases described above. In the scan phase virtually no central processor (CP) power is required but this phase lasts a significant length of time. On the other hand, during the measuring phase one needs rather little CP-power but response in real time to satisfy the time critical functions of the measuring unit. On the contrary, during the reconstruction phase, massive allocation of CP power is necessary. The rescuing is then a mixture of scanning and measuring. In addition it was felt necessary for ERASME to provide a good time sharing service for program development in parallel with production.

It was thought that the best solution for a new system would be to have a system in which all these different phases could be combined (ref.1,2); e.g. the operator scans the film for the interesting events, measures them and geometrical reconstruction can be done immediately. Whenever an event fails, one can immediately rescue it. In this way, the film has to pass through the system only once and only acceptable data from the reconstruction phase has to be output. In this way one avoids some of the problems present in traditional systems, where all these different phases have been kept separated. To pass an event through such a system may take several weeks. Data handling is very complex and management of large amounts of data has to be organised.

The ERASME system consists of several scan and measure tables (or S/M units) each controlled by a minicomputer and all connected into a medium size computer. For a few, special tasks which need very fast data processing, a micro programmed special processor (called ESOP for ERASME Special Online Processor) is placed between the S/M unit and the control computer. The operators sitting at the tables are given several means to interact efficiently with the system, such as displays, keyboards and a 'track ball'. This gives a certain structuring of the components in the system (fig. 6). The computing power can be thought of as being distributed vertically on three different levels, while the S/M units form a horizontal extension of the system. This scheme allows good matching of the CP power and real-time response required in the different parts of the system. The vertical structure is well suited for software development. Thus when one starts programming a certain algorithm, one can code it in a high level language (FORTRAN) and run and test it in the main computer. Once established, this code or part of it can be shifted one level down into the small computer, where it has to be coded in assembly language and runs more efficiently. When necessary, it can be shifted even further down into the micro programmed processor, where it runs faster again, but needs tedious coding with micro instructions. In this way an optimal balance between response time and computer usage can be achieved. It is clear that the less complex a task is and the more frequently it needs to be executed, the lower should be the level at which it operates.
3. Hardware

3.1 Scan and Measure (S/M) Unit

Each S/M-unit is made up of the following parts (fig. 3, ref. 5):

- the optical and mechanical part (fig. 4)
- the precision CRT and track detection unit
- the digitizing logic and scan control
- the stage and film transport system with its control
- the operator's table with displays, keyboards and a track ball.

3.1.1 Optical and Mechanical Part

The main structure carries a lower stage on which four filmgates and a calibration grid are mounted. This lower stage is used to move the views under either the measuring or projection channel. Within the base are mounted the condensor lens and photo multiplier tube for the measuring channel as well as the light source for the projection channel. A bridge structure over the base carries:

- the CRT-unit with coils, their mounts, shielding and four reference photomultipliers;
- the large aperture, high precision lens with a mirror to bend the axis of the measuring channel into the vertical;
- the x and y-stages which carry two lenses for the projection channel (small and large magnification);
- the first mirror of the projection channel.

A second, much larger mirror, to reflect an image of the film onto the operator's table, is suspended from the ceiling.

3.1.3 Precision CRT and Track Detection Unit

The CRT-unit is based on previous developments at CERN. It is used to generate a light spot whose position is defined at any moment by the contents of the deflection counters in the digital scan control unit. Very precise 16-bit D/A converters in the deflection control unit convert the digitally defined positions into voltages which are inputs for the deflection drivers. These are highly stable voltage-current converters that control the currents in the deflection coils. Dynamic focus and astigmatism correction is used to maintain the spot to a size of about 15 microns over the whole scanning area of the screen. The necessary correction currents are generated by the spot shape control (SSC) unit. The values for the correction at 81 points are stored in 9 x 9 matrices of potentiometers and 2-dimensional linear interpolation is used between these values.

The spot on the screen of the CRT is focussed by a large aperture high precision lens onto the film. A condensor lens projects the light which traversed the film onto a photomultiplier. The signal from this
photomultiplier is treated in the video amplifier and track detector. Inputs from four reference photomultipliers are used to compensate for the variations of the light output of the phosphor. Normalized track pulses are fed to the digitizing logic.

3.1.3 Digitizing Logic and Scan Control

The measurements are made by moving the CRT-spot rapidly along a series of straight parallel lines covering a small rectangular area. This is called a "slice scan" (fig. 5). All parameters of this slice scan, such as origin, orientation, number of lines, length of lines, line spacing, spot speed, etc. can be set by the control computer in registers of the scan control logic. The digitizing logic then generates from normalized track pulses the position and width of a hit in a local coordinate system. These values are written into the control computer's private memory by direct memory access (DMA) or are picked up directly by the microprogrammed processor.

3.1.4 Stage and Film Transport System with its Control

The film transport system has been designed for low speed only. It needs approximately one second to move the film one frame. This is adequate for the planned mode of operation, since scanning implies that the operator has to examine the film, frame by frame, spending a significant time to examine it each time the film stops. The stopping accuracy is of the order of 0.5 mm and either 50 mm unperforated film from the CERN 2m HBC or 70 mm perforated film from BEBC or Mirabelle can be used. There are three servo-controlled stages (x, y and lower) on the machine. The lower stage carries four film gates for up to four stereoscopic views and a special gate for a calibration grid, which is mounted all the time. Film transport and all stages can be controlled by the computer. In another mode of operation, the x and y-stage positions can be controlled in a non-linear fashion from the track ball. For the film transport manual control is provided for loading and unloading the film.

3.1.5 Operator's Table with Displays, Keyboards and Trackball

The operator station consists firstly of a large table, onto which the film can be projected, together with a reference mark in the form of a bright cross. A track ball allows the operator to move the projected image and to point with the light cross to different items on the picture. For convenience two different magnifications are available which can be selected by means of a button. There is a function keyboard with a numeric keyboard incorporated, as well as an alphanumeric keyboard. To inform the operator how far the measurement of an event is advanced, to show him the result and to inform him about problems found by the system, there are two storage scope displays. On these, messages with different character size in both normal and bright characters can be written as well as pictures consisting of points and vectors. Tracks found by the program can be plotted on top of digitizations, which allows the operator to check all the automatic measurements. A cursor can be connected to the track ball and its position can be read out when any keyboard button is pushed. The buttons on the functional keyboard can be individually enabled by the program and
illuminated to show the operator the available choice of actions.

3.2 Computer System

3.2.1 Main Computer (DEC System 10)

A DEC-system 10 was chosen as the central computer (fig. 6). At present it consists of a KI-10 central processor with paging hardware, 256 K 36-bit word core memory (1 μs) which can be up to 4 ways interleaved, three 5-million words disc pack drives, two magnetic tape units, three DEC-tape units and a fast line printer. To provide access for the users a line multiplexer at present connects some seven display terminals, five Teletypes and one special line per S/M unit.

3.2.2 Control Computer (PDP-11)

These minicomputers are either PDP-11/20's or PDP-11/45's with 8 K 16-bit word memory each, but no standard peripherals of their own. On the PDP-11/20's, only core memory is used, whereas on the PDP-11/45's faster MOS memory is used. To interface the different hardware controls, a special bus is connected to the standard UNIBUS of the PDP-11's.

The PDP-11 has a single bus, the UNIBUS, which connects to both memory and peripherals and allows them to be accessed in an identical way by the CP. Interrupts may be on one of four hardware levels and are directly vectored through the first 1000 bytes of memory. In this way the vector addresses and the priority on which the handler routine will run may be set by software.

3.2.3 Microprogrammed Special Processor (ESOP)

This processor, called ESOP (ERASM Special On-line Processor) was completely developed and built at CERN (fig. 7). Its initial design goal was to perform the histogramming of the individual digitizations in real time. This means it has to be fast enough to update the four quantities stored per histogram bin in the time normally taken to transfer the coordinates to the PDP-11 (about two microseconds). It consists of four main units:

i) a data memory of 256 16-bit words which can be made to appear like ordinary PDP-11 memory for communication purposes. When used by the microprocessor it has a read/write cycle time of 45 nanoseconds.

ii) an instruction unit which has its own memory of 256 words, each 48 bits wide (both memories have their own arithmetic unit for address calculations).

iii) an arithmetic unit which will perform arithmetic and logical operations on the contents of two 16-bit registers and is also able to perform logical shifts or rotates on either of these registers on its own.

iv) a multiply and divide unit which will perform 32-bit operations in about 2 microseconds.
The cycle time of the machine, except for unit four, is 70 nanoseconds. There are two main buses: a data bus which conveys all information between the data memory and the arithmetic unit and also carries all incoming and outgoing data. The instruction bus carries all control and command functions for each cycle. To obtain the necessary speed of operation the processor was designed to execute several functions concurrently; the four main units will normally all work at the same time, and the address calculations in the memory units overlap with the read or write of the previous cycle. A conditional skip or jump system makes four way branching on a single loop instruction possible. External quasi interrupt flags are included to initiate a routine or signal the end of the incoming data to be processed.

The next task to be implemented on the microprocessor is a histogram scanning routine which will calculate master points in a local coordinate system. Further tasks may be performed in this processor, such as scaling and coordinate transformation for the on-line displays or execution of short time critical routines.

Programming of this processor has to be done on a very basic level. Each instruction is made up from 48 bits which must be separately set up. As many bits have up to four or five different functions, there are numerous situations where clashes occur. To ease the programming, an instruction set, consisting of 31 mnemonic operation codes, has been defined which encompasses all functions of the processor. About half of these OP-codes are used to set up bits which control data paths, the rest normally have associated parameters to specify the way in which the data is actually processed. Each complete 48-bit instruction is then made up from sets of these OP codes. A cross-assembler running on the DEC System 10 has been written which produces 48-bit micro-code directly from programs written in these OP-codes. It also makes checks for instruction compatibility, clashes and timing.

Via the cross assembler, a set of programs have been written to test microprocessor functions and a control package has been written for the PDP-11 to load and run the microprocessor. From experience gained on the prototype, which is now able to histogram digitizings, a number of improvements and additions will be made to the second and subsequent machines.

3.2.4 Special Interface, DEC System 10/PDP-11

The PDP-11's are connected to the DEC System 10 through parallel interfaces which have been designed to meet the specific needs of the ERASME system (ref. 3). Each of these interfaces allows the connected PDP-11 to map its addressing space into DEC System 10 memory (fig. 8). This is achieved in the following way. In each interface two sets of address switches define lower boundaries of two UNIBUS address ranges (windows) to which the interface will respond. The sizes of these windows and their positions in the PDP-10 memory are defined by two registers loaded from the DEC System 10, so that the two memory windows can be set independently for each PDP-11 to point to arbitrary memory areas in the DEC System 10. One of the windows is used as ordinary read/write memory, whereas the second window appears as read-only and can be used to share code for the different PDP-11's (this code then exists only once in DEC System 10 memory). A 16-bit
PDP-11 word is packed in each half of the DEC System 10 36-bit word.

The actual communication between the two processors is done by first depositing data as task parameters in the communication window and then sending an interrupt request to each PDP-11 on any of its four bus request levels and for any vector address 0 to 7448 to trigger a specific task in the PDP-11. The PDP-11 then in turn can interrupt the DEC System 10 processor on an appropriate PI-level, provided the DEC System 10 has enabled it first.
4. Software

4.1. General

As already described, a certain structure of the system hardware was adopted which must be reflected in the software. There are several, sometimes even contradictory requirements, which have to be met by the software. For example, during the scanning phase, which is rather long, virtually no CP time is required but the operator wants a fast response on any action he takes. This means that this particular part of the program has to be available in the system in a way that it responds quickly to any requests without using the resources (e.g. CPU, memory channels) of the computers more than necessary. During the measuring phase the program typically needs only relatively little CPU time, but the program doing the pattern recognition and data reduction is usually rather big (typically some 30K words, including buffers) and has to respond in real time to the requests of the measuring machine in order to finish the measurement of an event as quickly as possible. During the reconstruction phase, a very big program (40-50 K words) needs a lot of CP time for all the floating point calculations. During the rescue phase, the load of the computer looks like a mixture of the scanning and measuring phase, where the operator looks at the results for a while and then wants to measure a single track and so on.

Because software exists on three different levels, namely the main computer, the control computer and the micro processor, one has to optimize the distribution of the code as well as the communication between the machines to achieve maximal use of the available hardware and operators. The software also has to allow the simultaneous operation of all S/M-units for different experiments requiring different modes of operation. Because of the long time scale of the whole project, one also has to allow for production and development work to go on concurrently.

4.2 System organisation

The main computer (fig. 6.) is provided with a terminal oriented time-sharing operating system which also has extensive file handling capabilities and allows a privileged user to handle his own real-time devices. A scheduler allocates the system resources such as CP, memory, channels and I/O devices. It also does swapping of programs onto discs when memory space is needed for other users. A program is brought back into memory when either space is available again or the scheduler decides to run it (ref. 4.). The CPU contains paging hardware which the current operating system (OS) uses to allocate single pages of user programs to arbitrary physical memory locations in blocks of 512 words. Every user job consists of one or two logical segments, the low and the high or shared segment. The latter can be write-protected and therefore can be shared between several users.
Most of the system software makes use of this feature. For example, there need only be one copy of a compiler working as a shared high segment for several users at the same time. Each of these has his own low segment, where all private data are kept. The OS also allows one to keep one or both segments of a user job in the same place in memory, called "locking" it in core. In hardware, communication between user programs and the OS is done by means of unimplemented operation codes, so called monitor UU0's which transfer program control to a fixed address in the monitor addressing space.

In the control computer there is no monitor in the usual sense. Each basic task is directly triggered by a vectored interrupt from the DEC System 10 and also responds to interrupts from the ESA/SM E hardware. Priorities are treated by the natural hardware interrupt levels in the PDP-11. A real time clock is used to time out hardware malfunctions and retrigger repetitive tasks. A task sequencer allows the chaining of basic tasks to perform more complex functions.

At present there are about 20 tasks implemented, ranging from very simple ones like setting up slice scan parameters or transferring title values from the DEC System 10 to rather complex ones like track segment following or fiducial finding.

4.2.1 Current implementation

To meet the special needs of the different phases described above, and to allow normal time-sharing together with production, it seemed necessary to split the production program into suitable modules, each of which is dedicated to a special function. In such a scheme, easy communication between different modules is absolutely necessary. In addition in order to avoid problems with standard software and OS updates no significant OS changes should be necessary. This was very easily done under the OS available on the main computer. By making all the different modules user jobs, it is possible to arrange communication between each of them by a shared high segment containing all data particular to one table. Control is passed from one job to another by a simple mechanism in the standard OS called "wake" and "hibernate" - UU0's. By this, a user job can simply issue a wake request to another job and hibernate itself. The scheduler in the OS passes control to the woken job, which can then attach itself to the high segment of a table and work with the data found there. When it has finished its tasks it in turn can wake the next job and hibernate.

The modules are mainly written in FORTRAN, except for a few routines either for setting up data for PDP-11 tasks, where packing has to be done, or for short tasks which run frequently. These are written in the assembler language. As all the table dependent data are contained in one high segment per table, these code modules, which are low segments, may be used for several tables. This is possible only in a sequential fashion as the current FORTRAN compiler does not produce reentrant code. This means that one module can serve different tables but only at different times.
To do this allocation of modules to the S/M units, some tables describing the present status and availability of each module have been added to the OS. This mechanism allows the setting of a "busy" bit in an interlock table when one particular module is scheduled for an S/M unit. When the module returns control the bit is cleared and the module becomes available again. In addition there are sequence tables allowing a particular S/M unit to queue for a sequence of modules. All requests for service from a single module or a sequence of modules are entered into these tables from control jobs by means of OS calls. These tables also allow the provision of several identical copies of one particular module. In case one module is already busy for another S/M unit the next free module of the same kind can be allocated to the requesting S/M unit. By these means it is possible to avoid bottlenecks in the system for either time critical modules or modules which are used for a rather long time by one particular S/M unit.

As already mentioned there is one control job per S/M unit, which determines the mode of operation of this unit. So by changing the sequence of module calls one can adapt the mode of operation easily to what is needed for the experiment being processed on this table. As the program flow is often very similar for different experiments, the operation of a module is steered by data read into the high segment private to each S/M unit. Modules particular to one experiment or to one S/M unit can be declared private for that unit so when an experiment requires special techniques a module dedicated for this can be provided. When the programmer is debugging new routines, he can just "plug in" his changed module for one S/M unit and debug it there. He also has to reload only a small part of the programs, namely the changed module, which saves time for the programmer and on the computer.

To better define the interfaces between the different modules, the HYDRA system (ref. talk at this school.6.7.), developed at CERN was used. For the pattern recognition part, only the titles-package and the memory manager were used. This allows the definition of a dynamic data structure as input to and output from every module (which is something like a processor in HYDRA terms). So the programmer can change the code in a module as much as he wants and can still use the other modules as long as he does not change these defined input and output structures. This gives him a great deal of flexibility and modularity.

The pattern recognition and data reduction part of the programs (fig. 9.) is split in this way into eight different modules, namely a control job which mainly contains the steering of the S/M unit, an initialisation job which initialises data in the high segment, and then six functional modules which are a scan job, a fiducial job, a track initialise and follow job, a vertex job, an interaction job and a job to write the output. The geometrical reconstruction program can be split into up to five different modules, where the initialisation part is always split up. The rest can be split in a point match and reconstruction a track match job and a final fit job. The functioning and the methods used in the individual parts will be discussed later on.
There is one additional job per S/M unit which controls the interface to the PDP-11 of this unit. This job, the "Link Job", which is locked in core, contains the code to service all program interrupts arriving from the PDP-11. It first initialises the interface to the PDP-11 by setting up the two relocation and protection registers which point to a data communication area (DCA) and a common code area (CCA) respectively. Then the interface is enabled to interrupt both the PDP-11 and DEC System 10 and the PDP-11 private core is loaded by putting the code read from a disc file into the DCA, where a simple, absolute loader running in the PDP-11 can pick it up and transfer it into PDP-11 private memory. In addition, it can contain the CCA loaded with special code for the PDP-11 of this unit. When a module wants to "talk" to a PDP-11, it first puts input data for the task to run in the PDP-11 into the DCA, then sends an interrupt to the appropriate interrupt vector in the PDP-11 and hibernates. The PDP-11 executes the task, puts its output into the DCA and sends back an interrupt to the DEC System 10, which activates the interface control job. This job handles the received interrupt and passes control back to the module originally requesting the execution of the task by sending a wake ULO to it. The module finds the result of the task in the DCA and can proceed to use it.

It was already mentioned that the PDP-11's can share code which is loaded in the DEC System 10 memory. This is implemented by two more jobs locked in core, one job for the PDP-11/20's and one for the PDP-11/45's. The Link Jobs for PDP-11's using this code set the interface relocation register for the CCA to point to this area in DEC System 10 memory. If a particular PDP-11 does not use this shared code, additional code can be held in a private CCA by the Link Job.

To handle the large number of jobs which are present in this sort of system when all S/M units are running, a modified version EROPS of a standard system program OPSER is used. All necessary console commands to start the total system or one particular S/M unit can be put in a file. The file is read by EROPS which then executes the commands.

4.2.2 Future Plans

It is clear that one gets a certain overhead in the OS due to the large number of jobs necessary to run all S/M units in production. In addition, the fact that the code is not properly reentrant leads to the duplication of certain parts of it, which occupy additional memory and have to be swapped in and out.

It was felt that the system could be made more efficient and easier to handle if reentrant code could be used. In fact there is a new FORTRAN-compiler available from the manufacturer which produces reentrant code, again using the two segment feature of the DEC System 10. The code and space for local data and variables are generated in such a way that they can be loaded into different segments. So, the code can be loaded into a high (pure) segment and be write protected. This segment can then be shared between several users. All the data, like constants, variables, buffers, etc. can be loaded into a low (impure) segment, different for each user. Another feature, available with the next OS release will be user demand paging which will be very useful for our application. This means that a user can himself specify at which moment he wants certain pages of his
program to stay in core or to be swapped out. Hence parts of a program (data or code) can be swapped out from memory.

If we proceed with these ideas we will reverse things with respect to what we have now. That means that all code of the pattern recognition, data reduction and reconstruction programs, which will become reentrant, can be loaded into one high segment, shared between the S/M units. All data, buffers, title values, etc. and the code which determines the flow of the program (the old control job) which are different for the individual S/M units can be loaded into a low segment, together with the DCA and the code to handle the interrupts from the PDP-11's. In addition code to be debugged can be loaded into the low segment which is private for that S/M unit. In this way there is only one job per S/M unit. If priority should be given to any particular S/M-unit, one would only have to raise the priority of the corresponding job in the DEC System 10. Buffers of sometimes considerable size, not being used through a certain phase, can be swapped by the user demand paging and the memory freed for other programs. However, every single program modification would imply reloading the whole program, which is certainly a disadvantage.

4.3 Application Software

Many of the methods and algorithms used in the ERASME system are well established and have proved effective in other systems. The programs for the pattern recognition and data reduction have been especially written for the ERASME system, whereas only modifications have been made to the CERN standard reconstruction programs to run them in an on-line fashion.

4.3.1 Calibration

The position of the spot on the screen of the CRT is not a linear function of the currents applied to the deflection coils, the main effect being a pincushion distortion. A calibration is used to remove this non-linearity from the measurements done with the CRT. This is done by measuring crosses on a very accurate glass grid with the CRT. By mapping these measurements onto the known positions the coefficients of a 5th order polynomial in x and y are calculated. A typical distribution of residuals of such a fit peaks at around one micron with a maximum value around three microns (fig. 10).

4.3.2 Pattern Recognition and Data Reduction

There is one basic method, called histogramming, which is very often used for the pattern recognition in the ERASME System. It consists of projecting a set of digitizations onto a line at a given angle which is divided into equal sections, the "bins". Counting the digitizations in all bins gives a distribution, the histogram. A "pulse" is defined as a series of consecutive bins with contents greater than a threshold and a total contents not less than a given value, the width not exceeding a given limit. This is a very powerful method of identifying a string of digitizations forming a straight line, removing most of the background. It is simple enough to run even in the ESOP.
At present the system is based on vertex guidance (fig. 11). That is a human operator feeds the positions of the primary and all secondary vertices into the machine by measuring them on the scanning table. When the machine comes to measure an event, an operator has already indicated that there is an event on the frame and where it is located. It is up to the machine now to identify and measure this event as well as possible. On ERASME the operator can also give additional help to the programs by pointing in a clear region at tracks belonging to the event (crutch point), by pointing to tracks which should be deleted if found (anticrutch point), or by giving end points of tracks. An end point means pointing to the end of a track or the place where the image of the track becomes confused with other track images. For difficult tracks the operator can even give two or more points on the table, thus defining a curve along which the track will be precisely measured later. For all these measurements the projected image of the film is moved by the track ball relative to an illuminated cross.

When the operator has finished all operations on the optical projection, the view is moved under the measurement channel by displacing the lower stage. The stopping position of this stage can be read from a precision encoder and so all the measurements in the optical channel can be transformed into the CRT-coordinate system. Firstly, the system needs to measure reference marks, so called fiducials. Because these marks are engraved in the chamber glasses or glued on the chamber walls, the positions of these marks do not change and they are known to a very great precision. They are used later to transform all measurements back into a reference plane in chamber space.

The fiducial finding program now tries to find one reference fiducial in a fairly large search region. Having found this first mark the position of the other marks can be predicted fairly well. To measure one fiducial mark, the program makes one slice scan with the scan lines approximately bisecting the larger angle between the fiducial arms. The digitizings from this scan are then histogrammed twice at the angles of both fiducial arms. A straight line fit through the digitizings falling into the nearest pulse to the prediction is then made, giving an improved angle for a second histogram. A final fit, of second order when the fiducial arms are curved, is made and the intersection of the two straight lines or parabolas give the final fiducial position. When any problems are encountered by the program, for example not enough fiducials are found, it immediately becomes interactive, showing the operator what it has found and leaving the choice to him of what to do next, such as restarting everything or measuring the missing fiducials by hand.

Having finished with the fiducials, the program proceeds to measure the tracks. From the measurements on the projection table, the approximate positions of the vertices are known and the program first tries to identify and initialize the tracks emanating from it. This is done by using a search pattern of very narrow slice scans in the form of full or partial octagon pairs around each vertex (fig. 12). The scan lines are parallel to the octagon sides so that the intersection with tracks coming from the vertex is approximately normal. The digitizings of these slice scans are histogrammed and masterpoints are calculated as the centre of gravity of the histogram pulses. Masterpoints from the two different octagons whose distance does not exceed a given value are considered to define possible track candidates if their connecting line points roughly to the vertex.
It is checked whether a track lying close to this track candidate was already followed. If not, track following is started towards the vertex and only when this track comes relatively close to the vertex is it also followed onwards. This procedure of initializing tracks can be repeated at up to five different radii so that the chance to pick up even a confused track is very much increased.

Track following consists of making slice scans at positions calculated from points already measured along the track. The digitizings from every such slice scan are then histogrammed. A centre of gravity of the digitizings in each histogram pulse is calculated and that nearest to the predicted track position is taken as a measurement and is called a masterpoint (ref. 11).

We are convinced that a very high percentage of track segments are easy to follow and do not need very sophisticated algorithms. In particular no floating point arithmetic is needed. Only when this simple track follower fails for any reason should a bigger effort be made using a more complex algorithm. Thus it is possible to implement the relatively simple track segment follower in the small control computer and only to give additional help to this by more powerful methods programmed in the main computer.

The track segment follower contains two different methods of predicting the next slice scan. As starting values two points and a direction are required. It first makes a straight line extrapolation and puts a slice scan not far from the second point. If no master point is found by this, the slice scan is shifted onwards half a slice length at a time up to four times. If this still does not give any acceptable master point, the track candidate is given up, otherwise the new master point is used to predict the next slice scan along a circle through the last points. This is repeated along the track and, with increasing confidence, the skip distance between the slice scans is increased for every new master point. The prediction along the circle is done with an iteration formula, using two points and the two tangents at these points (ref. 12). This track segment following is done until it is stopped by:

i) reaching a given end point;

ii) needing to switch the coordinate system in which the predictions are made. This is done at about $+90^\circ$ to avoid problems with tangents of angles;

iii) finding no more master points in four consecutive slice scans.

In case (i) the segment is possibly added to already existing segments and stored away as a track measurement. Case (ii) just needs switching of the coordinate system in the main computer, then track following can be resumed. Only in case (iii) is the higher level track follower in the main computer involved. One might be either at the end of the track or the track follower may have made a wrong prediction because of one or more wrong points. In either case the last points are thrown away and a circular least squares fit is made through the remaining last six points. With this an extrapolation beyond the dropped points is made and track following is reinitialised. If this does not find any new points after four slice scans, the track is terminated and stored away. With this restart feature, many
tracks are saved and also the end of a track is defined much better. Tracks can be followed through any turning angle until a given maximum azimuth or track length is reached.

After a complete track is followed, an attempt is made to remove bad measurements and to detect kinks. The program also tries to match special points given by the operator, such as charged vees, stop, end, crutch and anticrutch points to the track. Whenever a charged vee, stop or end point is matched or a kink is detected, the track is cut. When an anticrutch point is matched, the whole track is deleted, but the track parameters are kept so that the track is not reinitialized. The number of master points is reduced to a given value by weeding out points in such a way as to give as uniform a distribution along the track as possible.

If there are any unmatched crutch points left when track initialization from the octagons is finished, the program tries to initialize tracks from these points. Four complete "octagons" are made around each point and a track candidate is chosen by putting the master points from these octagons into "roads". The road taken is the one with a maximum number of master points within a minimum distance of the original point.

Having finished automatic track initialization and following, checks are made to eliminate identical tracks. This is made by sliding a parabola along a track, checking at the same time the distance of points of any other track which has approximately the same azimuth angle at the vertex. When a certain number of points is found to fall within a minimum distance, the tracks are declared to be identical and the shorter one, or the one without a crutch point is deleted.

With the remaining tracks the program tries to improve the vertex position (ref. 9). Intersection points of circle fits through master points near the vertex are put into classes. One class is formed by the set of intersection points where the distance of the points to each other does not exceed a given value. The new vertex is calculated as the average point of all points in this class including the manual measurement of the vertex. The distances of all tracks from this vertex are checked and the tracks too far from it are deleted.

The remaining tracks are now shown to the operator on one of the display units (fig. 13). He can request displays of the result at different magnifications. He can add missing tracks by giving crutch points on one of the display units. He can add one or several points to already existing tracks. He can also delete complete tracks, parts of a track or just single points. The operator has to check and, if necessary, to patch up the measurements. The program always tries to help him by writing messages or warnings on the display units.

In this way all vertices in a view and all views are measured in turn. At the end of this phase the event is presented to an on-line geometry program which performs the spatial reconstruction of the event.

4.3.3 Geometrical Reconstruction

The main tasks of the geometrical reconstruction (ref. 7) of an event are:
i) fiducial handling;
ii) point match and point reconstruction;
iii) track match and track reconstruction with the possibility of mass
dependent fits.

For each view, vertex and track, measurements must be transformed
to a reference coordinate system, defined by the expected positions of a
number of fiducial marks on a plane parallel to the film plane. The
transformation is linear and is determined by matching measurements of the
fiducial marks to their expected positions.

Given a number of views where vertices have been measured,
point match tries to find the vertex associations, i.e. to identify the
vertices on different views which are images of the same space point. Where
there are points close together it may be necessary to resort to a final
point fit in order to resolve the ambiguities which arise. For this reason,
and because it requires relatively little computing time, the final fit,
with minimization in the film plane, is performed for each space point
candidate. With the approximate coordinates of the position of a point in
space \((X_1X_2X_3)\) a least squares fit is made by minimizing, with respect to
\(X_1X_2X_3\), the sum of squares of deviations in the film reference planes
between the measurements and the projected points.

It is the task of track match to associate those images at one
vertex which correspond to the same track in space. Furthermore it should
eliminate any spurious tracks not concerned with the interaction or decay
and should make sure that the resulting set of tracks in space is unambiguous.

The solution to the problem can be rather trivial when the topology
is simple and when the measurements are clean. It may, however, become
quite complex for high multiplicity events where not all tracks have been
measured, or where some spurious tracks are present. The basic steps are
the following:

- Lists of track images to be considered are set up and crude film plane
  parameters (circle fit) are computed for each image.
- The space to film transformation near the vertex are used to derive
  candidate multiples, together with a first estimate of their space
  parameters.
- This first approximation is employed in the reconstruction of near-
  corresponding points along the first section of the track. These
  points, in turn, are used to improve the track parameters so that,
  when necessary, more near-corresponding points can be reliably computed
  further along the track.
- Ambiguities are diagnosed and resolved from the results of the near-
  corresponding point computation.

After that, a final fit to the track trajectory, usually mass
dependent, can be made. A steering processor decides which masses are
required for the track and judges the success of the fits. Track
parameters for a helix fit and the various mass fits are output.
5. **Status**

The ERASME project was started towards the end of 1970. The main computer with a KA10 central processor and 96 K words of core memory was installed by August 1971. The first S/M unit was completed in May 1972.

By now three S/M units are ready. Two of them are being used to measure film from the CERN 2m HBC and BEBC, while the third one is used for program and hardware development. The main computer has been upgraded as planned and a KI-10 central processor with 256 K words core memory was installed. The construction of SM4 and SM5 is progressing on schedule. SM4 will be completed in autumn and SM5 by the end of the year.
References


Fig. 1. Picture of CERN 2m Hydrogen Bubble Chamber.

Fig. 2. Picture of BEBC.
Fig. 3. Block Diagram of an S/M-unit.

Fig. 4. ERASME, optical and mechanical part.
Fig. 5. Slice scan.

Fig. 6. General lay-out of the ERASME-system,
Fig. 7. ESOP (ERASME Special On-line Processor).

Fig. 8. DEC System 10/PDP-11 interface address mapping.
Fig. 9. Software structure.

Fig. 10. Calibration: a) Residuals, b) Distribution of residuals.
Fig. 11. Sequence for event processing on ERASME.
Fig. 12. Track initialisation with partial octagons.

Fig. 13. Digitisings and measured tracks plotted on display unit.
SOFTWARE ENGINEERING

J.N. Buxton

University of Warwick
Coventry, Warwickshire
England

1. INTRODUCTION

This course of lectures is on the general topic of Software Engineering. It might be subtitled as a series of talks across the academic-practical user interface: an area of computing characterized not so much by an interface as by a morass of misunderstanding.

Software, in this context, is taken to mean large programs of some generality and with many users -- so it includes substantial applications programs as well as suppliers' basic software. What are the desirable properties of software? For example, generality, flexibility, reliability, efficiency, etc.; more precisely, working reliably to specification, on time and for a reasonable cost. The present state of affairs is otherwise: in many sensitive areas, software is late, expensive, unreliable and does not work to specification: this is known as the "software crisis".

The importance of the situation can be shown by quoting a few figures. Over-all software costs in the USA are estimated to be of the order of one thousand million dollars per year, or more than 1% of the Gross National Product. Some individual projects: the "man in space" project software costs from 1960-1970 were about one thousand million dollars and the development cost of OS 360 to 1969 was about two hundred million dollars. The indirect costs of late delivery and poor performance are hard to quantify, but are probably at least as great. So, the problem of how to improve software is very real and practical.

2. HISTORY

Relatively speaking, 15 or so years ago there were no big software problems: also, there was no big software. In 1961 the released software for the IBM 709 amounted to about 100K words of program. Software was built by small groups of highly-qualified people and in many cases was
the result of close, if informal, cooperation between university and industry. Looking back it seems like a "golden age" -- however, the seeds of future problems were becoming apparent. Software was normally undocumented and bugs could only be fixed by the man who built it originally. It was inflexible and normally inextensible.

A software industry was beginning to grow up and meet its first real problems. One of these was the problem of over-personalized and undocumented design in an industry which quickly became one of high mobility. Another, in a rapidly growing business, was the problem of using less well-trained staff. High level languages were regarded as the solution to this problem as well as an aid to productivity.

In the early 1960's two major steps occurred and in these steps in particular I think we can see the origins of the software crisis. The first step was the early attempts to tackle the really big problems -- for example, in aerospace and real time control. These problems often combined massive data processing with stringent real-time complexity and they greatly over-extended the technology of the day. The second big step was the introduction of the first big comprehensive machine ranges -- in particular, system 360. This set the software designers a problem two orders of magnitude harder than before -- both to handle a configuration range from tiny to elephantine and also to provide a functional range covering computing from data processing through scientific applications to the edges of real-time control.

The magnitude of the problems of software was discussed at length in the NATO conferences held in 1968 and 1969. During the last few years I think it is possible to distinguish three main lines of thought as to how software should be designed and built, which I characterize as the cottage industry, heavy engineering and applied logic approaches.

3. COTTAGE INDUSTRY

This starts from the observation that most good software in general use has been developed by small groups. Furthermore, its aims are probably limited to giving good service in rather narrow areas. At first such software often came from universities with little documentation and less support: however, this is no longer the case and much good and
well-supported work is produced, in particular by small software houses often with strong university connections. The main limitations on the cottage industry approach is that it does not help us to face the really big problems, where we must deploy massive resources to make any real impact. The cottage industry approach probably only works with projects which can be done by tightly-organized groups within less than 20 man-years.

4. **HEAVY ENGINEERING**

This leads us to the view that software designers and constructors must use the techniques of a big and complex technological industry to tackle the really big problems. This leads to considerations of, for example, the management techniques appropriate to organizations of hundreds or even thousands of engineers. Problems arise partly in control and also in recording or documentation of the work — both are severe and costly but not insuperable, and the heavy engineering approach has some outstanding successes — for example, the Apollo project.

Some suggested principles can be extracted as possible guidelines for such an industry — for example, here is a possible list:

i) Software is incompressible in time: if a project is running late it is usually disastrous to take on more people to help.

ii) Design the system to use not more than 75% of the available resources: then you have something in hand to implement design changes.

iii) Employ good staff.

iv) And keep them gaining experience on similar work — this was one of the keys to the Apollo success.

But one crucial problem is begged in this analysis — how do you actually design a big system? It seems that on the most successful examples the system is not so much designed as evolved through many iterations by a process of Darwinian selection. Unfortunately, economic pressures on the industry are often such that we cannot wait that long and so the dinosaurs go into service rather than quietly being allowed to fade away.
5. **APPLIED LOGIC**

According to this approach, software is an abstract product whose behaviour is governed by the laws of logic. The construction of software is therefore determined by the rules of applied computer science in the same way as the design of an aeroplane is bounded by the laws of materials science and of aeronautics. Unfortunately, we do not yet know all the laws, and until our theoretical knowledge has advanced further, we are limited in the size and complexity of the software projects which we should undertake. Professor Dahl's notes give a good summary of our knowledge of the rules in the structured programming field -- we do now have useful theorems to guide us, but their application is slow and arduous.

For some time we have suffered from a basic dichotomy of view as to the nature of software, whether it is an abstract product subject to logic or an engineering product subject in some way to less rigid laws. I suggest the dilemma can be resolved as follows. Software, on an ideal machine, is indeed abstract. However, a software system on a real machine, with interfaces involving people, is not fully abstract and so has some characteristics which have analogies in the indeterminacy principles of physics and in biology. So, in a sense, both parties to the argument are correct. Where possible, and in an increasing area, we need to apply rigour and proofs. Elsewhere we apply typically biological techniques, such as redundancy and rescuing a system drifting into trouble by drastic re-initialization.

6. **SYSTEMS DESIGN**

Let us now take up the basic problem -- how should one design big systems? There are no short cuts to success: design is a question of style and of the designer's ability and experience. The basic purpose of a system is to perform to its functional specification in an economic and reliable way. Our thesis is that to be economic, it must be reliable, and in designing reliable systems lies the difficulty. We can only establish reliability by testing to the full, and we can only do this if we can design systems such that we can specify an exhaustive set of test cases, small enough to be tested.
Big systems are large and complex and the basic design principle is that of "divide and rule". Divide the system into modules by the process of system design -- eventually validated by system tests. Then program the modules, for example by structured programming, validated by module tests. The problem we now face is that of how to divide the system into modules.

Modularization was first used as a managerial device to break the work of a big project up into controllable units and apparently for this purpose the details of the division are not very important. To produce a testable system, however, the splitting up is crucial. It must be done so as to minimize, to order and to make explicit the connection between the modules. The key to producing a system which can be validated is to massively reduce and to order the system connectivity.

7. DESIGN METHODOLOGY

There are no rules on how to do this, but some helpful methodological guidelines have been developed in recent years. The key-note behind these guidelines is that of hierarchical ordering as a technique, and possibly the only technique, to control complexity. The principles discussed here have hierarchical ordering as their aim: they overlap considerably and in some respects they are different ways of expressing the same ideas. An unstructured multiply-connected network is a good model for many classically designed systems -- and a hierarchical structure is orders of magnitude more accessible. The design effort needed to produce a well-ordered system may well seem at first to be much greater, but the improvements in quality is in the long run far more important.

The first idea is that of "levels of abstraction" due to Dijkstra. The modules in the system are ordered into levels in a tree-structure. The basic principles are that modules in a particular level may only call on the services of modules at lower-levels, and that resources used by modules on one level are hidden from modules at higher levels. Each level in effect specifies an "abstract machine": the lowest level is the hardware -- or the software provided by the manufacturers -- and the
highest level is an abstract machine whose functions are precisely those required by the system specification. Each intermediate level specifies one important level of abstraction in the process of converting the given hardware to the required system. How do you divide up the system functions into well-ordered abstract machines? Well, by practice and experience, and by doing it wrong a few times first!

An alternative way of looking at the problem has been given by Parnas, who proposes "information hiding" as the criterion for division into modules. The idea is that every module encapsulates a design decision which it conceals from all other modules. This is a direct attack on the problem of system flexibility -- if a design decision needs to be changed then it is the affair of only one module. All other modules communicate only by explicit function calls and parameters and so, if a module is changed, implicit effects of the change are not propagated throughout the system.

A third and composite approach is concerned with minimizing the overall connectivity as its aim. System functions are grouped into modules, and generalized where necessary, so as to produce a minimum number of connections while retaining modules of manageable size and internal structure.

The study of design methodologies is clearly in its early days, when much discussion is really about terminology rather than facts, and when few clear principles have emerged. Nevertheless, what is already available can in practice be effective and useful.

8. **THE CHIEF PROGRAMMER TEAM**

As the final stage in this series of lectures, I wish to draw together the following ideas:

i) That design is an affair of individual style and experience, like architecture, and so the ultimate responsibility for a computing system design should be in the hands of an identifiable individual.

ii) That the structure of the group building a system should reflect the structure of the system design, not the other way round.
iii) That in the vast majority of cases it is economically far more effective to produce a structured and rigorously testable design in the beginning, rather than to save on design effort and hope to debug it later.

These threads are brought together in the "chief programmer teams" experiments carried out by Harlan Mills at IBM. The best known is the NY Times Information Bank system -- when Mills undertook to build a substantial databank system for a real customer. The design was done in top-down style, specifying the total system first with "program stubs" representing subsidiary modules or sub-systems. Structured programming was used as the implementation technique. The team consisted of a nucleus with Mills as chief programmer, a back-up or reserve chief programmer, an administrator to handle contractual interfaces and a program librarian to administer the program and documentation texts. Specialist programmers were called in as required to complete the "program stubs": the analogy is with a surgical team under the control of one responsible surgeon but with specialist help as needed.

The result of the experiment was markedly successful. The system was delivered on time, working to specification and within the budget. The quality was exceptionally high and the system contained no serious errors and very few minor ones. The over-all programmer productivity was estimated to be about double that obtainable by conventional techniques.

Of course the success of one experiment is not a complete justifica-
tion for a new approach, but clearly this line is well worth further investigation. In my view, a combination of the chief programmer team organization with design methodology and structured programming is the best way to proceed at present with a software project.
BIBLIOGRAPHY


7) N. Wirth, Program development by stepwise refinement, Commun. ACM, April 1971.
INITIATION TO HYDRA

R.K. Böck(*)
CERN, Geneva

1. INTRODUCTION

1.1. About this paper

The HYDRA conventions and support programs, developed for use with analysis programs in high energy physics, have found a wide distribution. This note gives a short introduction to and the raisons d'être for the HYDRA system in a somewhat more casual style than the existing HYDRA system manual offers. Calling sequences of user routines and examples are included in the hope that this paper may also serve as a reference for the unsophisticated user. For any more detailed information, the reader is referred to the HYDRA system manual (which can be obtained from Mrs. K. Gieselman/TC).

1.2. About the implementation of HYDRA in FORTRAN

Programming languages at high level have made computers accessible to users with very little specialized training. They have also reduced the impact of developments in computer hardware or operating systems on existing user programs.

Scientific programming is predominantly done in FORTRAN as the oldest and most widely implemented high-level language. None of the alternatives, like ALGOL or PL/1, offers the same degree of portability between

(*) The many authors of HYDRA software and concepts are not responsible for this note; obviously their contributions are vital beyond the point of simple acknowledgement.
different computers for programs, or is understood by an equally high number of computer users.

HYDRA conventions are therefore tailored to the common understanding of FORTRAN and the corresponding support packages are "embedded" in FORTRAN, i.e. they are FORTRAN subroutines or functions. They are themselves mostly written in FORTRAN as defined by the American National Standards Institute or ANSI. They resort to using non-FORTRAN statements only in very limited and well defined ways, and for three different reasons:

(a) To fill in very few obvious loopholes of FORTRAN by subroutines (bit-byte-character handling, transfer addresses).

(b) To make critical subroutines more efficient by hand-coding.

(c) To achieve computer and system independence by use of switches and non-FORTRAN statements recognised by the precompilation editor PATCHY (itself a FORTRAN program).

1.3. About the problem HYDRA attempts to solve

FORTRAN contains all ingredients to express basic algorithms adequately. At the elementary level, variables, mnemonics, assignment statements with expressions and branching instructions are sufficient. Larger programs can be structured by introducing the FUNCTION or SUBROUTINE notion with formal parameters ('calling sequence') constituting the interface between calling and called program.

With growing program size, subroutines become numerous, and their logical relation may become non-trivial. The accompanying data often have to be grouped and stuctural relations between data groups have to be introduced. Also, memory efficiency dictates overlapping use of storage. To attack such problems, FORTRAN makes available only the simple hierarchical subroutine and the COMMON storage blocks organisable in arrays. The burden of organising program and data structures is left entirely to the programmer.
HYDRA can not take over the structuring of program and data. It introduces notions, though, which alleviate this burden and which allow one to express and to communicate structures more easily. Program parts can then be more readily defined in terms of the interface data and the transformations applied to them between 'input' and 'output'. Hence, program parts are more easily understood, replaced, rearranged and documented.

These problems of communication between program parts are characteristic of a multi-user environment in which many programmers may contribute to the writing of a program, and in which the program's objectives evolve with time; in short, an environment which makes the 'black-box' approach for large programs near-impossible.

In order to achieve these objectives economically, supporting HYDRA 'system routines' and user written programs make use of concepts such as data blocking and structuring, dynamic memory, program modules, and trapping. The corresponding terminology and rules together with the support package interfaces constitute a learning threshold that one has to pass to obtain access to HYDRA.

2. HYDRA CONCEPTS

2.1. Blocking and structuring of Data

The organisation of data is vital to the writing of a program. FORTRAN gives the programmer some tools, such as mnemonics, arrays and multiple subscripts. Such variable assignments are static, i.e. the corresponding memory space is reserved at load time. Programmers use data arrays often in a relation defined implicitly by using them in a related way, or by introducing relating variables like pointers. The normal FORTRAN tools are enough to solve such problems locally.

If, however, data are communicated through many stages of a program, their structuring has to be worked out with more care. Also, mnemonics for variables become a burden rather than a help for long-range communication;
they cannot be remembered any more, they clash with local variables, and they do not usually show the structural qualities of data.

2.1.1. Data Banks

HYDRA therefore makes the distinction between shortlived data, for which no rules other than FORTRAN's are needed, and longlived data, for which the following terminology and rules are applied:

(a) Data elements are single precision floating point words.

(b) Data elements are grouped according to logical affinity (e.g. all data elements describing a particle track) and stored into contiguous storage words. This area is called the data-part of a bank.

(c) The bank carries a one-word BCD identifier and is referred to by a pointer called a link. Locally one uses the identifier as a mnemonic for the integer variable containing that link.

(d) Logical relations between banks, or rather between the data entities they represent, are expressed by links which themselves are also part of a bank. The simplest link is the pointer allowing one to go from a bank to 'the next bank of the same type'.

(e) Single-bit information concerning the group of data elements and associated links in a bank is stored in a status word, which is also part of the bank. We have by convention allowed 15 bits of this word to be freely assigned a meaning by the user.

2.1.2. Data Structures

Links allow the construction of 'graphs' of any kind, in which the nodes are data banks and the (directed) edges are links. Such pictorial graph representation of data can be used to visualise the logical relation of data. Data banks interrelated in a well defined way are called a data structure.
The simplest hierarchical data structure is called a tree structure. It is so common and useful that some HYDRA system routines support it specifically. In a tree structure, banks exist at distinct levels of hierarchy, and each bank may be one of a set of identical banks, all with the same identifier and all depending in the same way from the hierarchically next higher bank. Each bank may be the starting node of one or several tree structures at lower level.

Example: High Energy Physics events are made up of vertices, every vertex has tracks associated to it. Also, to each event is associated a bank of information concerning electronic counter information to be used later. Assume the event to be a two-prong with an associated $V^\circ$.

The pictorial graph for this event information is then

![Diagram](image)

This information is structured into three levels, and vertices and tracks constitute sets of banks. HYDRA, for storage simplicity, replaces the 'fan-out' links from 'event' to the 'set of vertices' and from 'vertex' to the 'set of tracks' by one (hierarchically) downward link to one member of the set, and by one link per member joining this member to the next member of the set. The chain of links ends at the last member with a zero link.
The HYDRA representation of the above information is then

or in a frequently used shorthand allowing for several events

Associated to each bank are bank descriptions. Highly simplified they might look for our example like this:

**EV:**
- 1 data word: event number
- 3 links: EV, VX, CO
- no status bits

**VX:**
- 6 data words: X, Y, Z and errors
- 2 links: VX, TR
- 1 status bit: primary vertex

**TR:**
- 9 data words: 1/p, λ, φ, and errors and correlations
- 1 link: TR
- 3 status bits: incident, +, -

**CO:** data words only
- no links
2.1.3. **Linkage Conventions**

All links connecting data banks into a tree structure are called *structural links* in HYDRA. The example shows that they can be divided into *horizontal links*, i.e. links from one member of a bank set to the next, and *vertical links*, i.e. links from a bank to a hierarchically lower level bank, which may again be a member of a set. In any given bank there may only be one horizontal link, but any number of vertical links.

Data structures are not always of a tree type, and banks in a tree structure may, for programming convenience, make reference to banks in some other data structure. Such non-structural links are called *reference links* in HYDRA. Any number of reference links is permitted in a bank. The storage convention for links in banks is to store structural links before reference links, and to have the first structural link always reserved for the horizontal link. 'First' and 'before' are defined with respect to the addressing inside the bank, described in 2.2.1. below.

2.2. **Dynamic Memory**

HYDRA relies for storage of all data, shortlived or longlived, on a dynamic store. The dynamic store is a FORTRAN array, arbitrarily called Q or IQ (in FORTRAN word-by-word equivalence) and usually assigned to blank common.

2.2.1. **Bank Storage**

For longlived data, bank space is obtained by calling the HYDRA subroutine MQLIFT, which returns an address relative to Q. At this address room for data elements, links and status word has been provided. The address points to the status word of the bank. Data elements of the bank are referenced by adding a bias to this address, links in the bank are referenced by subtracting a bias. Thus, for instance, the 'first link' and the 'fifth data word' of the bank VX are referred to by IQ(LVX-1) and Q(LVX+5). MQLIFT will fill up the array Q dynamically starting at the high-address end.
2.2.2. Working Space

The same FORTRAN array \( Q \) is used, in its low-address part, to contain shortlived data, and this part is called working space. As its origin is not fixed to keep flexibility for the HYDRA system, it must never be referred to by the name \( Q \), but by local mnemonics. They are introduced by appending them to a COMMON statement invoked by a PATCHY macro (CDE-card) which defines \( Q \), as well as some more system-user interface variables (see para. 4).

There are two conventions to be observed in organising the working space:

(a) links, which are relocatable integer variables, must be assigned space at lower addresses than other data variables.

(b) the extent of link and data working space must be announced to the system by CALL MQWORK, whenever their limits change.

2.2.3. Dropping of banks and Garbage Collection

Working space no longer needed is released by announcing new limits to the system. Bank space is released by dropping banks: If the information contained in a bank is no longer needed, the bank is marked as dropped by setting its drop bit, reserved for this purpose in the status word of the bank. One uses CALL QDROP which can also drop entire tree structures, or CALL SBIT1 (Q(L), IQDROP). Dropped banks stay in memory until bank space and working space clash during a new request. When this happens, the HYDRA system executes a garbage collection squeezing out dropped banks and shifting live ones. This moving of banks in store will require updating of links, done automatically by the garbage collector. Those links pointing to live banks will be relocated, i.e. a bias is added to each link so that the old address with respect to \( Q \) is converted into the new one. Links pointing to dead banks will be set to zero, unless the link is structural and the dead bank has a horizontal link connection to a live bank. In this case the link will be bridged, i.e. be replaced by the
link to that live bank.

Example for bridging: (B1 and B3 are dropped)

Garbage collection is not normally under user control! The user has to be concerned only with the total extent of Q so that too frequent garbage collection and memory overflow are avoided. The obedience to storage rules for links in banks and working space ensures smooth functioning of the garbage collection mechanism.

2.3. Program Structuring and Interfacing

Program units written according to the modularity aims of HYDRA must be defined in such a manner that they may constitute the building blocks of larger programs. On the elementary programming level, FORTRAN functions and subroutines are defined by their calling sequence and by the operations they perform to transform 'input data' to 'output data'.

HYDRA attempts to facilitate the definition of higher level program units called processors by prescribing an interface definition in terms of data structures only. The definition of 'longlived' for data that are grouped and structured in banks thereby takes a meaning: Longlived data serve for inter-processor communication. Conversely working space serves only for intra-processor use.
2.4. Logical communication between processors

2.4.1. Call Banks

Processors will frequently be designed to operate on parts of existing structures, so that their 'calling sequences' must include indicative information about more complete data structures in store. This part of the processor interface is also stored in banks, and these banks are called call banks. Call banks are created and used in a way very similar to ordinary data banks. A subroutine JQBOOK is used for creating them, which routine automatically links call banks into the call bank data structure (*). Standard link names are used in processors to refer to call banks. There must be two such links: LQUP referring to the call bank generated at higher lever to call the current processor, and possibly LQDW which points to the call bank generated in the current processor for calls to lower levels.

2.4.2. Processor Calling

HYDRA processors are groups of ordinary FORTRAN subroutines: One is the primary routine which may invoke others, if there are more than one. Calling and returning conventions inside a processor are those of ordinary FORTRAN. For communication with the primary routine of the processor, a HYDRA calling and returning procedure is used passing through the system routines JQJUMP and JQBACK. This is necessary for correct handling of the call bank structure. Processors can thus easily be intercepted, for instance for overlaying. Also, the content of the call banks includes system status information and the return address, so that processors are 'reentrant', i.e. they can be interrupted whenever calling other processors, and during this interruption they may be called in a different context without destroying the information necessary for resumption of the original task.

(*) The call bank structure is a 'stack' i.e. a linear chain of banks. Its entrance link, LQUP, points to the last created call bank corresponding to the lowest level in the processor calling hierarchy. From there links point in chronological order to previously existing call banks. For every processor call (return), the stack is augmented (reduced) by a corresponding call bank.
2.4.3. **Abnormal Communication: Trap Returns**

Normal calling and returning conventions assume normal behavior of programs. In other words, they assume the invoked procedures can transform by the programmed method the input data structure into the output data structure according to the interface definition.

In many cases, programs at any level can find themselves in an abnormal situation which makes continuation impossible, or changes at least the procedures to be adopted.

Classically, signalling of such conditions is performed by including in the data interface of subroutines markers or flags which may determine the subsequent logic.

**HYDRA attempts to single out two problems in this area, thus separating more clearly data and logic interfacing of processors.**

(a) When an abnormal condition arises, the program flow can be directed back to a processor at a higher level, which has previously signalled its capability to resume operation after detection of the particular trouble. This process is called **trapping**. A trap is set by a call to a HYDRA system routine RQTRAP; signalling an abnormal condition is a call to RQTELL. Trapping may be imposed by the calling sequence of RQTELL, if the condition detecting program, which calls RQTELL, has no programmed recovery.

(b) Traps may also be requested for certain conditions during the initialisation phase, even though a recovery is foreseen in the program. To make this possible, all interesting conditions are given unique condition ID-numbers and signalled to the system by calling RQTELL, regardless whether recovery is programmed or not. All conditions occurring can thus be counted and summarised at characteristic points in the program, usually at the end of the run. For test and production runs, this provides a powerful diagnostic and checking tool. Causing a certain amount of overhead, though, condition ID's should not be used for general accounting purposes.
3. LIMITATIONS OF HYDRA

3.1. Limits of the modularity concept

HYDRA concepts provide ways for an orderly construction of large data and program structures. The pool of application programs using HYDRA which CERN is accumulating now, is, hopefully, more easily described, reconfigured and extended than programs using many different local conventions.

Processors taken out of a larger program are not, however, very general toy bricks that can be used in any place desired. HYDRA provides a means of avoiding to have pieces of a jigsaw puzzle which fit only in one unique place. For large analysis programs, though, standard and proven processor combinations are commonly used with a high-level steering program operating very much like any conventional black-box FORTRAN program. The HYDRA conventions allow to extract the desired parts of such programs and rearrange them or to reuse them in a new context, after careful checking or adaptation of data interfaces.

3.2. Programming discipline

HYDRA is an 'embedded language'; there are no compile-time checks for misuse of concepts nor for plain programming mistakes, as long as these do not contradict FORTRAN. HYDRA support routines are reasonably well protected against basic inconsistencies in the data they are given, but diagnostics occur at execution time and necessarily are not as revealing as a compiler's check list.

The common use of explicit pointers into an anonymous dynamic store necessitates programming discipline which, if violated, may lead to clobbering of the bank storage. A number of debugging aids exist, in particular the routine DQSNAP, which can, under a control of option-letters, map and dump all parts of the dynamic store. Some non-vital conventions on link names facilitate writing and reading of programs. Links to banks in the dynamic store are given variable names which are the identifier of the
bank preceded by the letter L, e.g. LVX for the link to bank VX. Links which point to a cell of the dynamic store which itself contains a link (a frequent occurrence), are given the name of the bank which that link points to, preceded by the letter K; thus, LVX = IQ (KVX). Links also known to the HYDRA system, such as LQUP which points to a processor call bank, all carry variable names beginning by LQ or KQ.

3.3. Some examples of common mistakes

- A bank is created with certain limits on the number of links and data words. The subsequent use of a bias exceeding those limits may destroy vital information in other parts of Q.

- A status word is entirely replaced by the statement IQ (L) = (expression). This is illegal, because status words contain system information. Only use CALL SBIT or CALL SBYT to enter information into status words.

- It is false to assume that dropping a bank by CALL SBIT1 (Q(L), IQDROP) removes it from the data structure. The removal and bridging happens only when garbage collection occurs. CALL QDROP, instead, does remove a bank logically from the indicated structure, but again any number of reference links to this bank continue to appear 'live'.

- Unwanted data structures or parts thereof are often not dropped upon return to higher level programs; this is a particularly frequent error for 'returns' by trapping. Trap handling therefore should be programmed only at high level and must take care of the data corresponding to this level.

- The effect of improper structuring of banks is frequently noticed by the necessity to loop in a complicated way in processors, and particularly when data are to be dropped. Banks should be part of one data structure only, and this structure should be of the tree type whenever possible.
Most HYDRA system packages must be initialised at the beginning of the run, and in a specific order. The correct order of CALL statements is that of the description (para 5) below.

On any computer installation exist various abort possibilities which should be properly trapped. Any abnormal end of program should still execute a CALL QFATAL producing a dump of the dynamic memory. This is the only way to make sure a diagnostic can be construed for an otherwise meaningless illegal situation.

4. THE HYDRA SUPPORT MATERIAL

CERN distributes HYDRA system support routines in the form of PAM-files, i.e. card images which the precompiler program PATCHY transforms into compilable material. On any given installation, HYDRA-routines will usually be made available as load libraries or modules, the details of which depend heavily on the computer system and the person(s) responsible for HYDRA.

In addition, a set of very few COMMON-, DIMENSION- and EQUIVALENCE-statements are necessary, which can be inserted, where PATCHY is available, by a PATCHY statement + CDE, Z = Q. These statements give processors access to several common system links and to the dynamic store variables Q and IQ. They must, of course, correspond to the CDE-statements used in compiling the system support routines.

For the technically interested reader, the expansion of the PATCHY macro + CDE, Z = Q is actually the following:

COMMON / QBITS / IQDROP, IQMARK, IQGO, IQGONE, IQSYS, IQCRIT
DIMENSION IQUEST(30), Q(99), IQ(99)
EQUIVALENCE (QUEST, IQUEST), (LQUSER, Q, IQ)
COMMON /* QUEST(30), LQUSER(7), LQMAIN, LQSYS(22), LQPRIV(7),
LQ1, LQ2, LQ3, LQ4, LQ5, LQ6, LQ7, LOSV, LQAN, LQDW, LQUP */
It shows how variable names for some bit positions in the status word (e.g. IQD DROP), for system links (e.g. LQUP), for user available links (LQUSER, LQPRIV) and for the dynamic store (array Q, IQ) are introduced.

The HYDRA support routines are largely independent of each other, but it is inevitable that some routines necessitate a preceding calling of others. Routines come in packages; only one of these packages (M, for memory management) is necessary, all others should be called upon only if their functions are desired. Some routines (RQTELL, QTITLE) are called both frequently and from other system routines, so that short (dummy) versions of these are supplied in case the corresponding package is not otherwise desired. Some of the packages (M-, T-, J-, R-package) must be initialised at the beginning of the program by calling a routine XQINIT (x for M, T, etc.). Routines that need no initialisation and are not part of any specific package, are called HYDRA utility routines.

5. THE MORE IMPORTANT CALLING SEQUENCES OF HYDRA

5.1. Memory Management (M-package)

(For details see paper B MQ of the HYDRA systems manual).

Example sequence: (using bank conventions of the example in 2.1. above).

+ CDE, Z = Q
  +, LINKS(24), LB, K(3), LVX, LEV
  +, FIRSTD, A(9,9), B, C(13)
  +, LASTD
  +, SPACE(4000), EOMEM
  DIMENSION MVX(4)
  DATA MVX/2HVX, 4, 3, 6/
CALL MQINIT (EOMEM)
only once at beginning of run, to initialise the memory
manager for entire dynamic store length; here, total
length is 4128 words plus an unknown number of system
links

CALL MQWORK (FIRSTD, LASTD)
whenever working space limits are changed. Here: 30 links
and 97 data words are made available

CALL MQLIFT (LVX, LEV, - 2, MVX)
a bank with the identifier VX, with 4 links, 3 of which
are structural, and with 6 data elements (words) is created.
Its address (wrt Q) is returned in LVX, i.e. address limits
are IQ(LVX - 4) to Q(LVX + 6). The meaning of LEV,-2
is the following: if LEV ≠ 0, the bank will be appended
to an already existing tree structure by the statements
IQ(LVX - 1) = IQ(LEV - 2)
IQ(LEV - 2) = LVX
If LEV = 0, such linking actions as necessary have to be
performed by the calling program.

5.2. Title reading (T-package)
(for details and some further options see paper B TQ of the HYDRA
systems manual)

Titles are groups of data, usually punched on cards, which permit to
steer certain parts of a program. These data groups are called title items
and preceded each by a card carrying

<table>
<thead>
<tr>
<th>in column</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>an asterix</td>
</tr>
<tr>
<td>&quot;</td>
<td>normally blank</td>
</tr>
<tr>
<td>&quot;</td>
<td>a Hollerith identifier</td>
</tr>
<tr>
<td>&quot;</td>
<td>the number of data words on the following card(s)</td>
</tr>
<tr>
<td>&quot;</td>
<td>The FORTRAN format under which the following card(s)</td>
</tr>
</tbody>
</table>

are to be read (brackets included).
After the last title item a card with *FINISH in col. 1 - 7 will signal the end of title items.

The initialisation of titles in memory is done by

CALL TQINIT (LUN)

with LUN logical unit number containing title items. This has to be preceded by memory initialisation (CALL MQINIT).

Reference to title items can be made by

CALL QTITLE (LTIT, ID, IFLAG)

which returns in LTIT a link to a bank containing as data elements the word-by-word information of the title item with the Hollerith identifier ID, i.e. Q (LTIT + 1) will be the first data word, etc.

IFLAG controls the action of QTITLE in case the title item can not be located: if IFLAG = 0, LTIT will be returned zero and hence the situation must be handled by the calling program. If IFLAG = 1, QTITLE will trap-exit to the R-package (see below) with condition ID 61, and the calling program will not regain control.

Example:

<table>
<thead>
<tr>
<th>Program</th>
<th>On logical unit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>CALL MQINIT (BOMEM)</td>
<td></td>
</tr>
<tr>
<td>CALL TQINIT (1)</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>CALL QTITLE (LBEAM, 4HBEAM, 0)</td>
<td></td>
</tr>
<tr>
<td>IF (LBEAM.EQ.0) GOTO 50</td>
<td></td>
</tr>
<tr>
<td>PB = Q(LBEAM + 1)</td>
<td></td>
</tr>
<tr>
<td>ELB = Q(LBEAM + 2)</td>
<td></td>
</tr>
<tr>
<td>PHIB = Q(LBEAM + 3)</td>
<td></td>
</tr>
<tr>
<td>50 ....</td>
<td></td>
</tr>
</tbody>
</table>

| * BEAM 3 (3F 5.1)                          |                  |
| 15.3 .0006 3.14                            |                  |
5.3. Jumping into and out of processors (J-package)

(For details see paper B JQB of the HYDRA system manual)

For calling (the primary routines of) processors, and for the corresponding return instruction, HYDRA system routines are called which assure the building of a call bank data structure in memory.

Example sequence:

+ CDE, Z = Q
    EXTERNAL PROC
    DIMENSION MPROC (4)
    DATA MPROC / 4HPROC, 2, 0, 1/
    ....

CALL JQINIT
    once at beginning of run, preceded by CALL MQINIT (....)

CALL JQBOOK (MPROC)
    this corresponds vaguely to a CALL MQLIFT; the created call bank will contain 2 reference links (the third word of array MPROC is meaningless) and one data word for communication between calling and called processor. The link to the created call bank will be found in LQDW.

IQ (LQDW - 1) = ....
IQ (LQDW - 2) = ....
Q (LQDW + 1) = ....
    fills in the call parameters for the processor

CALL JQJUMP (PROC)
    executes the transfer of control into PROC. Note that upon this instruction all information to be used after return must be in bank storage, i.e. ordinary data structures or the call bank. The called processor (PROC) will normally destroy the content of working space. A data saving possibility for larger amounts of data is described in the HYDRA system manual. Our
example is completed by giving the usage of call parameters from the call bank in the (called) processor PROC, and the return sequence:

SUBROUTINE PROC
+ CDE, Z = Q
  +, LNEED, ....
 +, ARG, ....
 +, LAST
CALL MQWORK (ARG, LAST)
  readjusts work space limit
LNEED = IQ (LQUP - 1)
ARG = Q (LQUP + 1)
LQUP contains now the link to the call bank, LQDW is free for calls further down
....
CALL JQBACK
  will return, restoring work space limits and LQDW, LQUP to the state prior to the execution of CALL JQJUMP (....).

5.4. Reporting and Trapping (The R-package)
(For details and further options see paper B RQ of the HYDRA system manual).

The concept of trapping as an abnormal termination of procedures has been introduced in 2.4.3. above. Let us here look at this mechanism from a different angle.

(a) Assume there are situations in a program whose occurrence deserves to be reported to a central reporting routine. We label each of these situations by a unique condition ID, a 3-digit positive integer, and report it by calling the HYDRA system routine RQTELL. At the end of a processing step, or at the end of the program run, we will then be able to print with each condition ID the associated number of occurrences.
(b) Now some of these situations may be too severe for the routine in which they occur to handle them. So we associate to the CALL RQTELL a flag with the meaning: Do not only report the fact that this condition has occurred, but take care to restart the program at the appropriate (higher level) point. In this case, RQTELL will trap, i.e. return not to the point of calling, but to a point previously indicated to it by a CALL RQTRAP.

(c) The setting of traps is under user control. Title items group condition ID-s into trap classes and any condition ID mentioned in a trap class will be trapped regardless whether the RQTELL calling sequence requests this or not. In turn any condition ID given to RQTELL with a trap request will also be trapped: if it is not initialised by appearing as part of a trap class in a title item, and by calling RQTRAP for this class, RQTELL itself will generate a new condition (ID numbers 90, 91, 92) which are always defined by default as belonging to trap class 1, and can hence be intercepted. If they are not, the run will properly abort.

Examples: I want to intercept condition IDs 612, 622 in one place called AA as class 3, every enforced trap by default class 1 in another place BB. I use the T-package to read the title item (see 5.2 above).

* RQTR 5 (5 F 10.0)
  3. 0. 0. 612. 622.

Word 1 is the class number, words 2 and 3 are zero to avoid complicating the example. Several title items with the name RQTR can be given. During initialisation, one per run, I execute

CALL RQINIT

preceded by MQINIT, TQINIT and, if applicable, JQINIT.
In the routine intercepting class 3, I execute

....
CALL RQTRAP (3)
IF (IQUEST (1). NE.0) GO TO AA

both for initialising the trap class (IQUEST (1) = 0) and for intercepting condition IDs 612, 622 (IQUEST (1) = 612 or 622). Note this is one of the uses of the array IQUEST defined by the +CDE, Z = Q card (see 4. above).

Similarly, for intercepting condition IDs requesting a trap, but not mentioned in a title item RQTR, I set the trap for default class 1 by:

....
CALL RQTRAP (1)
IF (IQUEST (1). NE.0) GO TO BB.

After executing these instructions, the behaviour of some reporting statements will be as follows:

CALL RQTELL (115, 0)

will simply be reported for accounting;

CALL RQTELL (612, 0) and
CALL RQTELL (622, 1)

will both be trapped to statement AA, with IQUEST (1) = 612 or 622;

CALL RQTELL (489, 1)

will be trapped to statement BB, with IQUEST (1) = 90 and IQUEST (2) = 489.

A list of condition IDs in use by the HYDRA system is given in paper COL1 of the HYDRA system manual. A list of condition IDs in application programs is part of the HYDRA application manual. IDs less than 100 are reserved for system use.
Finally,

CALL RQEND

will print a summary of the condition IDs that have occurred.

5.5. Utilities

Some HYDRA system routines have been specifically made available to allow handling of data structures, dumping, termination etc. Also some routines of what we call the 'General Section' are frequently used in HYDRA programs. The following examples are an introduction to such utilities. More details and other utilities specifically available to HYDRA programs like histogramming, two-dimensional plotting, the S-package for production accounting, must be looked up in the HYDRA system manual.

(a) Examples of dropping, referring to the data structure in 2.1. above.

CALL QDROP (LEV - 2, 0)

will remove, i.e. drop and bridge the first VX-bank with its attached TK-banks from the data structure;

LVX = I(Q(LEV - 2)
CALL QDROP (LVX - 1, 1)

will remove all VX-banks, but not the first, and the associated TK-banks. The first call parameter is a K-address (see 3.2. above), the second is a flag to indicate whether the horizontal link should be followed (= 1) or not (= 0) for dropping.

LVX = I(Q(LEV - 2)
CALL QTTOUCH (I(QDROP, LVX, LHS)

will drop the first VX-bank and its attached TK-banks without bridging. I(Q (LEV - 2) will now point to a dropped bank. The Hollerith S is an option to set the required bit I(QDROP in the status word of the
Starting bank, whose L-address (LVX) is given.

\[ LVX = IQ(LEV - 2) \]

CALL QTTOUCH (IQDROP, LVX, 'SH.')

will drop all VX-banks and associated TK-banks. The Hollerith H is the option to follow the Horizontal link.

CALL SBIT1 (Q (LVX), IQDROP)

will set the bit IQDROP in the status word of the VX-bank. For a single bank to be dropped, this is the most economic way. IQDROP is the position of the drop bit in the status word and present by the card:
+ CDE, Z = Q

(b) Examples for printing dynamic memory contents for debugging purposes.

CALL DQSNAP ('FIRST', 'LM.' )

will output Links (system links and working space links) and a Map entry (address, ID) of each bank. 'FIRST' is a text associated to this print.

CALL QTTOUCH (IQCRIT, LVX, 'S.' )
CALL DQSNAP ('SECOND', 'WEMCV.' )

will dump the entire Working space, links and data, an Extended Map (address, ID and links) of each bank, a full dump of banks with bit IQCRIT set (here the sub-structure LVX), and in _variable format.

Other options for the second call parameter:

F : all live banks, full dump
O : octal format for data

(c) Examples using bit/byte handling routines.

CALL SBIT9 (Q (LVX), IQCRIT)
sets bit IQCRIT in the status word of bank VX to zero.

CALL SBIT1 (Q (LVX), IQDROP)

sets bit IQDROP to one.

CALL SBIT (N, Q(LVX), IQCRIT)

sets the same bit to N (0 or 1). Whilst IQCRIT and IQDROP are bit positions in the status word used for system - system and user - system communication, the following instruction tests the user bit 3 (bits 1 to 15 have a user assigned meaning):

IF (JBIT (Q (LVX), 3).EQ.0) ....

To handle a group of several bits (a byte), the corresponding routines exist:

CALL SBYT (3, A, 5, 4)

will set a 4-bit-byte starting at position 5 of word A to the value 3; bits 1 to 4, and 9 to maximum (15 in the status word, 32 in any other packing context) will be unchanged.

N = JBYT (A, 5, 4)

extracts the same byte in an obvious way.
PROGRAMMING DISCIPLINE

O.J. Dahl
Institute of Mathematics, Univ. Oslo, Norway

Abstract.
Good programming discipline is to produce programs which are:
easy to use and to understand, reliable and easy to debug (if not
already correct), and easy to adapt to changes in the environment.
In order to fulfill these requirements programs must well structured
and well documented. Research on techniques for program correctness
proofs has shed some light on what good structure and adequate document-
tation is. Indeed a program easily proved correct is easy to under-
stand, and vice versa.
Programming language features and certain mental techniques are aides
to produce well structured programs. Discipline is required to obtain
good documentation. The latter is even more important.

Short Summary of Proof Techniques.
The idea of program correctness and program proofs used here are
those introduced by R.W. Floyd and C.A.R. Hoare. Thus conditional
correctness means that a program behaves as specified provided
that it terminates properly. The notation
\[
\{P\} \ S \ \{Q\}
\]
where P and Q are logical assertions about program variables (and
possibly auxiliary variables), and S is a program statement or
statement sequence, means that Q is true immediately after an
execution of S, provided P is true immediately before, given
that the execution terminates.

\{P\} is called a precondition of S and \{Q\} a postassertion. When
embedded in a larger program text an assertion \{R\} specifies the
truth of R at that particular program point, in general provided
that the precondition of the program was valid on program entry.
The following are rules valid for proving the validity of program
assertions.

Simple assignment. \(\{P^x_e\} \ x := e\{P\}\)
holds for arbitrary P, where \(P^x_e\) is obtained from P by textual
substitution of e for every free occurrence of x.
Concatenation.

\{P\} S_1\{Q\} \text{ and } \{Q\} S_2\{R\} \text{ gives } \{P\} S_1; S_2\{R\}

Logical Consequence.

\text{P} \implies \text{Q} \text{ and } \{Q\} S \{R\} \text{ gives } \{P\} S \{R\},
\text{Q} \implies \text{R} \text{ and } \{P\} S \{Q\} \text{ gives } \{P\} S \{R\}.

Conditional.

\{P \land B\} S_1\{R\} \text{ and } \{P \land \neg B\} S_2\{R\} \text{ gives } \{P\} \textbf{if } B \text{ then } S_1 \text{ else } S_2 \{R\}

alternatively
\{P_1\} S_1\{R\} \text{ and } \{P_2\} S_2\{R\} \text{ gives } \{P_1 \land B \lor P_2 \land \neg B\} \textbf{if } B \text{ then } S_1 \text{ else } S_2 \{R\}.

Free loop.

\{P\} S_1\{Q\} \text{ and } \{Q \land B\} S_2\{P\} \text{ gives } \{P\} \textbf{loop: } S_1 \textbf{ while } B: S_2 \textbf{ repeat } \{Q \land \neg B\}.

Note: This rule must in general be supplemented with additional reasoning in order to prove termination. A sufficient proof might be that a specified integer valued function f of program variables decreases during each execution of S_1 followed by S_2, and Q \land B \implies f \geq 0. Note also that P or Q (the "loop invariant") cannot in general be constructed from the program text alone, but must be provided as additional information.

Example.

\{\textbf{real} x = 1 \land \textbf{real} y = a \land \textbf{integer} d = b \geq 0\}

\textbf{loop: } \{x \cdot y^d = a^b \land d \geq 0\}
\textbf{ if } \text{ odd } (d) \text{ then } x := x \cdot y \textbf{ fi } ;
\textbf{ d := } d - 2 ;
\textbf{ while } d \neq 0 : \{d \text{ decreases}\}
\textbf{ y := } y^2 ;
\textbf{ repeat: }
\{x = a^b \land d = 0\}
for-loop.

Assume that $S$ does not change any of $k,a,b$. Then

$\{a \leq k \leq b \wedge R_k \} \ S \ (\{R\})$ gives $\{R_k^{a-1}\}$ for $k := a$ to $b$: $S$ repeat $\{R_k^c\}$,

where $c = \max(a-1,b)$.

This rule follows by applying the free loop rule to the program

$k := a$; loop while $k \leq b$; $S$; $k := k+1$ repeat

choosing $R_k^{a-1} \wedge k \leq c+1$ for $P$ and $Q$ of the free loop rule.

Termination is proved by considering the function $c + 1 - k$.

Subscripted assignment.

Given $\langle \text{type} \rangle$ array $a [m:n]$; $m$, $n$ constant, then

$\{m \leq k \leq n \wedge P_a^{b}(k|e)\} \ a[k] := e[P]$

holds for arbitrary $P$, where $a(k|e)$ stands for the array value obtained by the assignment

$\forall i \ (m \leq i \leq n \Rightarrow a(k|e)[i] = \begin{cases} \text{if } i=k \text{ then } e \text{ else } a[i] \end{cases}$.

The alternative notation $(a[m:k-1], e, a[k+1:n])$ is sometimes useful.

Aggregation of operations.

Let the statement (-list) $S$ contain assignments to the variables $v_1, v_2, ..., v_n$ (only). Then functions $f_1, f_2, ..., f_n$ of program variables $w$ accessed in $S$ exist, such that

$\{P\} \ S \ (\{Q\})$ gives $\{Q_{f_1(w)}, f_2(w), ..., f_n(w)\} \ S \ (\{Q\})$

where $\forall w (P \Rightarrow Q_{f_1(w)}, f_2(w), ..., f_n(w))$ holds. The latter formula expresses what is known about the functions $f_i$ (apart from the fact that they exist).

This important rule allows us to view the total effect of a section of program as a simultaneous assignment of new values to the variables which are (or may be) altered.

Example.

Given the operation $\text{swap}(x,y)$ which satisfies $\{R^{x,y}_{\text{swap}(x,y)}\} \ (\{R\})$

for arbitrary $R$. Consider the statement

$S = \begin{cases} \text{if } x < y \text{ then } \text{swap}(x,y) \end{cases}$
which is equivalent to the concurrent assignment \((x,y) : = (f(x,y), g(x,y))\)
for definable functions \(f\) and \(g\). Choose the postassertion
\(x = a \land y = b\), where \(a\) and \(b\) are arbitrary numbers. Using the
swap rule and the second Conditional rule (with \(S_2\) empty) we prove
\[
\{ y = a \land x = b \land x < y \lor x = a \land y = b \land x \geq y \} \ S \ {x=a \land y = b}
\]
and conclude
\[
\forall x,y (y = a \land x = b \land x < y \lor x = a \land y = b \land x \geq y \Rightarrow f(x,y) = a
\land g(x,y) = b ) .
\]
This gives
\[
x < y \Rightarrow f(x,y) = y \land g(x,y) = x, \text{ and}
x \geq y \Rightarrow f(x,y) = x \land g(x,y) = y,
\]
which defines \(f\) and \(g\) for all values of \(x\) and \(y\). These
functions are usually called max and min, thus \(S\) is equivalent to
\[
(x,y) : = (\max (x,y), \min (x,y)) .
\]

Procedure call.

The general substitution rule above is valid for arbitrary postassertion \(R\).
\[
\{ R_{v_1,v_2,\ldots,v_n} \ f_1(w), f_2(w), \ldots f_n(w) \} \ S \ {R},
\]
which is useful if \(S\) is invoked at several places in the program.
This leads to the following rule for procedures.

Given proc \(p(v_1,v_2,\ldots,v_n)\); name \(v_1,\ldots,v_k\);
(specification of \(v_1,\ldots,v_n\) \(S\);
where \(S\) does not defer directly to nonlocal variables, and \(k \leq n\).
Then
\[
\{ P \} \ S \ {Q} \ gives \ \{ P_{f_1(A), f_2(A), \ldots f_k(A)} p(a_1,a_2,\ldots,a_k,\ldots,a_n) \} \{ R \},
\]
provided that \(a_1,\ldots,a_k\) are different variables, and where \(A\)
is the list \(a_1,\ldots,a_n\) and \(f_1,\ldots,f_k\) are as above. The rule is
easily extended to procedures with nonlocal variables. It is
valid for recursive procedures.

Blocks.
\[
\{ P \} \ S \ {Q} \ gives \ \{ P \} \begin{align*}
declare \ & v_1,v_2,\ldots,v_n; \\
S \ & end \ \{ Q \},
\end{align*}
\]
provided that \(P\) and \(Q\) contain no free occurrences of \(v_1,v_2,\ldots,v_n\).
Abstraction.

Aggregating operations and data, both at the same time, provide a mechanism of abstraction. Let $p$ be a procedure updating nonlocal variables $v_1, v_2, \ldots, v_n$ and whose parameters $x$ are called by value.

\[
\text{proc } p(x); \langle \text{specify } x \rangle; S;
\]

Then \{P\} $S$ \{Q\} gives \{R\}_{f_1(a, v_1, \ldots, v_n), \ldots, f_n(a, v_1, \ldots, v_n)} \{p(a)\} \{R\}

for arbitrary $R$, where $f_1, f_2, \ldots, f_n$ satisfy

\[V, v_1, \ldots, v_n(p \Rightarrow Q_{f_1(x, v_1, \ldots, v_n), \ldots, f_n(x, v_1, \ldots, v_n)}, g(x, v_1, \ldots, v_n)).\]

We collect the procedure $p$ and the variables $v_1, v_2, \ldots, v_n$ by a class declaration.

\[
\text{class C};
\]

\[
\text{begin } \langle \text{declare } v_1, v_2, \ldots, v_n \rangle;
\]

\[
\text{proc } p(x); \langle \text{specify } x \rangle; S;
\]

\[
\text{end of C};
\]

Given $C \text{ var } V$; which declares an instance named $V$ of the class body, we may take $V$ to be a variable of an abstract type $C$, represented by the variables $v_1, v_2, \ldots, v_n$, and whose abstract value is a function of the latter, the "abstraction function", \[3\].

\[V = F(v_1, v_2, \ldots, v_n)\]

The procedure $p$, local to $V$, is an abstract operator updating the abstract value of $V$. We use the notation $V \cdot p(a)$ for invoking the operator. Then the rule

\[(*) \quad \{R \}_{f(V, a)} \quad V \cdot p(a) \quad \{R\}\]

holds for arbitrary $R$, where

\[f(V, a) = F(f_1(v_1, \ldots, v_n, a), \ldots, f_n(v_1, \ldots, v_n, a))\]

and $f_1, \ldots, f_n$ are as above.

Often the abstraction function $F$ is meaningful only if a certain invariant relation $I$ holds for the arguments $v_1, \ldots, v_n$. The invariant $I$ may be established initially by statements $S'$ in the block tail of $C$, and I must be preserved by $p$.

Then the rule $(*)$ is established by proving

\[\{P \wedge I\} \quad S \quad \{Q \wedge I\} \quad \text{and} \quad \{P_0\} \quad S' \quad \{Q_0 \wedge I\}.\]

Furthermore \{P_0\} $C \text{ var } V \{V = V_0\}$ is true provided

\[Q_0 \wedge I \Rightarrow F(v_1, \ldots, v_n) = V_0.\]

It is assumed that the variables $v_1, \ldots, v_n$ are not updated textually outside $C$, except through invoking the local procedure $p$. 
Informal examples of abstraction are given in [4], pp. 205-208, and in the following section.

References.


Bottom-Up Construction, an Illustration.

Problem: Process sequence of telegrams for accounting purposes. (Cf. Henderson and Snowdon: An experiment in structured programming, BIT 12,1 (1972) pp. 38-53.) Each telegram should be printed out and in addition its number of words should be counted and printed, and a warning message should be given if any of its words is longer than K characters. Each telegram ends with the word ZZZZ. The words STOP and ZZZZ do not count. The sequence ends with a telegram containing no countable words.

The telegrams are stored on an input medium as a record sequence. Each record contains N characters. No word is divided across records, and blanks are used for filling up. The same rules apply to the output medium. Output records have length M.

Given: the type char (character value) with the operators =, ≠, and the following I/O-mechanisms.

\[ \text{proc read (A); char array A;} \]
which reads the next input record into the first N positions of A, where the length of A is N or more.

\[ \text{proc print (A); char array B;} \]
which outputs an output record from the first M locations of A, possibly padded with blanks if the length of A is less than M.
A string notation is available for `char array` constants. Also the equality operator is assumed to apply to character arrays.

```plaintext
class incharseq;

{An input character sequence is formed by "concatenating" the records of the input file, each extended by a blank character.}

begin cnar array buf [1:N+1]; int i;
    {i points to the current character of buf, which contains the current record.}
    cnar proc c; c:=buf[i]; {the current character}
    proc adv; {advance to the next character}
        if i <= N then i:=i+1
        else read(buf); i:=1 fi;
    {The initial character is a simulated blank considered the last character of a mythical record preceding the input sequence.}
    i:=N+1; buf[i]:=blank
end of incharseq;

class string(n); int n;

begin cnar array w[1:n]; int lg;
    {A string of length lg is contained in w[1:lg], where 0 <= lg <= n}
    proc clear; lg:=0;
    proc add(x); char x;
        if lg >= n then error ('string overflow')
        else lg:=lg+1; w[lg]:=x fi;
    clear {a string is empty initially}
end of string;

class inwordseq(n); {time sequence of words from input}

begin string(n) var word; incharseq var inc;
    {word contains the current word read from inc}
    proc adv; {advance to next word}
        begin word.clear;
            loop while inc.c=blank: inc.adv repeat;
            loop: {collect letters, including trailing blank}
                word.add(inc.c);
        end of loop;
end of inwordseq;
```
while inc.c ≠ blank: inc.adv; repeat
end of adv;
end of inwordseq;

class outwordseq; {sequence of words for output}
begin
  array buf[1:M+1]; int i;
  {buf[1:1] has been filled, buf[M+1] can only be filled with
    a redundant trailing blank}
  proc throw; {output buffer, unless empty}
    if i > 0 then
      for j := i+1 to M: buf[j] := blank repeat;
      print [buf]; i := 0
    fi;
  proc out(s); string val s;
    begin
      if i+s.lg > M then throw fi;
      for j := 1 to s.lg:
        i := i+1; buf[i] := s.word.w[j]
      repeat
    end;
i := 0 {empty buffer initially}
end of outwordseq;

Main program:
begin
  inwordseq(50) var Wi; outwordseq var Wo;
  int wcount; Bool longw;
  {wcount
    loop: {zero or more telegrams have been processed}
      wcount := 0; longw := false;
      {start processing another}
    loop: {zero or more words have been read of the current
      telegram. wcount of them were countable. longw
      means one or more were too long
      Wi.adv;
      while Wi.word ≠ 'ZZZZ': Wo.out(Wi.word);
      if Wi.word ≠ 'STOP' then wcount := wcount + 1 fi;
      if Wi.word.lg > K then longw := true fi;
    repeat; {another telegram has been processed}
  while wcount ≠ 0:
    Wo.throw; printrnum(wcount);
    if longw then print ('warning message') fi;
  repeat
end of main program
Reading List.

Books.
O.-J. Dahl, E.W.Dijkstra, C.A.R.Hoare:

G.Birtwistle, O.-J. Dahl, B.Myhrhaug, K.Nygaard:

O.-J. Dahl, D.Belsnes:

P.Brink-Hansen:

Short Selection of Articles:
E.W. Dijkstra:

E.W. Dijkstra:
Goto statement considered harmful. CACM 8,9,pp147-147 (Sept. 1968).

R.W. Floyd:

P.Henderson and R.Snowdon:


C.A.R. Hoare:

C.A.R. Hoare:
C.A.R. Hoare and N. Wirth:

D.E. Knuth:

Miller:
The magical number 7 plus or minus two: Some limits to our capacity for information processing. Psychol. Rev. 63, pp.81-87.

P. Naur:


A. Wang and O.-J. Dahl:
1974 CERN SCHOOL OF COMPUTING, GODØYSUND, NORWAY

ORGANIZING COMMITTEE

BOLLIEF, L. Université Scientifique et Médicale de Grenoble, France
DAHL, G.-J. Universitetet i Oslo, Norway
LILLESTØL, E. Collège de France, Paris, France and
            Universitetet i Bergen, Norway
LILLETHUN, E. Universitetet i Bergen, Norway
LOCK, W.O. CERN, Geneva, Switzerland
MACLEOD, G.R. CERN, Geneva, Switzerland
ZANELLA, P. CERN, Geneva, Switzerland

Conference Secretariat:
BARNETT, I. CERN, Geneva, Switzerland
CATON, D.A. CERN, Geneva, Switzerland

LIST OF PARTICIPANTS

SPEAKERS

BÖCK, R. CERN, Geneva, Switzerland
BUXTON, J.N. The University of Warwick, Coventry, U.K.
DAHL, O.J. Universitetet i Oslo, Norway
FRIEDMAN, J.H. Stanford Linear Accelerator Center, U.S.A.
GUIBOUD-RIBAUD, S. Centre Scientifique CII, Grenoble, France
HEWITT, C. MIT, Cambridge, U.S.A.
JANK, W. CERN, Geneva, Switzerland
MAZARE, G. Centre Scientifique CII, Grenoble, France
MONRAD-KROHN, L. Norsk Data Industri, Oslo, Norway
SUMNER, F. Department of Computer Science, Univ. of Manchester, U.K.
VERKERK, C. CERN, Geneva, Switzerland
AASEN, I.J. Agder distriksøgskole, Kristiansand, Norway
ALLIN, D. University of Birmingham, U.K.
ASTESAN, F. LPNHE, Université Paris VI, France

BALDO, B. Istituto di Fisica, University of Rome, Italy
BERTHON, U. LPNHE, Ecole Polytechnique, Paris, France
BILHAUT, R. LAL, Groupe Chambre à Bulles, Orsay, France
BISACCHI, G. Centro di Calcolo dell'Università, Turin, Italy
BJØRNENAK, K. EDB-avd., Bergen, Norway
BLENCKE, L. Zentrum f. Rechentechnik d. Akademie d. Wissenschaften
St. DDR, Berlin, German Democratic Republic
BLOODWORTH, I.J. Rutherford Laboratory, Chilton, U.K.
BOOTH, I. Mancaster Group, Daresbury Laboratory, Warrington, U.K.
BRANDT, A. CERN, Geneva, Switzerland
BRAUEL, P. Universität Hamburg, II. Inst. f. Experimentalphysik,
Hamburg, Federal Republic of Germany
BRIDET, G. C.C.P.N., Paris, France
BRUN, R. CERN, Geneva, Switzerland

DE BEER, M. Centre Nucléaire de Saclay, Gif-Sur-Yvette, France
DENES, E. Central Research Institute for Physics, Budapest, Hungary
DUMONT, J.-J. Inter-Univ. Inst. for High Energies, Brussels, Belgium
ENDSOR, R. Culham Research Lab. Abingdon, U.K.
ENGEL, K.-D. Rechensentrum, Berlin, Federal Republic of Germany

FJELLHEIM, R.A. CERN, Geneva, Switzerland
FRANZ, J. Fakultät f. Physik, Freiburg, Federal Republic of Germany

GLASNECK, C.-P. Inst. f. Hochenergiephysik der AdW der DDR, Zeuthen,
German Democratic Republic
GREEN, G.A. RUNIT, Trondheim, Norway
GRUNG, B. CERN, Geneva, Switzerland

HALLSTEINSEN, S.O. University of Trondheim, Norway
HOCHWELLER, G. DESY, Hamburg, Federal Republic of Germany

ISENBECK, W. Inst. f. Experimentelle Kernphysik, Karlsruhe,
Federal Republic of Germany
JARP, S. CERN, Geneva, Switzerland
JHA, S.K. Delhi Univ. Computer Cent., Delhi, India
JOHANSSON, H. Inst. of Physics, Univ. of Stockholm, Sweden

KHAKHAR, D.P. Dept. of Computer Sciences, Lund University, Sweden
KIRKMAN, H.C. Daresbury Laboratory, Warrington, U.K.
KUMMER, P.S. Daresbury Lab., Warrington, U.K.

LADRON DE GUEVARA, P.J. Collège de France, Lab. de Physique Corpusculaire,
Paris, France
LANG, J. Cavendish Lab., Cambridge, U.K.
LECOQ, J.  Lab. des Applications Electroniques, Strasbourg, France
LUNDBORG, P.  Institute of Physics, University of Stockholm, Sweden

MATT, W.  MPI für Physik, Munich, Federal Republic of Germany
McPHERSON, G.M.  Rutherford Laboratory, Chilton, U.K.

PERRIN, Y.  CERN, Geneva, Switzerland
PFEIFFER, G.  DESY, Hamburg, Federal Republic of Germany
PHARABOD, J.P.  LPNHE, Ecole Polytechnique, Paris, France

QUARRIE, D.R.  Rutherford Lab., Chilton, U.K.

RANJARO, F.  CERN, Geneva, Switzerland
REITBERGER, M.  MPI für Physik, Munich, Federal Republic of Germany
RICCI, E.  University of Bari, Italy
RUMPF, M.  LPNHE, Ecole Polytechnique, Paris, France

SCHAILE, O.  Fakultät für Physik, Freiburg, Federal Republic of Germany
SCHIKORA, R.  Inst. f. Medizinische Datenverarbeitung der GSF,
              Munich, Federal Republic of Germany
SCHIMETZK, H.  Inst. f. Medizinische Datenverarbeitung der GSF,
              Munich, Federal Republic of Germany
SCHMITZ, P.  III. Physikalisches Inst. der RWTH Aachen,
              Federal Republic of Germany
SCHUBEL, H.  DESY, Hamburg, Federal Republic of Germany
SIAHAAN, K.  Abt. für Informatik II der Universität Bonn,
              Federal Republic of Germany
STEYAERT, J.R.L.  Univ. Cath. de Louvain, Louvain-La-Neuve, Belgium
STRICKER, B.  Centro di Calcolo dell' Università, Turin, Italy

TENGSTRAND, T.R.  Research Inst. of National Defence, Sunbyberg, Sweden
THORSTEINSEN, P.  Central Inst. for Industrial Research, Oslo, Norway
TIMMERMANS, J.M.  Fysisch Laboratorium, Univ. of Nijmegen, Netherlands
TOTH, I.  University of Bern, Switzerland

WAKEFORD, R.  Oliver Lodge Lab., University of Liverpool, U.K.
WASSENAAR, E.  Zeeman Laboratory, Amsterdam, Netherlands