

NOTATIONS AND PHYSICAL COMPONENTS USED IN THE DESCRIPTION, DESIGN,
AND TEACHING OF COMPUTING STRUCTURES

C. Gordon Bell

Rapporteurs: Mr. J. G. Givens

Mr. J. F. Dunn

Abstract: Two notations, PMS and ISP, have been developed in order to describe computer structures. PMS (for Processors, Memories, and Switches) is a notation for showing the structure of physical information processing components in terms of information flow attributes. ISP (for Instruction - set Processor) is a notation for defining the behaviour of the components in terms of register transfers. A set of Register Transfer Modules (RTM's) has been developed which uses the notations. In three lectures, Professor Bell described the modules and notations and discussed their relationship to the description, analysis, and design of digital systems. Finally, he discussed briefly the integration of hardware into a computer science curriculum, such as has taken place at Carnegie-Mellon University.

Lecture 1

In response to Professor Randell's introduction, Professor Bell began with a few words about the book he and Allen Newell wrote (2) and said he was going firstly to discuss physical tools for teaching about machines. His lectures would move from physical detail towards notation and philosophy. Unlike Professors Barton or Lampson, Professor Bell was more concerned with teaching about the past and present, since he could not foresee the future of technology, and since the future is an evolution. His lectures discuss three topics:

- (i) Register Transfer Modules: A physical set of Modules for designing digital systems;
- (ii) Notations for describing computers, as discussed in Bell and Newell's book and used therein to describe about forty machines:

PMS (Processors, Memories, and Switches) describes the physical interconnection of a digital system, while ISP (Instruction-set Processor) is used to describe the behaviour of the instruction set of a computer, i.e. to describe formally what the machine does (ISP could replace conventional programming reference manuals).

- (iii) The Integration of Hardware into a computer science curriculum.

Register Transfer Modules (RTM) Trademark of Digital Equipment Corporation -- RTMs are marketed under the name PDP-16.

Register Transfer Modules occupy a medium level of logical design capability, having registers and register operations; they are below the level of components such as processors, secondary memories, disk controllers, and so on, but not at such a low level as 'and' and 'or' gates, flip flops, etc. In 1967 Carnegie-Mellon University applied for an equipment grant to use in teaching, intending to use Clark's Macromodules were a few years late in arriving and five to ten times more expensive than anticipated, they were impractical for teaching. Therefore, I set out to design a set of Modules for teaching. However, they are of academic interest and commercial use as well. (They were not designed solely for teaching purposes because of the limited market.) They are described in detail in an article (1). There were three influences behind the design: the greatest was that technology had only certain things with which to build, but also I had my own prejudice about design, and most of the principles were carried over from earlier machines on which I worked, like the PDP-6 and PDP-10. The other influence was my conviction about the teaching of logical design. While teaching about combinational and sequential circuits - switching circuits - has become a really reputable academic pastime, I believe that switching circuits are almost totally irrelevant to computer design and I want the modules to reflect this. Some earlier texts do use register transfer concepts, in fact, but in a very limited way. The basic requirements, then were:-

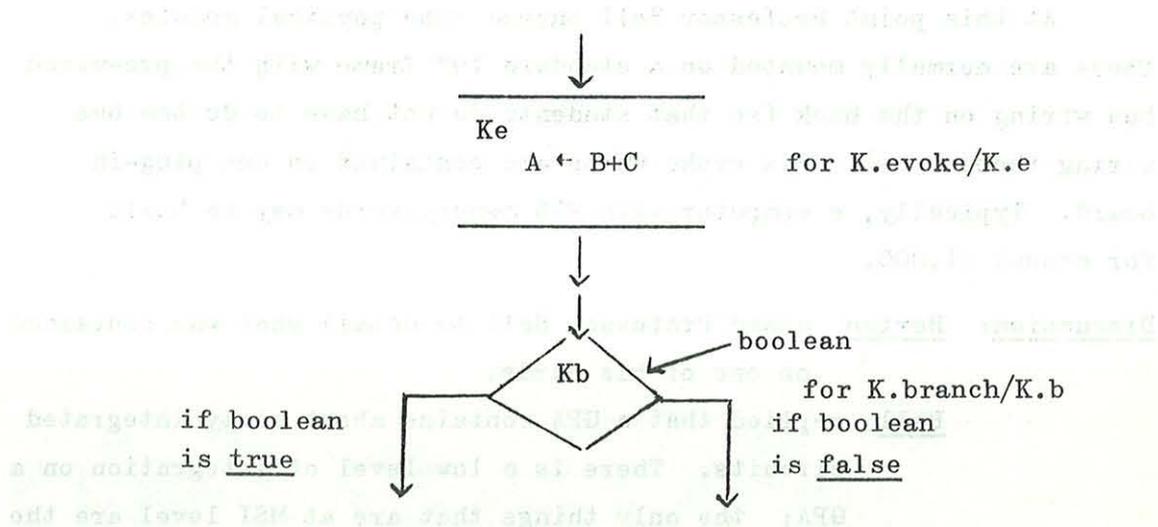
- (i) The absence of switching circuits in design.
- (ii) A representation which was similar to the physical structure. Whilst a conventional state diagram is fine for an initial design, as altered all of the logic is affected to a high degree. Hence I do not regard it as a useful design tool.

- (iii) Effective use of technology - nobody seemed to be making effective use of the new components, except to increase density.

The applications of register transfer modules are: special purpose digital systems; the computer-related area; and, least constraining, for teaching. The size of the problem covered is one of perhaps ten to one hundred control states and ten to one thousand words of read/write or read only memory, of speed say 200 nanoseconds to 1 microsecond per register transfer operation. Logic interconnection rules and TTL logic technology are used; the word length to encode an integer may be 8, 12 or 16 bits. So we arrive at a fairly small system. It is not worth competing with mini-computers, but in fact using this technology a mini-computer can be packaged easily. In fact, an RTM implementation costs less than a poorly packaged (or aged) mini-computer.

Figure 0 includes some key modules. There are four module types: K-control for controlling all interaction between other Modules; M-memory for holding data; DM-data operations with memory (e.g., an accumulator); T-transducers for getting data in and out of the system (e.g. analogue devices, Teletypes).

A system has two parts: the data part (consisting of M, DM, and T modules), (and the control part (K Modules). The control part of a system, costing about \$5 per control step, causes an operation such as $C \leftarrow A+B$, and C are registers, to take place in the data part using an evoke module, K. evoke. There is also feedback from the data part to the control part. A Kb control (K. branch) causes a particular operation to take place depending on the value of a Boolean input to it. It is quite useful that we can write out these 'K's in flow chart notation, such as:



where these also are physical components, and interconnecting lines represent wires.

We are now in a position to build an actual computer (fig. 0). All the data-type Modules are interconnected by means of a bus in the upper right of the figure, and there is a hidden bus controller K. bus. Consider the operation $P \leftarrow A$. A signal is sent from an evoke, K_e , to the first GPA, causing it to send the value of register A (the accumulator) on to the bus; the second GPA is signalled to receive a value from the bus and place it in register P (the program counter). The hidden bus controller, K. bus, signals when the transfer over the data bus is complete.

All transfers of information take place via the data bus. A register i holds an instruction, as obtained from memory. Consider a 16-bit word size, when an instruction has a 3-bit opcode ($i\langle 15:13 \rangle$) and an 11-bit address ($i\langle 10:0 \rangle$). This leaves two spare bits ("for expansion"). Some typical operations on memory M are shown also in fig. 0, where registers MA and MB are the memory address register and the memory buffer register respectively, and L is the link register for the jump and link (JML) instruction.

Various bits of i can be used for special purposes, as with the 'Operate' operation (opcode 7). This could be achieved with a manual evoke; a button is pressed to give a start signal. The entire execute finishes with a serial merge, which takes control flow back to the fetch step again.

At this point Professor Bell showed some physical modules. These are normally mounted on a standard 19" frame with the pre-wired bus wiring on the back (so that students do not have to do the bus wiring themselves). Six evoke units are contained on one plug-in board. Typically, a computer with 256 memory words may be built for around \$1,000.

Discussion: Barton asked Professor Bell to detail what was contained on one of his cards.

Bell replied that a GPA contains about sixty integrated circuits. There is a low level of integration on a GPA; the only things that are at MSI level are the four standard ALU chips (four bits at a time) of thirty two functions, most of which he does not use. There are also the A and B registers, and the data paths.

Barton then asked what the other ICs were on the card shown.

Bell replied that there are the bus interfaces: ICs to drive the bus and to receive from it. The card is filled with input gates so that many evokes can be done to a single Module. There are about five transfers into the A and B registers.

Horning asked how many cards were needed and how long it would take to construct in the laboratory the computer which had been discussed.

Bell said that for the control part about eight big cards ($8\frac{1}{2}$ " x $5\frac{1}{4}$ ") and 12 small ones with K. evokes and K. branches. All fit in one 19" x $5\frac{1}{4}$ " mounting panel. The time required to wire a computer is about four hours (i.e. about one wire per minute). A good student can get a computer operational in eight hours.

Lauer interrupted to ask if that included memory.

Bell said it does; the memory is bipolar.

Page asked what initial knowledge a typical student possessed when starting a course: what were Professor Bell's prerequisites?

Bell replied that the electrical engineering students had only programming knowledge. This work occurs in the second term. It is a lot more natural than working with 'and' gates. How, he remarked, does one add a couple of integers with an 'and' gate?

Suchard asked what an 'and' gate meant to a student.

Bell said it was a physical thing, and 'and' gates do not really interest him at all.

Barton wondered if students were told about the details of the components in the modules.

Bell explained that they learn afterwards, in the second semester. At this point, these are components just like 'and' gates.

Michaelson asked if the students were actually given instruction codes.

Bell replied that they were not; this is part of the design problem.

Typically, in the first semester, students may design a device, for example, to monitor the thickness of a coating on a continuous assembly line and display results of calculations, or the distribution, on the memory lights - a typical hardwired controller. The algorithm can be hardwired or an interpreter may be built.

Lecture 2

The modules described in the previous lecture are used in logical design within electrical engineering. In the laboratory class, six active lab. stations and twenty module panels are used. This level of design has been taught to students, both of computer science and of electrical engineering, for a couple of years now. We teach this "top down" approach, and the computer science students, at least, are happier starting with register transfer modules and going on to logic problems later.

To illustrate some of the things we do: a simulator was written last summer before we had the modules, but it used quite a bit of computer time. Using the modules, we can illustrate parallelism, as a parallel branch is just a wire and the consequent parallel merge is another control.

We can also illustrate synchronism. We show the sharing of common control logic; a "producer" module can be linked to a "consumer" module with a "queue" module, and P and V functions can be demonstrated (though the "busy waiting" state is satisfactory in our systems of modules since they are not multiprogramming). Half duplex and full duplex transmission can be illustrated. Here there are problems of design interaction. We pose problems where two groups of three people co-operate to design a system to transmit data to one another: a group works on either end, and a number is to be sent in both directions. Errors can be introduced into the transmission later.

NOTATIONS, as used in Bell and Newell's book.

PMS is used to describe the physical structure of the interconnection of computer components, and the ISP notation is used for describing instruction sets. ISP can be used for other purposes, but is not designed for expressing algorithms generally. These notations were devised in order to provide methods for describing the machines we have (not those we don't have yet) as clearly as possible, although there is every indication that it does the latter too.

PMS for Processors, Memories and Switches.

The primitives of PMS, as well as Processors, Memories, and Switches, are K/controls, Computers, Transducers, Links, Human (machine operators, etc.), and Data operations. The underlying motivation for PMS was to provide a fairly casual but formal method of describing parts of machines and illustrating their structure, and of comparing the structures of different machines. It is a representation by which one may posit the state of a design and modify it. Using it, one may check if configurations are legal and so on. Possibly, it could be used in setting definitional standards.

Fig. 1 summarises the ideas of PMS. PMS is defined in Bell and Newell's book, it deals only with information flows (bits per second). Now, we can also use it at lower levels (than processors, etc.); for example, we have D/data operation $\rightarrow \boxed{D(AND)} \rightarrow$; we can nicely classify current technology in this way. Is the notation general enough? $\rightarrow \boxed{AND} \rightarrow$ is almost legal PMS! At the other extreme, using PMS, one can check for what are sometimes called "overruns" (e.g. on a disk,

because of insufficient channel capacity), and check if one has a legal configuration.

Fig. 2 shows a simplified computer block diagram of Whirlwind I. It is a 1949 diagram, but is fairly typical of modern diagrams. Possibly it has more detail than a modern diagram. This is a classical block diagram; we wanted more detail than that.

Fig. 3 shows fig.2 translated into PMS - it is almost the same diagram. Note the all-powerful control and the primary memory.

Fig.4 shows how we can describe Whirlwind fairly accurately in the same space as the block diagram used. Engineers have complained that there are no boxes round the components. I tell them to surround the primitives with boxes if they want to.

In Whirlwind there were two parts of primary (program-storing) memory. The superscripts relate to footnotes and are used to clarify the diagram. Pc is a central processor (the commonly used term CPU is ambiguous - is there memory associated with it or not?). There is some switching, and there are some controls for transducers for I/O. Note the 5" and 10" cathode ray tubes, one for producing film and one for visual display. There are two drums and a magnetic tape drive.

Randell asked what the arrows meant.

Bell They show direction of information flow. The vertical arrows show the connection between the cathode ray tubes and the light pen and camera. Most horizontal arrows show information flowing out of the system.

Michaelson explained that the arrows were not typed very accurately.

Fig.5 shows the official definition of a computer. Mp is the primary memory.

Fig.6 shows a conventional functional block diagram and the corresponding PMS notational model. These are shown for comparison so that it may be seen what the mapping is. We usually don't see so much in a machine structure diagram as we do in fig.7 because blocks tend to have only their most important function associated with them. Note,

in this figure, that

- (i) memories tend to have control associated with them;
- (ii) using the notation, we can break components into more and more details in terms of lower level components;
- (iii) how can we say a processor is a primitive? We can, if we are interested only in the level of processors.

We can go down from the basic $C := Mp - p$ to as low a level as we want: to 'and' gates - even to circuits! If we have a computer network

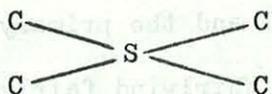
 (S is a switching medium or a fixed link device) we may not want more detail. We need only represent as much as is necessary for the task of description at hand.

Fig. 8 shows how we can write a decomposition of a component into sub-components, which are all well defined. The four switches, from left to right, select the disk drive, the platter, the track on the disk, and the word wanted on that track respectively. The substitution sign means that this is how we define the disk.

Randell asked how the difference between a normal disk and one getting information from 32 parallel tracks would be shown.

Bell The timing would show. The information flow into the device, as a bit rate, could be evaluated, and one would see 32 times the information rate if the disk was operating in parallel while otherwise it would be the same. The attributes of the switch give the transmission width.

Fig. 9 shows an illustration of the conventions we use for abbreviating parameters to save writing: we have aliases for names. In the case of memory, we must distinguish primary memory from secondary memory and so forth

Fig. 10 shows the essential structure of the PDP-8 multiprocessor system. On to the basic computer structure are added a display processor and another processor called a LINC processor; there is also a secondary memory, a console, and a transducer. The LINC processor has its own

secondary memory. The two independent lines are for data flow and control flow, but we need not show this much detail in our initial diagram. Fig. 11 shows on a page all the characteristics of the PDP-8 LINC computer.

Now, to illustrate a large computer, fig.12 shows part of the "IBM 6600". There is a large central processor, etc., there are also ten I/O computers which switch to a fairly large periphery. Fig.13 shows the same but provides more detail, e.g. how fast the machine is. We see, in the top part, ten smaller memories (peripheral processor memories) connected to ten processors, and the barrel, the processor state associated with a processor. The barrel has ten words each of 51 bits, with access time 100 nanoseconds per word. There are various types of switches to the periphery, and two controls called the read pyramid and the write pyramid for transferring state to the central computer.

In the CDC 6500, there are two processors. Fig.13 ignored all activity associated with the central processors. This is given in the footnote (fig.14) which defines the central processors and gives some important attribute values. We see some important characteristics of that machine and get an idea of the amount of data flow and hence what it can process.

Fig.15 shows the basic N('CDC 7600). N denotes that the machine is really a network of computers; the "'" denotes a manufacturer's name or the proper name of a component. This machine is a derivative of the 6600 and 6500; it has fifteen peripheral processing computers connected to fast channels and also one single processor connected into primary memory.

Fig.16 shows the footnote to fig.15; we see what we expect to get from the central processor. It's a fairly fast one; $27\frac{1}{2}$ nanoseconds per word, a 16 word processor state, and a 60-bit word. In note 4, the peripheral processing units are defined: a pair of small but fast memories connected to a single address per instruction processor.

Finally, fig.17 shows how machines are mapped into a general model.

Discussion Page

said that Professor Bell had a notation with which he could describe quite compactly machines of the present and the past. How much history did Professor Bell feel it was worth while to present in a course?

Bell asked to defer answering.

Randell complimented Professor Bell on his notation, whose main importance is the concepts of the classification which lie behind it. Since the notation is two-dimensional, the engineers' request for boxes struck Professor Randell as an eminently reasonable one.

Bell commented that line printers do not have boxes in their character sets, and said that boxes never struck him as being important. In fact, he remarked, he would like three dimensions, but it is hard enough to get books published without the need for stereoscopic glasses in each book. He went on:

PMS is more than just a way of describing older machines. We're really designing with it now. A hard part of design is trying to generate enough alternatives. Engineers tend to like their first design - not the one they're happiest with, but the only one they look at. With some way of describing what you've got you can generate more alternatives. I am tired of people re-inventing all the old concepts. The purpose of Bell and Newell's book was to get the point of view across where when we start talking about a new computer we are starting at a reasonably high level. A surprising number of 'new' machine ideas related to microprogramming were in either Ace or Atlas; I'd like people to read about these machines. Still, the constant re-invention of ideas works for some reason; maybe there are some new ideas?

Randell "He who forgets the past is forced to relive it!"

Lecture 3

Whether or not our notations will be of any use depends on what 'tricks' we can do with them. In the case of PMS, we can classify the set of components, which helps in design work. Designers operate better if they've got a well defined set of components that they understand. When marketing, also, it helps if units can be well defined and labelled with a name all too often just to catch people's imagination and attention. For example, names like multiplexor, channel, and so on catch on and are better than numbers.

With regard to the use of PMS inside an actual computer, we are

doing three things: we are putting the data structure of PMS into a machine, we are operating on this data structure and asking questions about it, and we are performing calculations on the structure; this is the real test of the notation. Fig.18 shows how we can print on a typewriter or oscilloscope the relationship of the components. It shows the kind of relationship that happens to be there. Now, given that we have a computer structure in our machine, we'd like to know something of its reliability, and fig.19 shows how we are able to specify various paths through the structure and obtain the overall reliability. This is a good way to look at reliability. Currently, for a multiprocessor/multimemory structure with a certain amount of I/O, it is possible to compute the cost and, in closed form, the performance of the system. Fig. 20 shows how we can plot quality, cost, and performance per unit cost as a function of the number of memories. We're trying to develop this for interactive use.

ISP for Instruction-set Processor. (Fig.21).

ISP is for describing the instruction sets of current machines, and also machines of the next few years at least. Its purpose is to define a computer (i.e. an instruction set) as seen by a program or programmer. Its primitives are memory, instruction formats, and so on. Its use is in the description, comparison, and formal specification of instruction sets, and it may be interpreted by machine.

It is desirable to be able to define a machine and its instructions precisely. In Bell and Newell's book, we described ten to fifteen machines using ISP. Since then, the ISP specification of a fairly large mini-computer has been produced and this specification has been placed in the programming manual. It is not in lieu of the programming manual, but as an experiment to investigate its usefulness. ISP tends to be fairly good for doing design work.

Incidentally, we didn't have the ISP of System/360 when we wrote the book. We thought it would be too difficult to work out. Since then, I've had the opportunity to start a project to work out the ISP of the 360, which is still not yet completed. It is a very difficult task to get the ISP of a machine of that scale. I have a great deal more respect for the 360 after trying to do that task. It is very difficult when one has access only to the programming manual.

Barton asked what Professor Bell thought of the APL description of the 360.

Bell It's a very impressive job of describing the 360. We've referred to the APL description from time to time when we've come across something not in the manual, but it is very difficult, even for someone fully conversant with APL, to extract instruction descriptions from the APL definition, and we want our notation to be used by people to understand what a machine is. As a 360 user, I couldn't make use of the APL description, but I want our language to be of use. More functions or procedures in the description would have helped.

Randell suggested that it was the structure of the APL that was the difficulty.

Bell I agree. It's too homogeneous. For example, we tend to define say floating point arithmetic in four or five steps. We might say

$$FAD \Rightarrow (F \leftarrow F + \{sf\} M [ea])$$

where the 'sf' defines a data step signifying the use of a particular kind of single precision floating point number, and later on we have to define 'sf', etc. In this way, one can see simply all one normally needs to know, but one can obtain greater detail if one wants to.

Now, let's take a small 12-bit machine and go through the definition of it (Fig.22). Note the use of italics to provide comments. The first stage in these ISP definitions, which are fairly highly stylised, is the definition of the central processor state, the memory which must be saved between starts and stops at the console switches. We could go at a lower level than we do, by clock pulse, but the definition is assumed to be for a program or a programmer who doesn't see the clock pulse. We have a 12-bit accumulator. We use a range marker to denote the range of digits within a register or memory cell. The notation $AC\langle 0:11 \rangle$ implies binary notation; for decimal we would have $AC\langle \dots \rangle_{10}$. For primary memory we have the array declaration $M\langle 0:7777_e \rangle\langle 0:11 \rangle$. These declarations are like memory declarations in Fortran or Algol. Then we define the subarrays: memory is divided into pages, and so on, and finally we have the state bits and the console and data switches.

Fig. 23 shows the instruction format, defined in the same way. We name the bits in the instruction format as we think people will want them named. Compare this to the usual familiar box diagram (fig.24).

Unfortunately, I think these box diagrams are better for showing instruction formats, and often use these boxes as comments! It's hard to beat a two dimensional representation in many cases.

However, when defining what the instructions do there is not much advantage in going to two dimensions. Fig.26 shows the instruction interpretation process as conditional statements: if a then b is written as $a \Rightarrow b$ (actually as $a \rightarrow b$ in these diagrams). The only reserved word in the language is "next" - execute the next statement sequentially. So the first two statements are executed in parallel, and then the next after these in sequence. The instruction is picked up from the memory location pointed to by the program counter, the program counter is incremented, and the instruction is executed: "fetch execute". Once the instruction is fetched, it is executed, and fig.27 shows what the instructions do. For $AND(:= op = 0) \Rightarrow (AC \leftarrow AC \wedge M[z])$ we may just write $AND \Rightarrow (AC \leftarrow AC \wedge M[z])$ - a shorthand, when we don't care what the opcode is. Being in a particular state when given a particular opcode, the machine executes a particular instruction. Note that all the operation statements are in parallel; hopefully only one operation will be executed.

Suchard asked what the symbol meant.

Bell denotes concatenation. We allow concatenation both on the left hand side and on the right hand side of a statement. The L in the diagram is a 1-bit register used in the two's complement add instruction.

Fig. 28 defines the microprogrammed operate instruction set, in which the remaining bits of the instruction are used to define the instruction.

In fact, figures 22, 23, and 25 to 28 provide the complete definition of this machine in two text pages, and people find this description is as precise as the programming manual. Whether I would want it to replace the programming manual I don't know. While I was drawing up the ISP, however, I found errors in the programming manual!

Discussion Suchard asked about the difference about left- and right- pointing arrows.

Bell replied that the arrows " \rightarrow " in the diagrams should really have been " \Rightarrow ", meaning if ... then It was a mistake not to separate them more, typographically.

Seitz asked if the ISP description would help one to design that machine.

Bell said that it provides a convenient description for helping a designer to express his thoughts on paper. ISP has been used in design situations, and some designers constantly use it. It is very good for expressing microprogram activity.

Michaelson remarked that this sort of thing struck him as being the unintelligent part of the design, and asked if it helped to teach students real design.

Bell pointed out that ISP helps to show the state of a design at one time, and transformations could be made in it - just like state diagrams, Karnaugh maps, and other abstract representations. He concluded that apprenticeship was, unhappily, the only way he knew to learn design, though the Open University's television methods were potentially great.

Michaelson commented that this was at variance with Professor Barton's comments, and asked Professor Barton for his opinion.

Barton replied that his feeling about apprenticeship was that it must not be used to force students to copy the methods used by their instructors, and Professor Bell agreed with this.

COMPUTING AT CARNEGIE-MELLON UNIVERSITY

I shall now discuss where all this fits into the educational programme. There are about forty full-time Ph.D candidates in the Computer Science department. We have about five students each year in the Ph.D and Master's programme in Electrical Engineering and about thirty students each year in the undergraduate Computer Engineering programme. The Register Transfer Modules are used by electrical engineering undergraduates. In the electrical engineering department, we use the programme outlined in a report (4) by a COSINE committee, which discusses computer engineering with electrical engineering.

In the Computer Science department we are not trying to provide computer designers - it looks as though Manchester can produce all the world needs ! and do a good job of it too. In another sense, there

is no choice, because ours is a computer science department - all our students are interested in operating systems and programming languages. That is the environment. Our whole programme consists of four "core" courses. I teach a semester "core" course, and try to produce graduates able to operate in a computer science environment. Assuming that students are going to be programming primarily, then I'm trying to teach them not to be frightened of the hardware so that they can challenge the engineer. I am continually annoyed with my fellow engineers, who are so pessimistic when discussing the possibility of some slight change to a machine. We are trying to teach our people to sort out when things are expensive and when they're not. We're trying to teach them to know how to read the rules of the game and maybe to play it. I think it's important to provide this defence mechanism, and also to teach another type of machine - a physical machine as opposed to mathematical, and language defined machines. This gives insight into the kinds of machine that one is working with. We are trying to eliminate the hardware/software barrier.

Discussion Page said that this was indeed a very powerful argument and asked if one should not neglect, even at greater cost, also putting into the course a study of the simulation of machines.

Bell replied that he did not want to spend too much time on that aspect. A student on this course would by this time have simulated a simple machine. Professor Bell commented that while it is fairly simple to write a simulator, it is on the whole very badly done and can waste a great deal of computer time. He supposed that students coming into this background have not been exposed to much of an engineering training, and so he liked to teach them about time, information, space, and cost. Problems can then be posed concerning minimising time, cost, and so on in circuits. Students should be encouraged to know, fundamentally, what components exist at various levels (though this is fairly technology dependent), and how to manipulate, analyse, synthesise, and

judge them. Students should be able to form various structures appropriate to the level of these components, and in some cases they are asked to write programs to generate structures; this is very interesting and very educational as well. These are the sort of things which are done in Professor Bell's department. When his students come out, they should have the attitude that they can design, no matter how theoretical their course has been. In some areas, notably mechanical devices such as disk files and tape drives, we badly need theoretical help!

References

1. C. GORDON BELL and JOHN GRASON: "The Register Transfer Module Design Concept" (Computer Design, May 1971, pp 87-94).
2. C. GORDON BELL and ALLEN NEWELL: "Computer Structures: Readings and Examples" (McGraw-Hill, 1971).
3. W.A. CLARK et al: "Macromodular Computer Systems" (Proceedings of the 1967 SJCC, pp 337-401 (6 papers)).
4. CLARENCE L. COATES, JR. et al: "An Undergraduate Computer Engineering Option for Electrical Engineering" (Proceedings of the IEEE, Vol.59, 6(June 1971), pp 854-860).

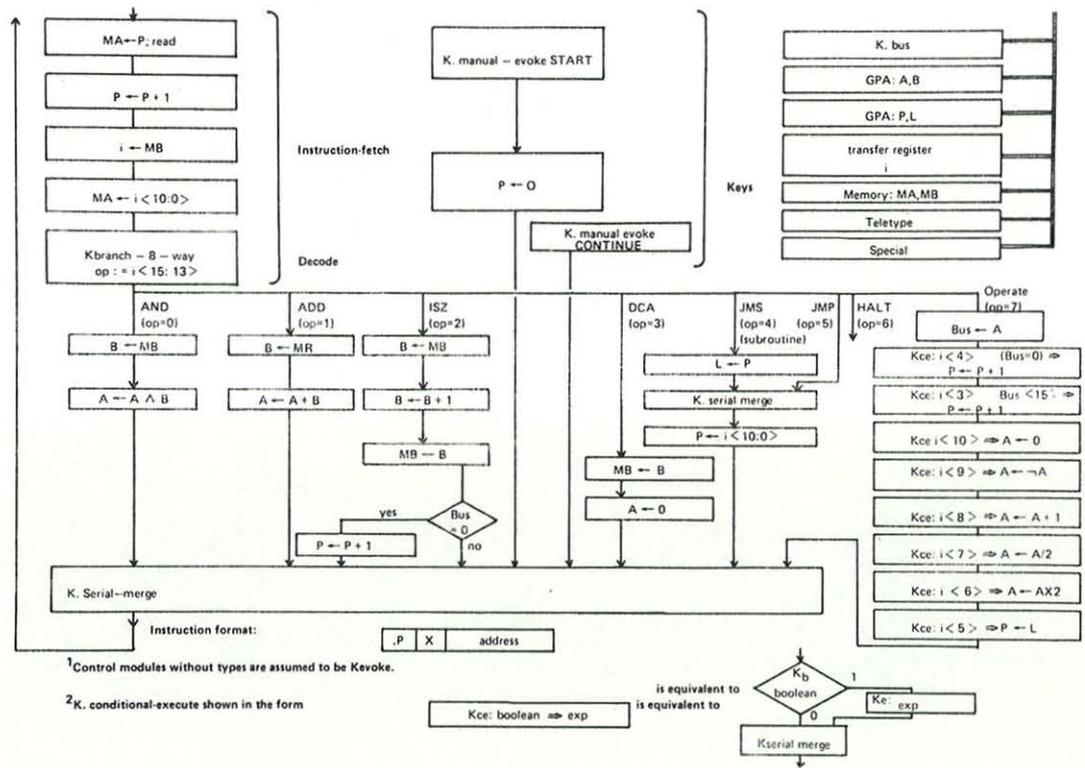


Figure 0

PMS - for Processors, Memories and Switches

- purpose: define the structure (inter-relationship) of computers from various viewpoints (e.g., information flow, power, space)
- primitives: P(processor), M(memory), S(switch), K(control), D(data operation), L(link), T(transducer, terminal), and H(human)
- uses: description, comparison, analysis (e.g., bottlenecks), specification, design (e.g., reconfiguration), standards

Figure 1

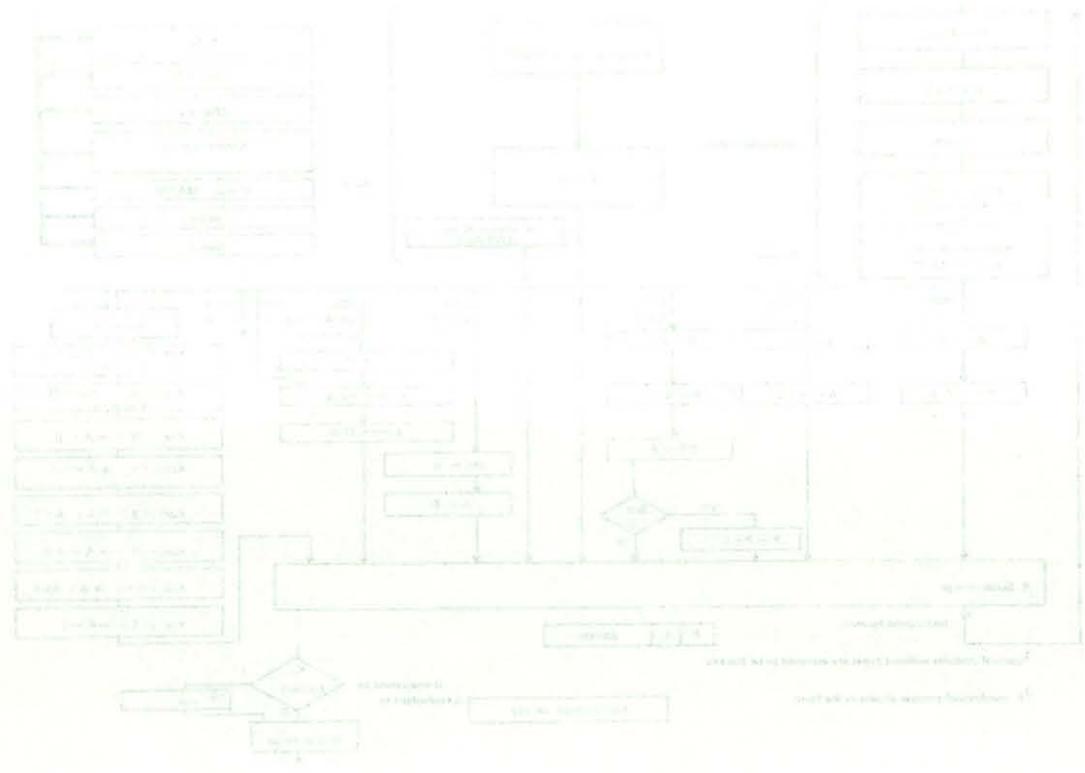


Figure 1

- uses: description, comparison, analysis (e.g., bottlenecks), specification, design (e.g., reconfiguration), standards
- H(human)
- L(link), T(transducer, terminal), and S(switch), K(control), D(data operation), P(processor), M(memory),
- viewpoints (e.g., information flow, power, relationship) of computers from various
- purpose: define the structure (inter-
- PMS - for Processors, Memories and Switches

Figure 2

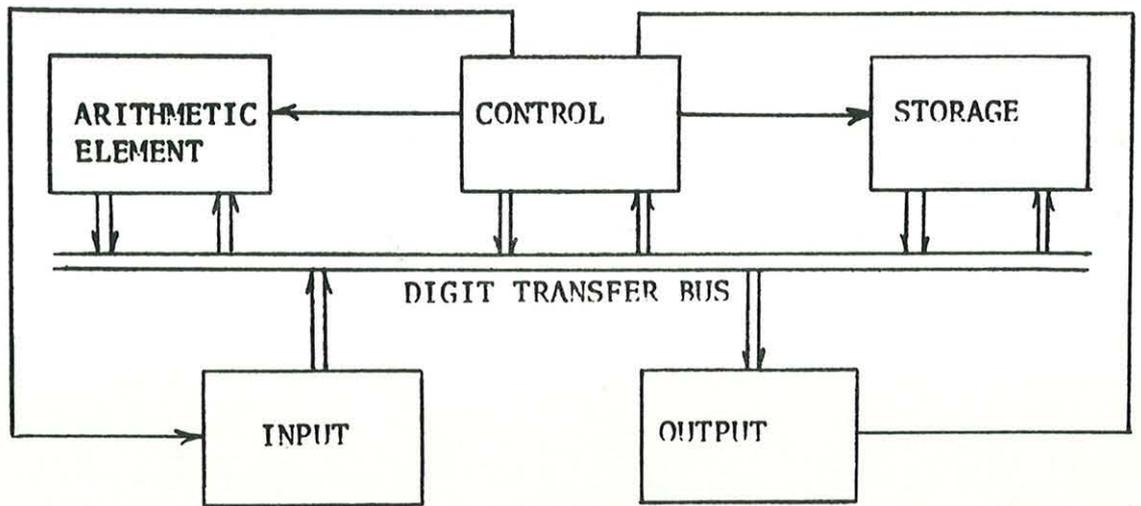


Figure 2. Simplified computer block diagram
Whirlwind I

Figure 2

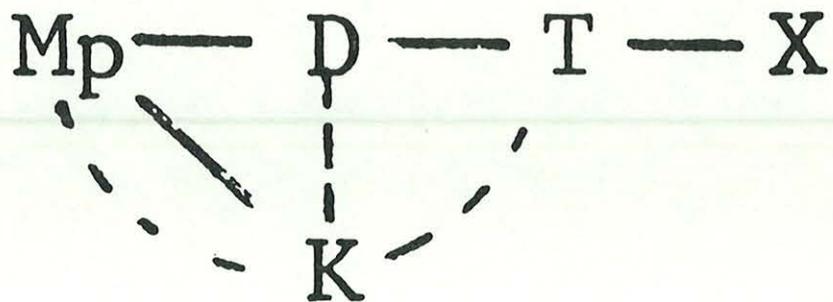


Figure 3

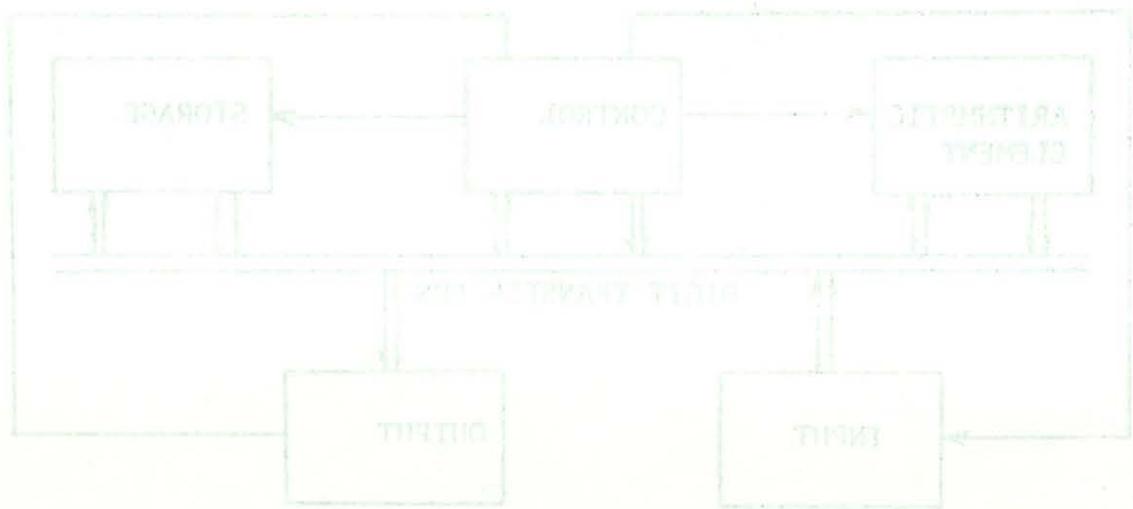


Figure 2. Simplified computer block diagram
Whitwind I

Figure 1

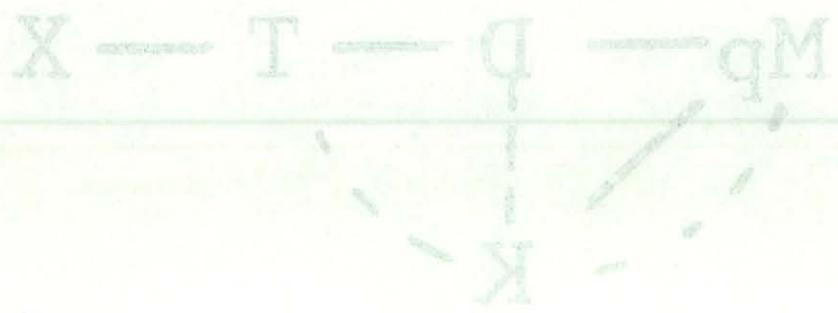
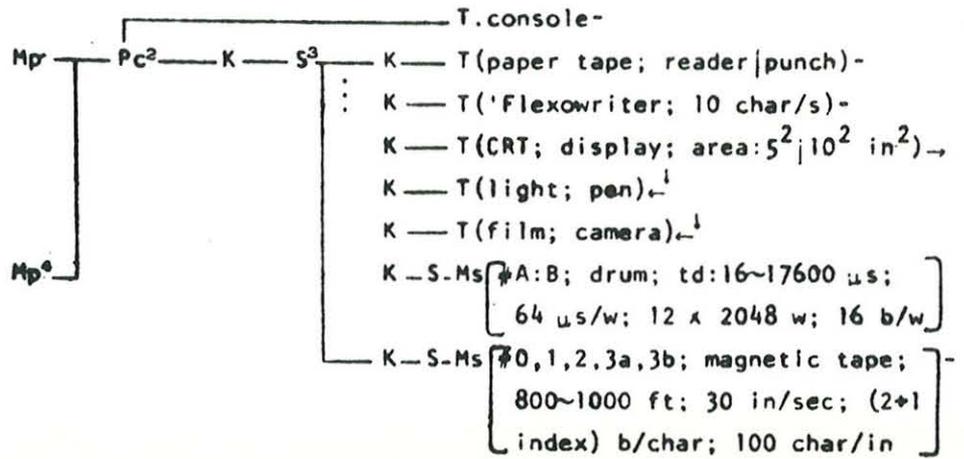


Figure 3

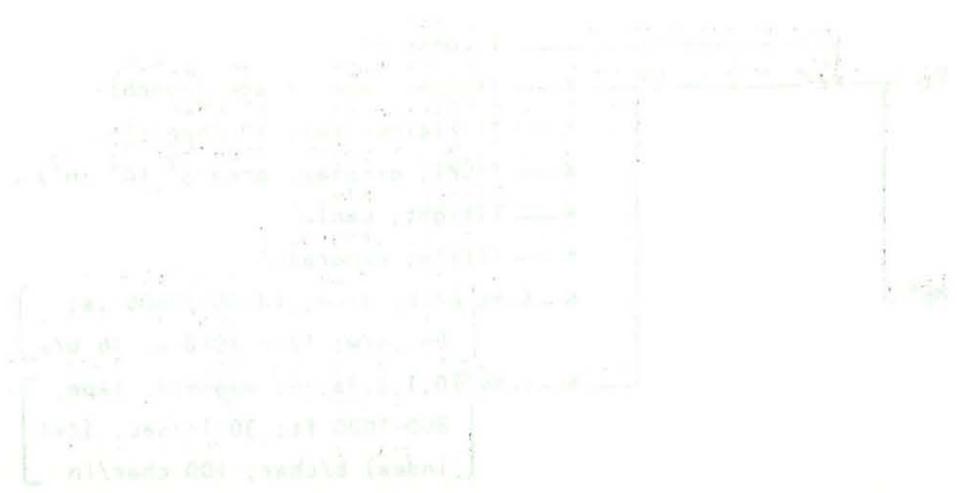


- ¹M(toggle switch; 8 us/w; 32 w; 16 b/w)
- ²Pc(50 kop/s; 16 b/w; 1 instruction/w; 1 address/instruction;
M.processor state(3 w); technology: vacuum tube; 1948 ~ 1966)
- ³S(fixed; from: Pc; ~~to~~: 8 K; concurrency: 1)
- ⁴Mp(#0:1; core; 8 us/w; 1024 w; 16 b/w; taccess: 2 us)

Figure 4

$$C := Mp - Pc - T$$

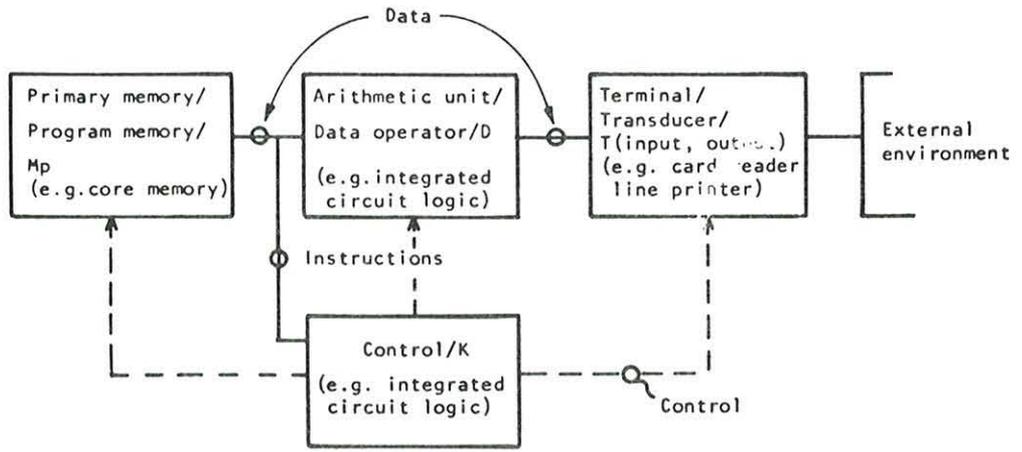
Figure 5



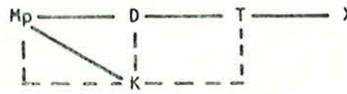
(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (28) (29) (30) (31) (32) (33) (34) (35) (36) (37) (38) (39) (40) (41) (42) (43) (44) (45) (46) (47) (48) (49) (50) (51) (52) (53) (54) (55) (56) (57) (58) (59) (60) (61) (62) (63) (64) (65) (66) (67) (68) (69) (70) (71) (72) (73) (74) (75) (76) (77) (78) (79) (80) (81) (82) (83) (84) (85) (86) (87) (88) (89) (90) (91) (92) (93) (94) (95) (96) (97) (98) (99) (100)

Figure 4

$C := Mp - Pc - T$

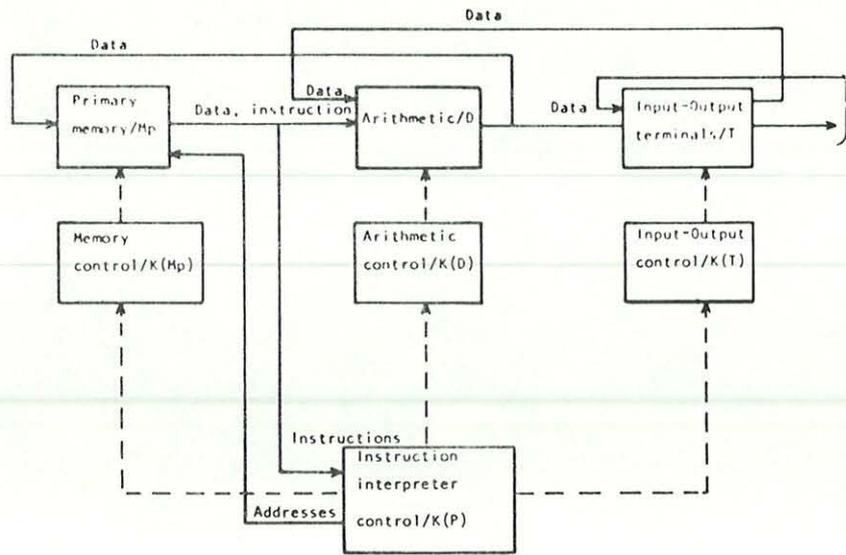


Conventional Functional block diagram

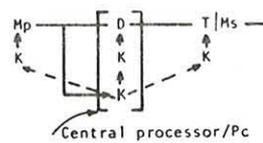


PMS notation model

Figure 6



Functional block diagram



PMS diagram

— Data flow
 - - - Control flow

Figure 7

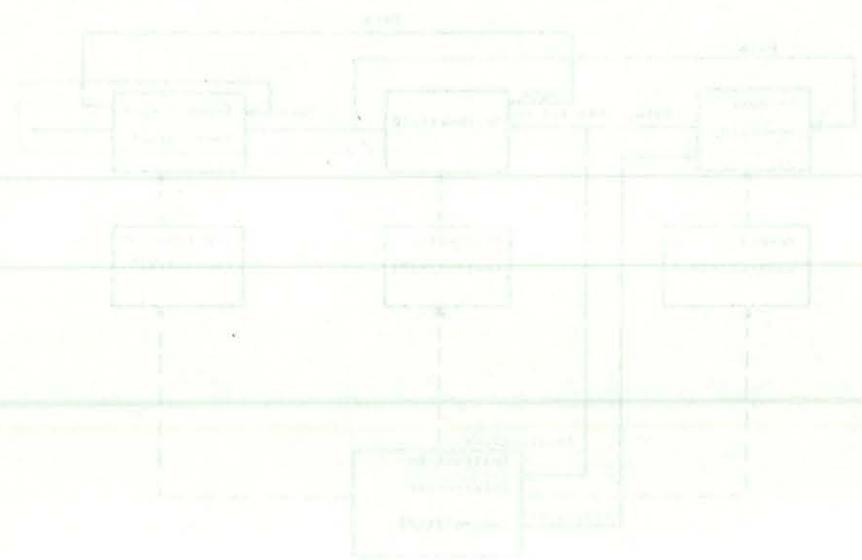


Block diagram of a control system



Relay symbol

Figure 1



Control circuit diagram



Ladder logic diagram

Figure 2

Figure 3

M.disk :=

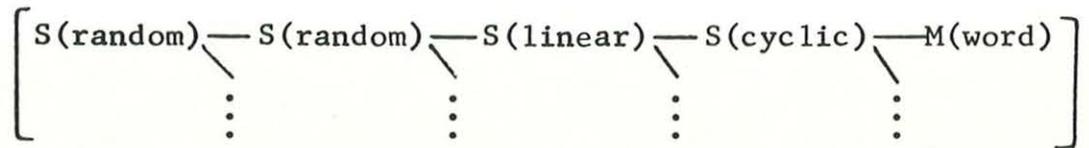


Figure 8

M(function:primary)	complete specification
M(primary)	drop the attribute, function, since it can be inferred from the value
M.primary	use the value outside the parenthesis, concatenated with a dot
M.p	use an explicitly given abbreviation, namely, primary/p (only if it is not ambiguous)
Mp	drop the concatenation marker (the dot), if it is not needed to recover the two parts (all components are given by a single capital letter--here M)

Figure 9

Model 10



Figure 8

M(primary)	drop the attribute, function, since it can be inferred from the value	complete specification
M(primary)	use the value outside the parenthesis, concatenated with a dot	
M.p	use an explicitly given abbreviation, namely, primary/p (only if it is not ambiguous)	
M.p	drop the concatenation marker (the dot), if it is not needed to recover the two parts (all components are given by a single capital letter-here M)	

Figure 9

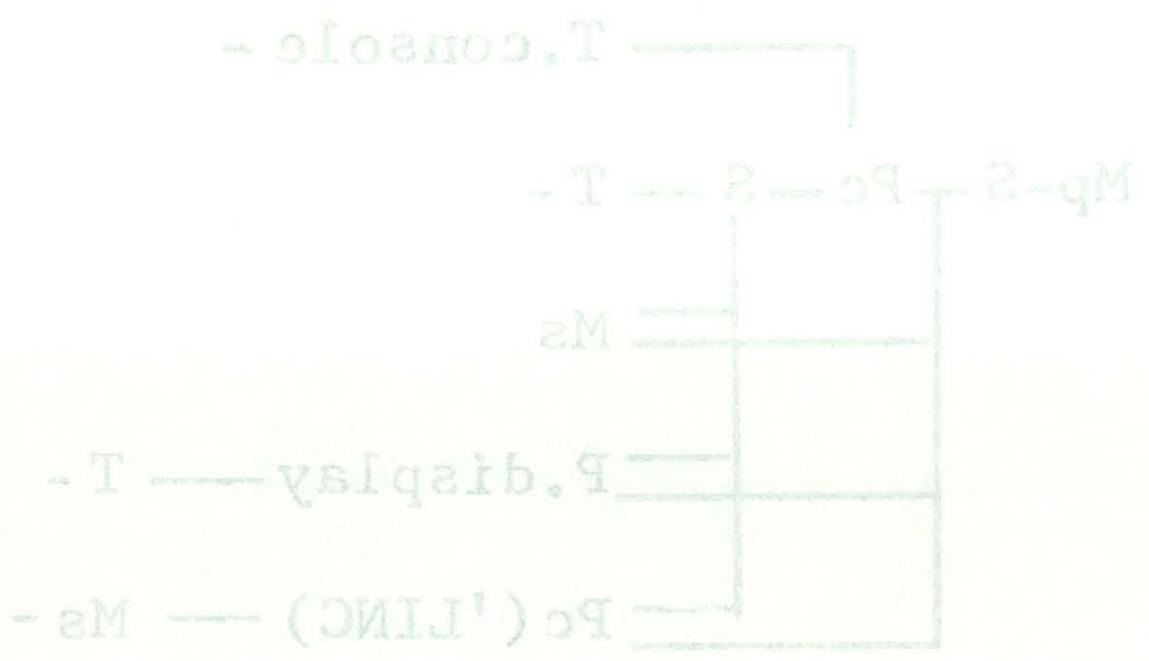


Figure 10

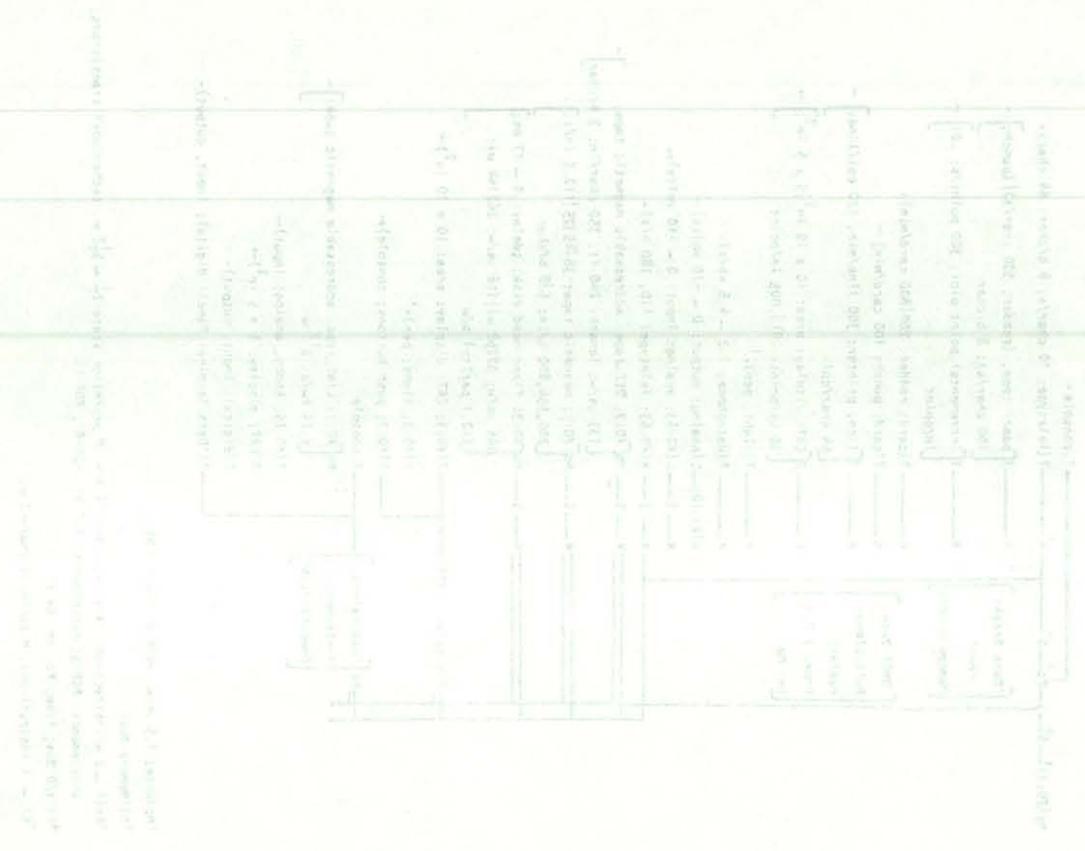


Figure 11

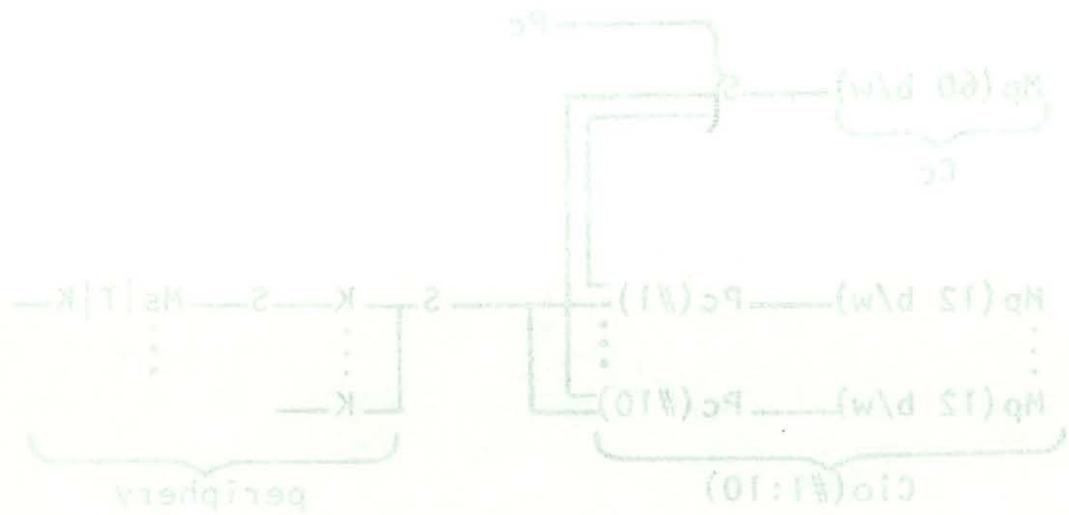


Figure 12

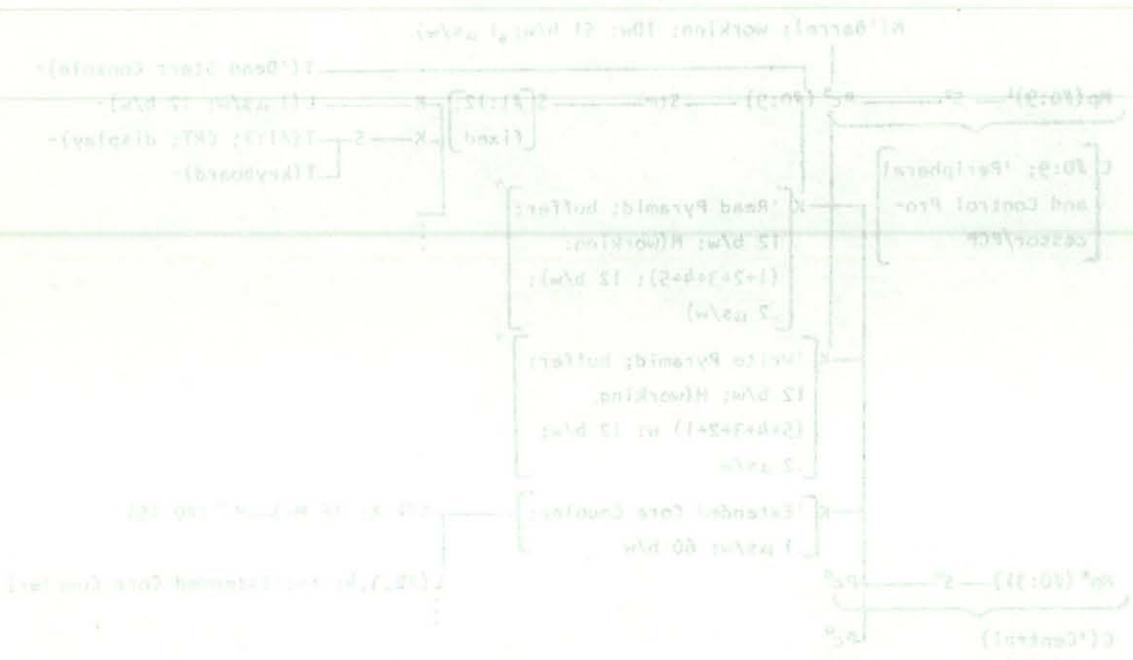


Figure 13

- ¹Mp(core; 1.0 μs/w; 4096 w: 12 b/w)
- ²S(time multiplex: .2 μs/w: 12 b/w)
- ³Pc('Peripheral and Control Processor; #0:9; time multiplex: .1 μs/w: 1 address/Instruction: 12 b/w: Mps('Program Counter, Accumulator) 1,2 w/instruction)
- ⁴Mp(core; 1.0 μs/w; 4096 w: (5 x 12) b/w)
- ⁵S(time multiplex: 0.1 μs/w: 60 b/w)
- ⁶Ms('Extended Core Storage/ECS: 3.2 μs/w: (125952 / 8) w: (8 x (60, 1 parity)) b/w)
- ⁷See Chapter 39 for operation.
- ⁸Only present in CDC 6500
- ⁹No C('Central) in CDC 6416; CDC 6500 and CDC 6400 do not have K('Scoreboard), separate D's, and M('Instruction Stack).

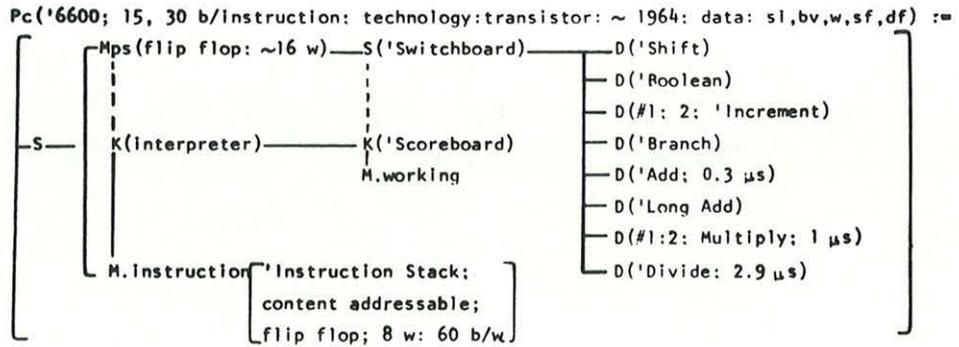
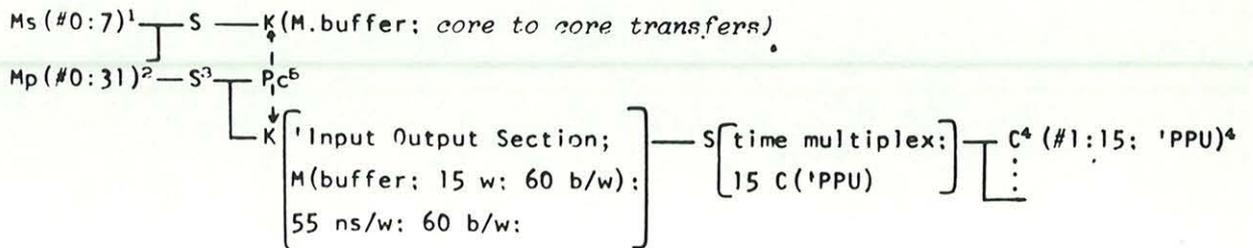


Figure 14



Basic N('CDC 7600)

Figure 15

The first stage of the design is to determine the required number of stages. This is done by calculating the required number of stages for each section of the filter. The required number of stages for each section is determined by the required attenuation and the available attenuation per stage. The required number of stages for each section is then summed to give the total number of stages required.



Figure 14

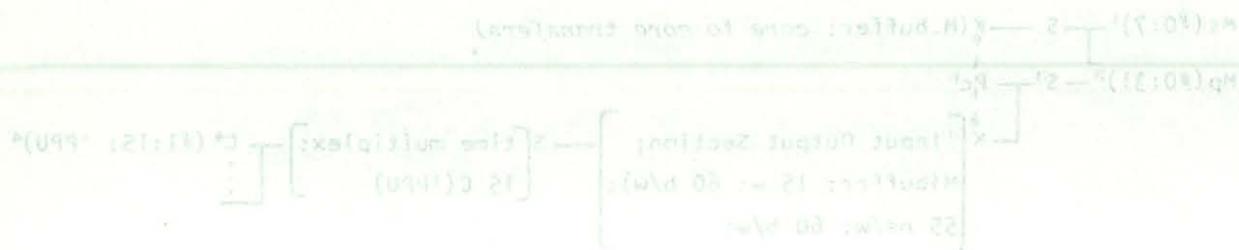


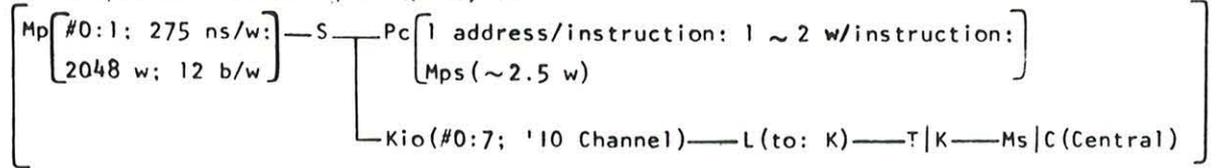
Figure 15

¹Ms ('Large Core Memory/LCM: 1.760 μ s/w: (64/8) kw: (60 x 8) b/w)

²Mp ('Small Core Memory/SCM: .275 μ s/w: 2 kw: 60 b/w)

³S (time multiplexed; 27.5 ns/w; 60 b/w)

⁴C ('Peripheral Processing Unit/PPU) :=



⁵Pc :=

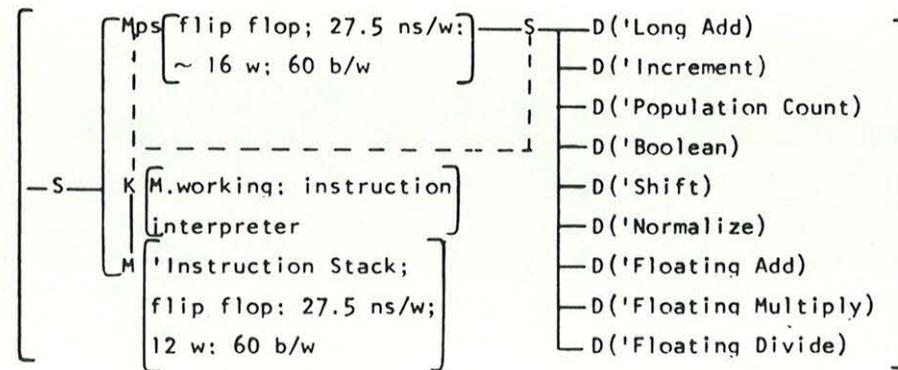
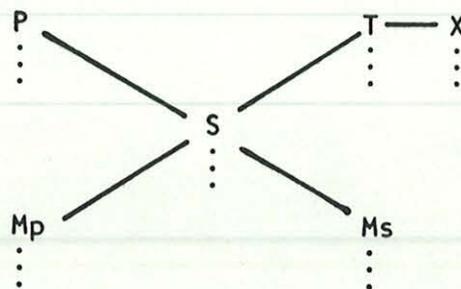


Figure 16



where P := Processor (e.g. central, input-output, display)

Mp := Primary memory (e.g. core, thin film, integrated circuit)

Ms := Secondary memory (e.g. tape, disk, drum, magnetic card)

S := Switch (e.g. multiplexor, crosspoint, bus)

T := Transducer (e.g. typewriter, line printer, card reader, display)

Figure 17

The program memory (1000-1000) is 1000 words long.
 The instruction memory (1000-1000) is 1000 words long.
 The instruction register (IR) is 1000 words long.

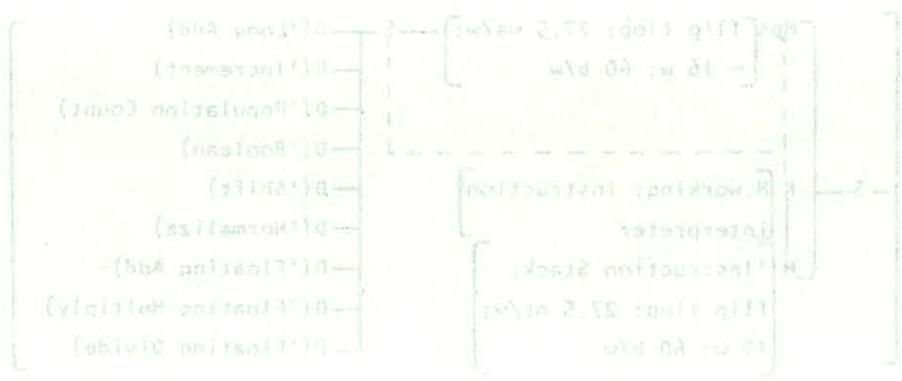
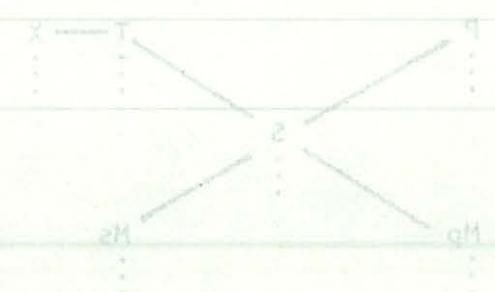


Figure 16



where P = Processor (e.g. central, input-output, display)
 M1 = Primary memory (e.g. core, thin film, integrated circuit)
 M2 = Secondary memory (e.g. tape, disk, drum, magnetic card)
 S = Switch (e.g. multiplexor, crosspoint, bus)
 T = Transducer (e.g. typewriter, line printer, card reader, display)

Figure 17

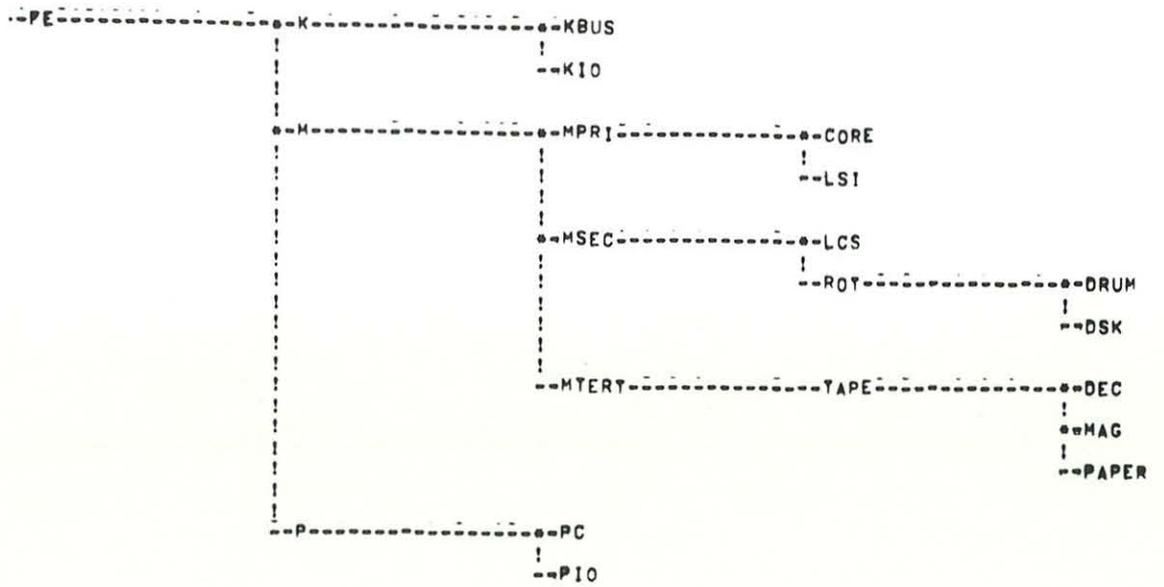


Figure 18

EDSK('REL. PRO)

PE
PE

**BACK TO YOU.

```

---
RELIB(PL,SL)
LOADING RELIB:RELIB.LIB/
STOPPING LOAD AT [ VALCOVP IDENT(PVAL,SVAL) IS(RETURN)) I<FR
RETURN)) AND COMPILING
APL PAJMAJMB;
APCP 11111;
ASL P11N11M2;
APC 21211;
REL 0.810.610.7;
REL - BEGINNING MATCHES
MINIMAL U.V. 11211;
MINIMAL U.V. 11111;
MINIMAL U.V. 11111;
RELCOMP
UPVECTOR 11211;
HAS RELIBE 0.0345600 TOTAL = 0.0345600
UPVECTOR 21211;
HAS RELIBE 0.0691200 TOTAL = 0.1036800
UPVECTOR 11211;
HAS RELIBE 0.0806400 TOTAL = 0.1843200
UPVECTOR 21211;
HAS RELIBE 0.1612800 TOTAL = 0.3456000
UPVECTOR 11111;
HAS RELIBE 0.1075200 TOTAL = 0.4531200
UPVECTOR 21111;
HAS RELIBE 0.2150400 TOTAL = 0.6681600
0.6681600
  
```

Figure 19

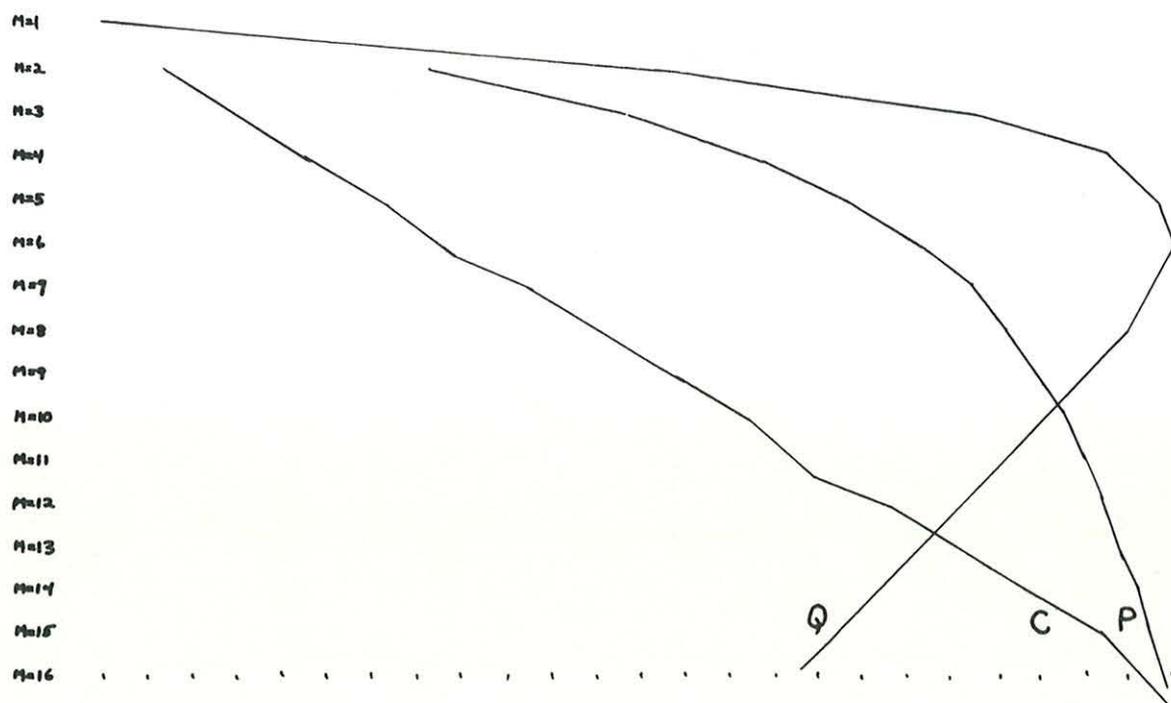


Figure 20

ISP - for Instruction-set Processor

- purpose: define the computer (instruction-set) as seen by a program (programmer)
- primitives: memory, instruction formats, data formats, effective address calculation process, instruction interpretation process, and instruction set execution definition
- uses: description, comparison, formal specification, interpretation (by machine)

Figure 21



Figure 30

129 - for instruction-set processor

- purpose: define the computer (instruction set) as seen by a program (programmer)
- primitives: memory, instruction formats, data formats, effective address calculation process, instruction interpretation process, and instruction set execution definition
- uses: description, comparison, formal specification, interpretation (by machine)

Figure 31

Pc State

AC<0:11>

L

PC<0:11>

Run

Interrupt_state

IO_pulse₁; IO_pulse₂; IO_pulse₄

Mp State

Extended memory is not included.

M[0:7777₈]<0:11>

Page_0[0:177₈]<0:11> := M[0:177₈]<0:11>

Auto_index[0:7]<0:11> := Page_0[10₈:17₈]<0:11>

Pc Console State

Keys for start, stop, continue, examine (load from mem

Data switches<0:11>

Figure 22

Instruction Format

instruction/i<0:11>

op<0:2> := i<0:2>

indirect_bit/ib := i<3>

page_0_bit/p := i<4>

page_address<0:6> := i<5:11>

this_page<0:4> := PC'<0:4>

PC'<0:11> := (PC<0:11> - 1)

IO_select<0:5> := i<3:8>

io_p1_bit := i<11>

io_p2_bit := i<10>

io_p4_bit := i<9>

sma := i<5>

sza := i<6>

snl := i<7>

Figure 23

Page 1

Page 2

Page 3

Page 4

Page 5

Page 6

Page 7

Page 8

Page 9

Page 10

Page 11

Page 12

Page 13

Page 14

Page 15

Page 16

Page 17

Page 18

Page 19

Page 20

Page 21

Page 22

Page 23

Page 24

Page 25

Page 26

Page 27

Page 28

Page 29

Page 30

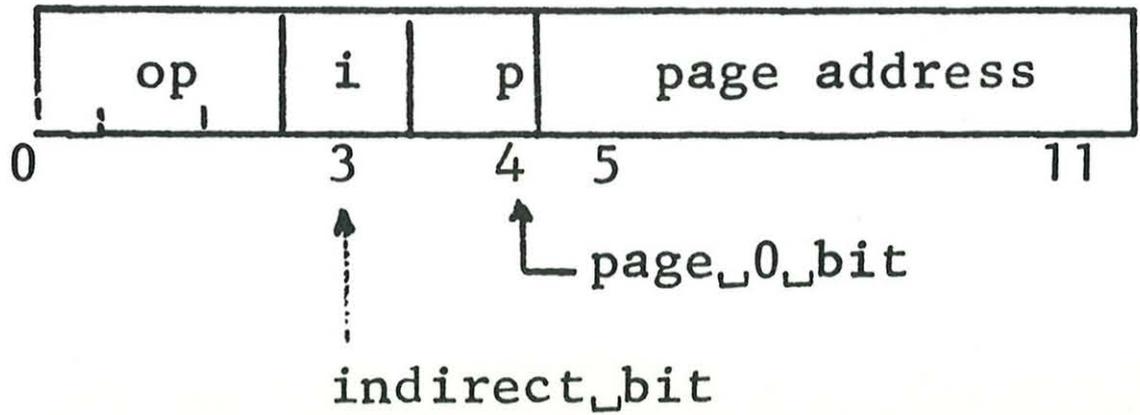


Figure 24

Effective Address Calculation Process

```

z<0:11> := (
    ¬ib → z'';
    ib ∧ (108 ≤ z'' ≤ 178) → (M[z''] ← M[z''] + 1; next);
    ib → M[z''])
z'<0:11> := (¬ ib → z''; ib → M[z''])
z''<0:11> := (page_0_bit → this_page ⊞ page_address;
    ¬page_0_bit → 0 ⊞ page_address)
  
```

μ microcoded instruction or instruction bit(s) within an instructio

Figure 25



Figure 24

Effective Address Calculation Process

```

<0:15> := (
  ~ip ~z';
  ip A (10' < z'' < 17'8) + (M[z''] + M[z''] + 1; next);
  ip ~M[z'']);
z'<0:15> := (~ip ~z'; ip ~M[z''])
z''<0:15> := (page_0_bit + this_page_address;
  ~page_0_bit + (page_address))

```

a) extended instruction or instruction bits within an instruction

Figure 25

Instruction Interpretation Process

```
Run  $\wedge \neg (\text{Interrupt\_request} \wedge \text{Interrupt\_state}) \rightarrow (\text{instruction} \leftarrow M[\text{PC}]; \text{PC} \leftarrow \text{PC} + 1; \text{next}$   
 $\text{instruction\_execution});$   
Run  $\wedge \text{Interrupt\_request} \wedge \text{Interrupt\_state} \rightarrow (\text{M}[0] \leftarrow \text{PC}; \text{Interrupt\_state} \leftarrow 0; \text{PC} \leftarrow 1)$ 
```

Figure 26

Instruction Set and Instruction Execution Process

```
Instruction_execution := (  
  and (:= op = 0)  $\rightarrow (\text{AC} \leftarrow \text{AC} \wedge M[z]);$   
  tad (:= op = 1)  $\rightarrow (\text{L} \square \text{AC} \leftarrow \text{L} \square \text{AC} + M[z]);$   
  isz (:= op = 2)  $\rightarrow (M[z'] \leftarrow M[z] + 1; \text{next}$   
     $(M[z'] = 0) \rightarrow (\text{PC} \leftarrow \text{PC} + 1));$   
  dca (:= op = 3)  $\rightarrow (M[z] \leftarrow \text{AC}; \text{AC} \leftarrow 0);$   
  jms (:= op = 4)  $\rightarrow (M[z] \leftarrow \text{PC}; \text{next } \text{PC} \leftarrow z + 1);$   
  jmp (:= op = 5)  $\rightarrow (\text{PC} \leftarrow z);$   
  iot (:= op = 6)  $\rightarrow (\text{io\_p1\_bit} \rightarrow \text{IO\_pulse\_1} \leftarrow 1; \text{next}$   
     $\text{io\_p2\_bit} \rightarrow \text{IO\_pulse\_2} \leftarrow 1; \text{next}$   
     $\text{io\_p4\_bit} \rightarrow \text{IO\_pulse\_4} \leftarrow 1);$   
  opr (:= op = 7)  $\rightarrow \text{Operate\_execution}$ 
```

Figure 27

Instruction Interpretation Process

```

Run A → (interrupt_request A interrupt_state) → (
    instruction ← M[PC]; PC ← PC + 1; next
    instruction_execution);
Run A interrupt_request A interrupt_state → (
    M[0] ← PC; interrupt_state ← 0; PC ← 1)
    
```

Figure 26

Instruction Set and Instruction Execution Process

```

instruction_execution := (
    and (:= op = 0) → (AC ← AC A M[x]);
    lrd (:= op = 1) → (LACC ← LACC + M[x]);
    lsz (:= op = 2) → (M[x] ← M[x] + 1; next
    (M[x] ← 0) → (PC ← PC + 1));
    dca (:= op = 3) → (M[x] ← AC; AC ← 0);
    jmp (:= op = 4) → (M[x] ← PC; next PC ← x + 1);
    jmp (:= op = 5) → (PC ← x);
    ldr (:= op = 6) → (
        load_ptr ← 10; next;
        load_ptr ← 10; next;
        load_ptr ← 10; next;
        opr (:= op = 7) → operator_execution
    )
    );
    
```

Figure 27

operate Instruction Set

the microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetical instruction set.

```

Operate_execution := (
  cla (:= i<4> = 1) → (AC ← 0);           clear AC. Common to all c
  opr_1 (:= i<3> = 0) → (                operate group 1
    cll (:= i<5> = 1) → (L ← 0); next    μ clear link
    cma (:= i<6> = 1) → (AC ← ¬ AC);     μ complement AC
    cml (:= i<7> = 1) → (L ← ¬ L); next  μ complement L
    lac (:= i<11> = 1) → (L ← L + 1); next μ increment AC
    ral (:= i<8:10> = 2) → (L ← L × 2 {rotate}); μ rotate left
    rtl (:= i<8:10> = 3) → (L ← L × 22 {rotate}); μ rotate twice left
    rar (:= i<8:10> = 4) → (L ← L / 2 {rotate}); μ rotate right
    rtr (:= i<8:10> = 5) → (L ← L / 22 {rotate}); μ rotate twice right
  opr_2 (:= i<3,11> = 10) → (           operate group 2
    skip condition ⊕ (i<8> = 1) → (PC ← PC + 1); next μ AC,L skip test
    skip condition := ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
    osr (:= i<9> = 1) → (AC ← AC ∨ Data switches); μ "or" switches
    hlt (:= i<10> = 1) → (Run ← 0);     μ halt or stop
    EAE (:= i<3,11> = 11) → EAE_instruction_execution) optional EAE description

```

Figure 28

