CONSIDERATIONS IN THE DESIGN OF A COMPUTER WITH HIGH LOGIC-TO-MEMORY SPEED RATIO

Leon Bloom,* Morris Cohen,* Sigmund Porter*

SUMMARY

Design assumptions of three levels of logic per nanosecond, 300 levels of logic per memory cycle, and multiprogram simultaneity lead to a machine with 1. Extremely powerful, efficient and flexible command structure; 2. No look-ahead, but a "look-aside" (a small logic speed memory invisible to the programmer); 3. Accumulators and Index Registers which exist logically but not physically; 4. Multiple sequence operations with data and program protection; 5. Extremely flexible and versatile input-output capability.

INTRODUCTION

The principal design assumption for the system described in this paper was a high logic speed to memory speed ratio. The numbers used were 10 memory cycles per microsecond and 3,000 levels of logic per microsecond, where a level of logic is "and", "or", "not", "nor", or the like. Our objective was the development of new machine organizations so that the various aspects of these organizations could be analyzed for their efficiency. Total system cost and mechanization were not considerations, although we anticipated that a large expensive machine, perhaps most suitable for use in a service center would result. We felt that the best way to utilize the large amount of logic time in each memory cycle is to have an efficiently encoded, and consequently complex, command structure so that as much of the logic capability as was practicable could be used for each memory cycle. This command structure will be described later after a discussion of principal organization features.

LOOK-ASIDE

The command structure includes variable instruction length and composition combined with a complex addressing scheme which incorporates many levels of relative and indirect addressing in any combination. This complex structure made a look-ahead type of organization infeasible. Another organizational technique called look-aside, which appears to supply at least as many benefits as would look-ahead, was adopted instead. Look-aside consists of a set of logic speed registers, which are invisible to the programmer as they are never addressed and are not addressable by the programmer. Thus, they are, philosophically, part of the main memory. The conventional memory in this system will henceforth be called the "store" and the entire memory, including look-aside, will be referred to as "main memory".

Each look-aside register consists basically of three sections: The first of these holds the contents of a store cell, the second section holds the store address of that cell, and the third is a usage indicator. (See Figure 1) The store address portions of the look-aside registers are connected to a comparator which has the ability to simultaneously compare the cell addresses in look-aside with the address of a cell requested by the system. If the address is in look-aside, an operation on the contents of that cell may take place immediately without cycling the store. If there is no matching address, the main store must be accessed. When the store is accessed, the contents of the cell and the cell address are placed in their respective places in look-aside.

* The National Cash Register Company, Electronics Division

Obviously, since the number of look-aside registers is not infinite, the placement of a cell in look-aside will often require that a cell already in look-aside be displaced.* (As a matter of fact, once the machine has operated for a short period of time, look-aside will always be full.) The first order of business then is to determine which cell is to be displaced. The classical method for determining such things is to have a digital usage algorithm which might determine, on the basis of elapsed time since the previous access or frequency of access, which cell is least likely to be required in the immediate future. This method, unfortunately, may be so time consuming that it completely destroys the advantage gained by this organization, unless the look-aside word is much larger than in the machine under discussion. Instead an analog usage indicator was chosen which might, for example, consist of having a condenser associated with each look-aside register, the condenser being charged each time that register is accessed. Thus, at any given time, the condenser with the lowest voltage has associated with it the register which has not been used for the longest period of time. A more sophisticated technique might involve two condensers per register, one of which is charged to full value and the other of which is charged with a constant increment, each time the register is accessed.

Differences With Other Machines

The organization of look-aside is reminiscent of organizations using hierarchies of memories in other systems, especially the Illiac II and the Atlas. Look-aside differs from the memory organization of both these systems. It differs from the Atlas in two ways: First, Atlas transfers information in "pages" of 512 contiguous words while look-aside concerns itself with the contents of individual store cells. (A quantative distinction which is great enough to be qualitative in its effect.) The second difference is the method of determining which portion of memory is to be displaced. As stated above look-aside makes use of an analog usage indicator while Atlas accomplishes the selection of the "page" to be displaced by a programmed digital algorithm. Atlas and look-aside have the same characteristic in that both are invisable to the programmer (in Atlas the algorithm program is part of the executive routine, and as such is not the concern of any programmer, save the executive programmer). The difference between look-aside and the use of fast registers of Illiac II is that the registers in Illiac II are not invisable and the programmer must be concerned with their operation.

Use With Single Program

If there is one program in the machine, there is nothing for the processor to do but wait for the store-access to be completed. It is anticipated however, that quite often the required cell will already be in look-aside, having found its way there in the manner described above. This will be particularly true in the case of accumulators, index registers, short program loops, or repetitively accessed data.

A simplified example of how a look-aside memory would work on a machine with conventional command structure might be useful in illustrating the manner in which a program's execution time may be reduced. Assume a single address computer with a command structure allowing for the addressing of multiple accumulators and index registers. Assume also that these registers are all buried in memory. Temporarily assume look-aside is very large; later the effect of small look-aside will be discussed. The problem is to sum a list of 50 amounts into a single grand total. Each amount is in a field one word in length.

* The word displaced, rather than replaced, in the store, is used advisedly here. Each look-aside register is provided with an extra bit, called a change bit, which indicates whether the contents of the register had been altered during its stay in look-aside. Naturally, if the store uses non-destructive read or a complete read-write cycle, an unaltered cell in look-aside need not be replaced in the store.

Such a routine might be:

(1) Clear Index Register 1

(2) Clear Accumulator 1

(3) ⌐ Add Amount into Accumulator 1 using Index Register 1

(4) | Increase Index Register 1 by 1

(5) | Compare contents of Index Register 1 to 50

(6) ⌐ Test

↓ = Finished

After the first instruction, the instruction and then the cleared index register are to be found in look-aside. The second instruction and the cleared accumulator will be found in look-aside following the execution of clear accumulator. When the third instruction is executed, it and the first amount will be placed in look-aside also. Note that we have already saved 2 memory accesses since the index register and accumulator required by instruction (3) are already in look-aside. Execution of (4) (5) and (6) bring those instructions into look-aside (we assume that the modifier (1) and the comparator (50) are contained as literals in the instruction). It can be seen that once the instructions, accumulator, and index register are in look-aside, each pass through the loop will require only one memory access, viz. that needed to pick up the next amount. This is compared to 9 memory accesses that would have been required per pass for this machine without look-aside; with live registers for accumulators and index registers 5 memory accesses per pass would have been required without look-aside.

Let us assume that the look-aside memory is 7 words long. After execution of command (4) we find that all seven words of look-aside are filled. Their contents is the four instructions, the index register, the accumulator and the first addend. When instruction (5) is brought into look-aside, the analog device will show that the cell containing instruction (1) has been inactive longest so that instruction (5) will displace instruction (1). Similarly instruction (6) will displace instruction (2). At this point, following the first pass through the routine, it should be noted that look-aside contains the four instructions of the tight loop, the required index register and accumulator and the addend. All this is without the knowledge or planning of the programmer. Each subsequent time through the loop, the addend will be replaced by the next addend because all other cells in look-aside will have been active after the cell containing the addend had been active. It should be noted that if a non-destructive type memory is used for the main memory, there is no need to take the time to restore any of the memory cells accessed during the running of this entire routine.

Multi Programming

If there are multiple programs in the machine, waiting for a store access will cause transfer of control to another program sequence which has its required instructions and data already in look-aside. When this newly activated program requires a store access, control is transferred again, perhaps to the original program, if by this time it has completed its store access and placement of the required cell contents in look-aside. This automatic transfer of control is particularly meaningful when input or output may be required before a program may continue.

Effect Of Analog Usage Indicator

It is interesting to note that because of the use of an analog usage indicator the procedure which the computer will follow need not be exactly the same each time the problem is run. This variability, however, does not alter the final result, and although it will cause different interactions

between programs, will not materially affect the composite running time of all programs. Further, as has been shown by the example, look-aside will tremendously speed up a single program with tight loops and which operates repetitively on the same data. In the case of a multiple program operation, an additional advantage lies in that there is no need to provide live registers for accumulators, index registers, etc., for each program which may be in the system at any one time. When a program becomes active, its active cells will automatically, without effort (or knowledge) of the programmer, find their way into look-aside and allow operations on these cells to proceed at logic speed.

Look-Aside Size

One other interesting facet of the look-aside structure should be mentioned before moving on to the remainder of machine organization. It is, that the size of look-aside need not be fixed for different size computers; that is, the speed of the machine is a function of the number of look-aside registers in that machine. To illustrate this, use will be made of the above example. It can be seen that if look-aside had fewer than seven cells it would be necessary during each pass through the loop to bring one or more of the instructions, the index register or the accumulator back into look-aside from memory, in addition to bringing in the new addend. Specifically, a reduction to 6 cells will cause only the index register and accumulator to be retained. The loop would in this case require seven accesses rather than the nine which would be required by a conventional machine. A five cell look-aside will retain only the index register. Conversely, more than seven look-aside cells would allow more complex and longer loops to be handled efficiently, since it is quite obvious that the number of words in look-aside must be at least equal to the number of instructions in the loop plus the number of accumulators and index registers used by the loop in order to be optimally efficient. The complex command and index register structure of the computer described in this paper drastically reduces the number of instructions required in a loop, thus reducing the required size of look-aside. The example problem shown would have required only one simple instruction used in conjunction with an index register and accumulator. It will become obvious as the instruction format and index register organization are described below that far more complex routines can be executed which still require only one instruction and one index register. A great deal more analysis will be necessary before the optimum size look-aside can be specified for this machine.

ACCUMULATORS AND INDEX REGISTERS

General Characteristics

The functional characteristics which distinguish accumulators and index registers from other cells in memory are that the former are few in number and are used relatively often. Because there are not many accumulators and index registers, it is not necessary to encode the data in the most compact manner, and it is possible to use just a few bits to specify a particular register. Because they are used often, it is desirable for them to have a short address, not be complexly encoded and be rapidly accessible. Look-aside causes any cell in memory to appear to be available at logic speed, and since often used data will remain in look-aside, it will actually be available at logic speed. Hence we have not used separate live registers, but have assigned the first thirty-two 128 bit fields of memory (for each program*) to be index registers and the next thirty-two to be accumulators. The reason for choosing 128 bits is that it is a power of 2, and hence there is a simple relation between the register number and the normal address of the data contained in the register. This is useful to a programmer because it allows him to modify the information in an index register or accumulator in a manner other than that permitted by the format of the index register or accumulator, by addressing this data in the normal manner rather than as a register.

* See following section on "X-Register"

Although thirty-two accumulators and thirty-two index registers are assigned to each program, the number of each type of register a program can have can be greater or smaller than thirty-two. In the description of the command structure it will be seen that any memory area in a program can be used as either an accumulator or an index register, but more bits will be required to address accumulators or index registers other than the basic thirty-two. If a program requires less than thirty-two accumulators and thirty-two index registers the area of memory assigned to any unused register can be used for general storage.

Accumulator Format

Each accumulator will have 120 bits for data proper and eight bits for a description byte, which contains the sign (if any), the description of the data (whether it is decimal or binary, fixed point, floating point, etc.), some interrupt criteria, and an extension bit. The extension bit is used when 128 bits is not long enough an accumulator for some particular application. If the extension bit is one, then the next 128 bit field following this accumulator is tacked on as an extension to the accumulator. The extension also has an extension bit and so we can extend an accumulator indefinitely.

Index Register Format

Index registers, like the accumulators, have relatively fixed format. The sections that comprise an index register are the value — 32 bits, increment — 16 bits, repeat counter — 16 bits, refill counter — 16 bits, refill address — 32 bits, increment sign — 1 bit, and format control — 15 bits. The value field is added into the eventual address register (where the operand address is generated). The increment field contains the amount added to the value field when the command requests that the value in the index register be incremented. The repeat counter is used to repeat a command without requiring another command to jump back to the command which is being repeated. The repeat counter keeps track of the number of times remaining that a command is to be repeated. It can be used for such things as generating tables of polynomial functions with one add command. The refill counter and refill address can be used when stepping non-uniformly through memory. The format indicates which sections of the index register are to be replaced by data from the area in memory beginning with the address in the refill address section. This replacement occurs immediately after the refill counter is counted to zero. The counting in both the repeat and refill counters is controlled by command.

X-REGISTER

We have stated in the description of the look-aside that we have multiple sequence control. In order to minimize the interference in use of index registers and accumulators by the various programs and minimize the complexity necessary in having programs completely floatable, we have what we call "X-Registers". One X-Register is assigned to each program. The sum of the X-Register and eventual address register is transferred into the memory selection register. This in effect, causes each address referenced in a particular program to be indexed to a common base address, in addition to whatever other indexing may be used. Since the accumulators and index registers are the sixty-four 128 bit fields with the lowest addresses in the main store, and each program has its own first 64 fields, consequently each program has its own accumulators and index registers. In this machine, the accumulators and index registers are not separate live registers, they have no physical location in memory and the actual number of them is a function of the number of programs which are being operated upon at a given time.

PROTECTION

A further necessity, since we have multiple programs, is to protect one program from another. The two protection problems are protecting programs from undebugged programs going on a rampage, and security. A user who rents time on a machine at a service bureau is less likely to want to have his programs and data in a machine which is also used by a competitor, if he thinks the competitor can read or alter his data. The scheme which we use for protection is to assign to each memory word 4 additional bits which contain a protection identification number. These bits are not included in the 128 data bits and we see that a memory word will consist of 128 data bits, four protection bits, and nine redundancy bits. The protection number in a memory word is used in the interpretation of commands in that word and in the use the word may be put by other commands. If the protection number is zero, then commands from this cell are supervisory program, and can access data with any protection number. They are the only commands which can execute instructions which affect protection bits. Thus, only supervisory programs may alter protection bits. If the protection number is 1, then commands are from the subroutine library and data are common constants. If the protection number is 2, the cell is unassigned. If the protection number is 3 through 15, the cell is part of an individual operating program. These operating programs may transfer or sequence only to commands having the same protection number or protection number 1. They may write only in the data portion of cells having the same protection number, and read only from the data portion of cells having the same protection number or protection number 1.

It is desirable for the whole library of subroutines to be available to all operating programs. When a transfer to a command with protection number 1 is executed, the protection number of the command which caused the transfer is stored, and as long as subroutine commands are being executed (i.e. commands with protection number 1) the machine behaves exactly as if the original protection number were the one in effect. Hence, a subroutine may affect only working storage or commands which are assigned to the operating program which called it up. Any number of operating programs may be using the same subroutine quasi-simultaneously, since a subroutine (except when called up by the supervisory program) cannot alter itself. Since subroutines can use and modify index registers, commands, and parts of commands from the operating program, the lack of self-alteration is not a problem.

EVENTUAL ADDRESS REGISTER

A central feature of the machine is the "ear" or eventual address register, around which the whole command structure is based. The ear is used to compute operand addresses. The command as shown in Figure 2 consists of a series of operand groups, each of which is made up of address groups. The information contained in the address groups is combined in the ear, generating the operand address. The first address group of an operand group states whether or not the operand is found in an accumulator, in which case there will be no other address groups required. If this address group does not specify an accumulator, then we have the option of clearing the whole ear, clearing just the portion of the ear that this address group refers to, or clearing none of the ear. We can then add the address portion (called A) of the address group into the ear, the length of the portion being determined by the 5 bit address length part of the address group. Where in the ear A is added in is determined by the address offset portion of this address group. Because of the indexing features supplied by the ear and the X-Register, it is rarely necessary to give a full 32 bit address in A, so the length of A must be specified to minimize wasted memory.

Usually it is not necessary to address to the bit, so if we are using a byte size which is a power of 2, then by eliminating least significant bits of an address and specifying the offset, we further compact the command. To achieve indirect addressing of the usual variety, instead of just adding A into

the ear, we can add the contents of A into the ear with all the combinations of clearing and not clearing and partial clearing which were described above. If we add the contents of A to the ear, then the length and offset can be in the command or with the field in main memory. The remaining variation is to put A plus the contents of the contents of the ear in the ear. In this case, the length and offset of the fields are in the command. Having specified a set of ear modifications, the command must next state whether or not the ear computations are sufficiently complete to partially execute the command. If we partially execute, there are three options as to what the content of the ear is. The content of the ear may be the operand itself, the address of the operand, or the address of the next command. In the latter case, we skip out of the command without looking at any later fields. This feature is useful when data is treated as a command. If we don't partially execute, there is the option as to whether or not the next address field of this operand field is to be contiguous with the field presently being processed; if not, then the contents of the ear is the address of the start of the next address group. Because of the possibility of jumping within an operand group, there is another option in the partial execute case, and that is whether the next operand group is contiguous with the last one, or with the last group which is contiguous with the physical start of the command.

GENERAL COMMAND STRUCTURE

The fields in the add command are described in the order they are scanned by the computer. Twenty three major fields in the add command have been numbered. All of the subfields have never been counted. Fields belong to one of three classes: command fields, operand fields, and address fields. Figure 2 illustrates each of these types of field. Many of the fields to be described need not appear at all, most may appear many times.

Command Fields

A command contains command fields (each of which appears not more than once in any given command) and any number of operand groups. An operand group contains operand fields (each appearing not more than once in any given operand group) and any number of address groups, each of which contains only address fields (each appearing not more than once in any given address group). The first field contains the command code, which describes the basic type of operation performed. "Add" does not specify whether the addition is fixed or floating or decimal or binary. This can be specified later in the command or it can be determined by the data operated on. A command programmed using the latter option can add floating point numbers or fixed point numbers or floating and fixed numbers and so forth, with no modification, or explicit testing on the part of the programmer required. Thus, the basic operation specified by the command code is modified by prefixes, suffixes, sign alteration bits, number of operands, etc. The next field determines what is done in the case of overflow and underflow. We have a field consisting of one bit which states whether or not the first operand address in the add command is used for an addend in addition to being used as a put-away. That is, if we have n operand groups in a given command, we can either put the sum of the n operands where the first operand was, or we can put the sum of just the last n — 1 operands where the first one was. Next follows an indefinite number of operand groups followed by a 2 bit link or transfer control field, which indicates whether the next command is contiguous, a return, or an arbitrary absolute transfer. If the transfer control field so indicates the last field in the command is the number of an index register used to modify the address of the next command.

Operand Fields

At each operand there is a sign alteration field which states whether the operand is signed or unsigned, if the sign should be inverted, or if the sign should be changed absolutely to plus or minus.

Next is a field which specifies whether this operand is in an accumulator (in which case, no further address groups are in this operand group) or instead, if this address group specifies an index register, or neither. If an accumulator or index register is specified, then it is also specified whether this register is addressed in long or short form. The next field specifies which ear we are going to be dealing with. There are 8 ears, each of which is addressable as data or is used to compute the operand address. Note that since the ear is not automatically cleared, it is desirable to have more than one ear. Next follows a string of address groups. Following this string is the data type or location field which tells if the data is fixed point, floating point, binary, decimal, or described with the data. The next field indicates if the operand field length is carried with the data or is in the following field of the command. Similarly the next field, byte size location, tells if the byte size is indicated with the data or in the command as the following field. The last field in an operand group tells whether or not this operand field is the last operand field in the command.

Index Register Control

The general type of address group (illustrated in Figure 2) has been explained in the section on "Eventual Address Register", above. The case that has not been described is that in which the address group calls out an index register. In this case the field following the ear clear control field is the five bit index register number. Next follows a very complex group with a number of sub-groups which are used for controlling the index register. The first sub-group, which is two bits long, tells how repeat is to be handled. In combination zero, this portion of the command will not request repeat and the repeat counter in the index register referred to is not counted. In the other three combinations, the repeat count is counted, unless it is inhibited, as will be described. These latter three combinations differ only in the initial loading of the repeat counter of the index register. In combination two, the first time we run through this command under the control of the repeat we will copy a number found later in the command into the repeat counter. In combination three, the address of the number to be put in the repeat counter is found later in the command, instead of the number of repeats being in the command itself. In combination one, the repeat counter is not changed.

In progressing through a command, when the machine first comes to a group which requests a repeat and which refers to an index register whose repeat counter contains a number greater than one, a flip-flop, which we will call R, is turned on. The state of R has no effect on groups which do not request repeat. If R is already on when the machine comes to a group which requests repeat, counting in the refill and repeat counters is inhibited and the value will not be incremented. When the end of the command is reached, R is turned off and the command will be repeated if and only if R had been on. Because of the action of R, if in each index register used for repeat in a given command, the refill and repeat counters start with the same number and the index register is refilled with the original number, then the total number of times that the command will be executed at a time is the product of all the numbers in the repeat counters. These features allow such things as the generation of a table of values of a polynomial with one add command.

The next five bits determine whether or not the ear is cleared, the value is added to the ear, the increment is added to the value, the refill counter counts, and the refill feature is enabled. Refill occurs immediately after the refill counter is counted to zero. Next appears the number of repeats, which is to be copied into the repeat counter. This field is present if and only if we have earlier stated that the number of repeats will be found initially in the command.

INPUT-OUTPUT

The last item to be described is input-output. There are facilities for extremely flexible control of the peripherals. One can plug any peripheral into any input-output receptacle on the CPU

allowing us to choose any mix of peripherals. One can use any present day peripheral, and allowance was made for future peripherals whose nature is not known at this time. There is no fixed logic directly attached to each plug, but rather a field in memory corresponds to each plug. Consequently, as far as the CPU is concerned, no wire has a fixed function. A given wire might be control, might be data, might be going in, might be coming out. Consequently whoever writes the supervisory program or executive input-output program must know the nature of the peripherals, and the supervisory program must be told what peripheral is on each plug. Not all of the operations here must be directly programmed in the usual fashion, however. In the memory field assigned to a plug, we have a data byte and a control byte which, everytime an output device requests data, are sent out along the lines from the plug to the peripheral. There is also a field which states how many and which bits comprise the control and data bytes. Associated with the data byte is a field which tells where the next data byte comes from (or goes to, in the case of an input device). Thus, at each peripheral clock time we automatically go to the plug, copy the data byte out, which contains new data, copy the control byte out, which changes rather rarely, and update the data byte.

What the programmer will call the data byte will not be what the machine designer considers data, because control information which changes often in a predetermined manner would be stored with the data, and consequently the data byte is enlarged by this factor. Control which is changed only slowly or is changed in an unpredictable fashion, will be changed by program control. There is an interrupt field associated with each plug, which determines which bits coming back from the plug should cause interruption of the main program, or to phrase it differently, initiate another program sequence by putting another control register into look-aside.

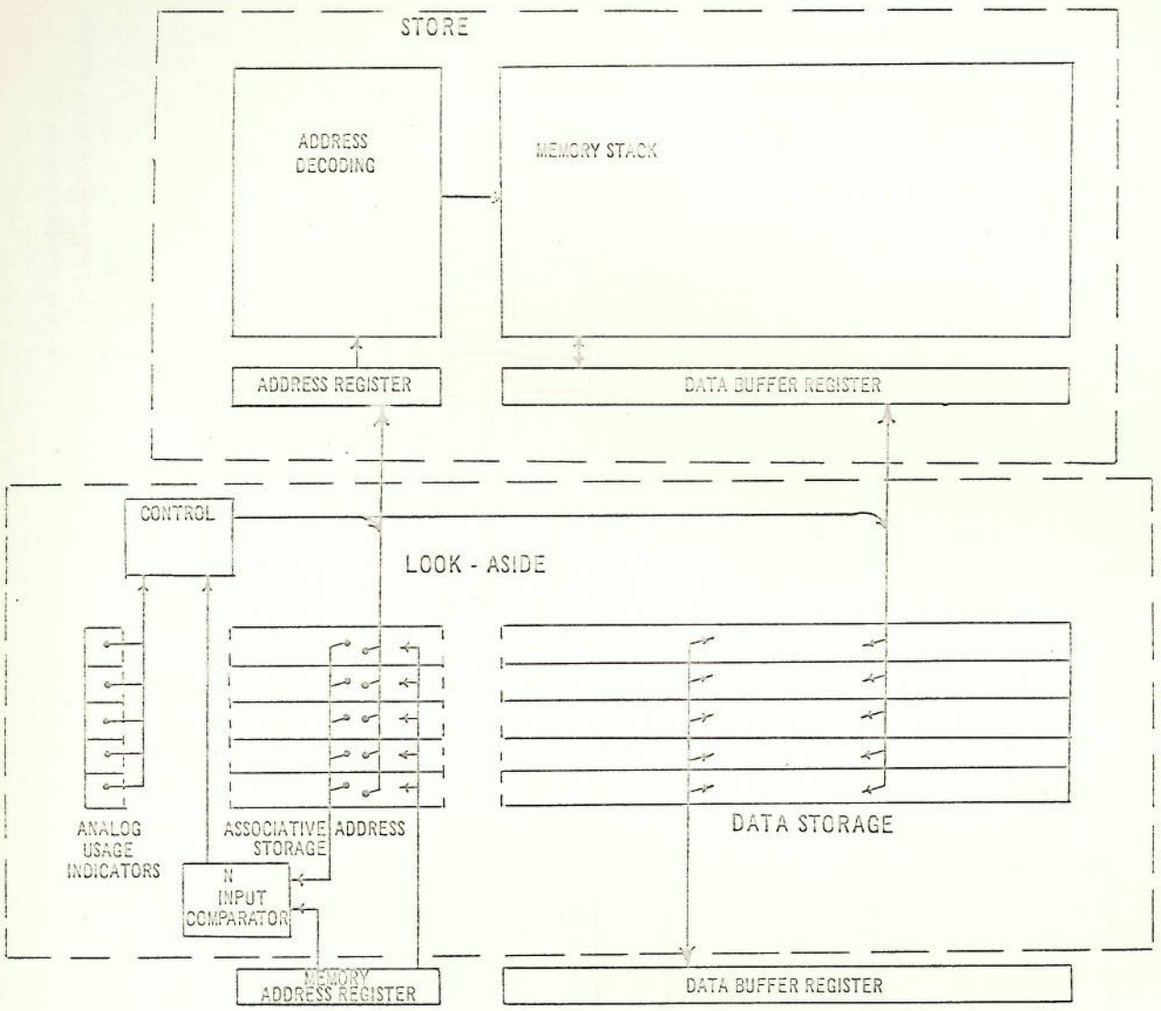
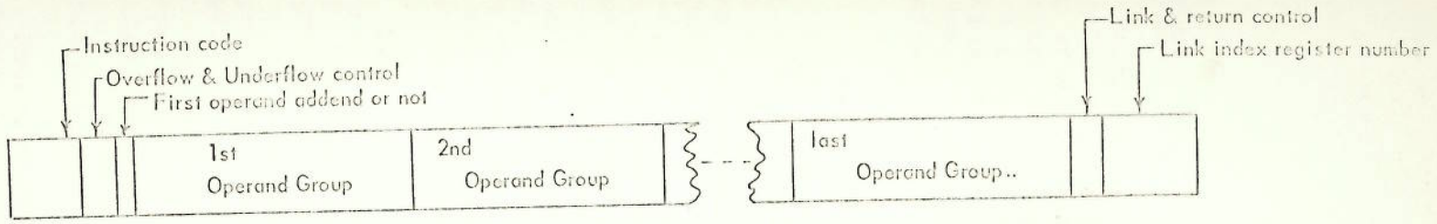
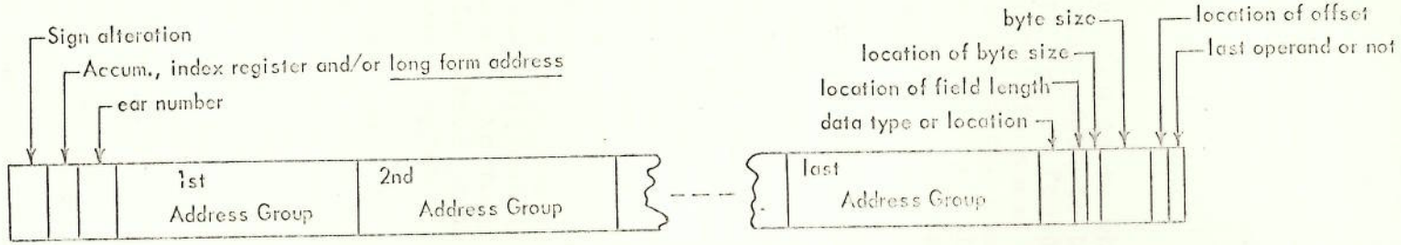
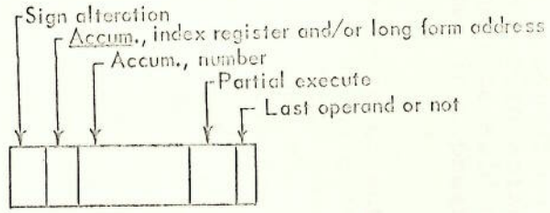


FIGURE 1. MEMORY (In this example, store is destructive read).

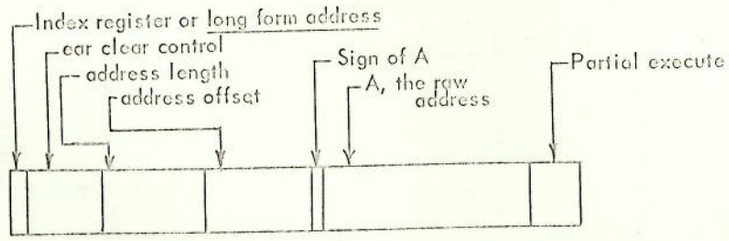


EXAMPLE OF A COMMAND





EXAMPLES OF OPERAND GROUPS



EXAMPLE OF AN ADDRESS GROUP

FIGURE 2.