ASP: a new concept in language
and machine organization*

by D. A. SAVITT, H. H. LOVE, JR., and
R. E. TROOP
Hughes Aircraft Company
Fullerton, California

*The work reported here was supported by the Air Force, Rome
Air Development Center under contracts AF30(602)-3279 and
AF30(602)-3669.

INTRODUCTION
The Association-Storing Processor (ASP) consists
of a language designed to simplify the programming
of non-arithmetic problems, together with a number
of radically new machine organizations designed to
implement this language. These machine organizations
are capable of high-speed parallel processing,
and take advantage of the low cost of memory and
logic offered by large scale integration (LSI).

The ASP concept has been developed specifically
for applications having one or more of the following
characteristics:

1. The data bases are complex in organization
   and may vary dynamically in both organization
   and content.
2. The associated processes involve complex
   combinations of simple retrieval operations.
3. The problem definitions themselves may
   change, often dramatically, during the life of
   the system.

A prime example of such an application is informa­
tion retrieval, particularly as applied to decision
making and intelligence operations.

Present solutions to problems in these application
areas frequently suffer from excessive programming
costs and time, and often from excessive running
time as well. In fact, a major problem in the design
of many information retrieval systems is that of trying
to insure that the programmers will be able to keep
pace with subsequent changes in data organization
and problem definition. These difficulties can most
generally be attributed, directly or indirectly, to the
fact that conventional digital computers, being the
sophisticated descendants of automatic calculators,
are optimized for the performance of serial opera­
tions on fixed-sized arrays of data.

An opportunity is now emerging for overcoming
these difficulties by making fundamental changes in
the organization of the processors themselves. This
is a result of the great progress being made in hard­
ware manufacturing technology. The recent advance
from discrete transistor circuits to integrated circuits
is about to be overshadowed by an even greater jump
to LSI circuitry. This new jump will result in 100­
gate and then 1000-gate circuit modules which are
little larger in size or higher in cost than the present
four-gate integrated circuit modules.

To best take advantage of this opportunity, the
authors took the approach of beginning with the de­
velopment of an appropriate language. In doing this
they were encouraged by the extraordinary ac­
complishments in the handling of complex problems
which have been made possible by some of the high­
level programming languages. The development of
machine organizations designed to implement the
language was the second step. Two organizations have
been completed to the level of a detailed logic specifi­
cation. This paper describes the general features
of the ASP language and one of the machine organiza­
tions. (Reference 1 gives a detailed language defini­
tion and a description of the other machine organiza­
tion.)

PART I - LANGUAGE

The ASP language is a machine-independent pro­
gramming language which is oriented to facilitate
the specification of queries and data-base modifica­
tions in information retrieval systems. The objective
has been to minimize the amount of procedure which must be specified to retrieve or modify data. The approach to meeting this objective is based upon the use of a data structure in which the associations between data items are expressed explicitly. The name “Association-Storing Processor” was chosen to suggest the significance of storing the data associations explicitly.

The ASP language is sufficiently problem-oriented to be used by non-programmers, and lends itself particularly well to man-machine discourse in on-line retrieval systems. In addition, its characteristics facilitate the writing of compilers to translate from higher-level problem-oriented languages into the ASP language. A close correspondence exists between the ASP language and query languages, making it relatively easy to write efficient compilers for a wide variety of such languages.*

There are two sets of antecedents for the ASP language. One is the family of list processing languages. In contrast to these languages, the ASP language could be characterized as a set processing language. In some respects, the ASP instruction is very much like a set-manipulating version of the string-manipulating “rule” used in COMIT and SNOBOL.

The other antecedent is the work being done by linguists in the field of information retrieval. Data associations or relations, similar to those used in the ASP language, are being considered, by workers in this field, as a basic building block for representing textual information.

Data representation

In the ASP language the basic unit of data is the “relation.” A relation consists of three components, an ordered pair of items and a link which specifies the type of association which relates the two items. Simple items and links are character strings. (A compound item is introduced later.) A relation may be written in the form of an ordered triple or a directed graph. For example, two items, A and B, which are related by a link, R, can be expressed as

\[(A, R, B)\]

or as in Figure 1a.

An item of data may be associated with any number of other items, with each association being expressed as a distinct relation. However, no item of data can appear by itself in an ASP expression because, in the ASP concept, an item which is associated with no other item would have no “meaning” (i.e., would convey no information). The directed graph format makes it easy to visualize complex structures of relations involving items which appear in more than one relation. (See Figure 1b.) Each distinct item appears only once in the directed graph representation of a set of relations. The physical orientation of a relation has no significance.

Because the associations between data items are explicitly stated in the data relations, the meaning of a data structure is generally apparent from an examination of the data alone. This may be illustrated by reference to Figure 1c, which shows the directed graph representation of a portion of a logistics information retrieval data base. The link labels \(ex, asg,\) and \(type\) are abbreviations for the relationships “is an example of,” “is assigned to,” and “is a type of,” respectively. The directed graph therefore conveys the following information:

- C119 is a type of TROOP TRANSPORT
- CA101 is an example of a C119
- CA102 is an example of a C119
- CA103 is an example of a C119
- CA103 is assigned to the 20TH TCS
- 20TH TCS is an example of SQUADRON.

---

*As an example, a translator has been programmed in the ASP language for translating queries expressed in a large subset of the OTC query language (developed for the Air Force 473L data processing system) into single ASP instructions. This translator consists of 23 instructions; the translation process, for a moderately complicated query, required 39 instruction executions.
An item in a relation may itself be an association between two items, and have the form of a relation. Such an item is referred to as a "compound item." In the directed graph format, a compound item is placed into a relation by connecting one end of the relation's link to the center of the link in the compound item, as illustrated in Figure 1d. Compound items may themselves have compound items as items, to any depth. As with simple items, a compound item can be connected to any number of other simple or compound items.

The ASP instruction

The ASP language instruction is a specification of a conditional transformation to be performed on the data. The transformation and the condition upon which it depends may be very elaborate.

A relatively short program of these instructions can specify any process to be performed on the data base, including those involving arithmetic, string and input-output operations.

Specifically, the ASP instruction commands that the data base be searched for matches to one set of relations and, if the search is a success, that the located data be replaced by data specified by a second set of relations. In addition, each instruction specifies the instruction to be executed next as a function of the search result (success or failure), and thereby provides the conditional branch capability needed for programs.

ASP instructions themselves are expressed as structures of relations, and are stored with the data. These structures are linked together to form programs. A special item and several special link labels are reserved for use in identifying instructions, and in distinguishing their major components. Because there is a common representation for data and instructions, one ASP program can be processed by another ASP program.

For programming convenience, an instruction format is used which differs somewhat from the representation just mentioned. Because it is also a more convenient format for discussing instruction interpretation, it is used in the present paper. This format, shown in Figure 2a, consists of four fields. The control structure field specifies the sets of relations to be located in the data base. The replacement structure field specifies the sets of relations which are to replace the relations located in the data base. To separate these two variable-sized fields, a bold arrow is placed between them. The instruction is placed in the name field. The names of the instructions which can be executed next are specified in the go-to field. The name of the next instruction for the search success case is preceded by an S. The name of the next instruction for the search failure case is preceded by an F.

The control structure

The control structure of an ASP instruction may contain any number of relations. Each item and link label in one of these relations is either a known data word, or one of the special symbols, X and Y. These special symbols are referred to as "variables," and are used to specify unknown items or link labels. The control structure in Figure 2a, for example, contains two relations with a common X variable.

The execution of an ASP instruction begins with a search for all sets of data relations which match the set contained in the control structure. For each relation in the control structure there must be a matching data relation in each data set. The criterion for a match between two relations is that each of the corresponding items and link labels must match. A pair of items or link labels are defined to be matches if they are identical or if one of them is a variable. A data item or link label which matches to a variable is referred to as a "value" for the variable. When, as in Figure 2a, two or more relations in a control structure contain a common variable, the corresponding relations in each set of matching data relations must contain a common value for this variable.

The ASP instruction actually corresponds, in power and generality, more nearly to a statement in a high-level programming language than it does to an instruction in a conventional computer.
The interpretation of the control structure may be illustrated by reference to Figures 2a and 2b. Figure 2a shows an instruction, and Figure 2b shows a data structure on which the instruction is to operate. The control structure of the instruction specifies a search for all pairs of data base relations such that

1. one relation in the pair contains an R₁ link from item A to any item, and
2. the other relation in the pair contains an R₂ link from item B to that same item.

The data base shown in Figure 2b contains three sets of relations which match to this control structure. They are

(A,R₁,F) and (B,R₂,F),
(A,R₁,G) and (B,R₂,G),
(A,R₁,H) and (B,R₂,H).

Each of these matching sets of relations contains a value for the variable X. The values are F, G, and H, respectively.

It can be seen, from this example, that the control structure provides the means to retrieve values of unknown items or link labels from the data base by specifying the context of relations in which the unknown appears. This feature of the ASP language, which may be referred to as “context addressing,” permits data to be retrieved by association. It may be contrasted to retrieval by memory location address, which is the basic retrieval criterion for conventional digital computers. The specification is basically a Boolean AND function of relations involving the unknown. However, any complex Boolean function can be specified by using the special link labels AND, NOT and OR. The details are not discussed here.

The X is used as the variable symbol when it is desired to specify the retrieval of the set of all values which can be located in the data. The Y symbol is used to retrieve just one value of the variable. The particular value selected for a Y is arbitrary as far as the programmer is concerned. (Any machine for implementing the ASP language would probably be deterministic in this respect; however, the language is not.) If a Y had been used in the example above, its value would have been one of the three items, F, G, or H.

A control structure can contain any number of relations involving any number of variables, and interrelated in any fashion. A particular relation may contain more than one variable. This permits the specification of one unknown in terms of one or more other unknowns. Subscripts may be used to distinguish the variables when more than one is used in an instruction. Examples of complex structures are given later in the paper.

As previously indicated, the result of the control structure matching operation determines the sequence of events in the execution of an ASP instruction. If at least one set of data relations can be matched to the control structure, the instruction execution is said to be a “success.” In this case the data modification specified by the instruction will be performed next, and control will then be passed to the instruction which is specified with an S prefix in the go-to field. Otherwise the matching operation is a “failure,” and control is immediately passed to the instruction specified, with a F prefix, in the go-to field.

Replacement structure

The replacement structure also may contain any number of relations, involving both known and variable items and link labels. It is required that any variable which appears in the replacement structure must also appear in the control structure as well.

If the matching of an instruction control structure is a success, the instruction execution concludes by replacing all of the data relations which matched to the control structure with the relations specified by the replacement structure. A replacement structure relation which contains a variable, indicated by the X symbol, is interpreted as a set of relations to be stored. There is one relation for each value of the variable which was identified when the control structure was processed.

The interpretation of the replacement structure may be illustrated by the example in Figure 2. The replacement structure shown there has one relation which contains the variable item whose values were determined when the control structure was processed. The values of this variable were determined from the data base in Figure 2b to be F, G, and H. This replacement structure, therefore, specifies that the three pairs of relations located by the control structure (see above) be replaced by the three single relations

(C,R₃,R)
(C,R₃,G)
(C,R₃,H).

The resultant modification in the data base can be seen by comparing the stored data base shown in the “after” picture (Figure 2c) with the data base shown in the “before” picture (Figure 2b).

Summary of instruction interpretation

The control structure specifies sets of relations to be located in the data base, and the replacement structure specifies sets of relations which are to replace the relations located in the data base. The difference
between the specification of relations to be located and the specification of relations to be stored is itself a specification of a modification to the data base. An instruction with a blank control structure will effect a simple store operation, since it only creates the new relations specified in the replacement structure. An instruction with a blank replacement structure will effect a simple deletion operation, since the data relations specified by the control structure are replaced with nothing.

**Matching data by implication**

The interpretation of the control structure specification, which is given above, refers only to the direct match feature of the ASP concept. Perhaps the most striking feature of the ASP concept is the capability that it provides for matching, indirectly, to data which may be inferred to be in the data base on the basis of statements of implication stored with the data base.

The capability for matching data indirectly in this fashion makes it possible for the control structure of an instruction to match data relations when the data base and control structure convey the same associations, but in different levels of detail or in equivalent, but not identical, formats. For example, this capability allows indirect matches to be made between a set of stored data relations and a single control structure relation on the basis of the data relation link label having transitive properties. If the data base included the relations

(C119, is a type of, TROOP TRANSPORT)
(TROOP TRANSPORT, is a type of, AIRCRAFT),

it would then be possible to find an *indirect* match to the control structure relation

(C119, is a type of, AIRCRAFT),

provided that the transitive rule of inference for the relationship "is a type of" is specified by a statement of implication stored in the data base.

The statements of implication are themselves expressed as relations of the same form as that used to represent data. The format employed is not discussed here, except to say that it involves the use of compound items. Since the statements of implication can be stored and manipulated in the same way as any other data, they may be regarded as statements of generality, as other relations are regarded as statements of particulars.

**Generated data**

In any practical device for interpreting the ASP language, there would be a number of types of relationships for which, because of sheer quantity, the explicit and implicit retrieval techniques discussed thus far would clearly be impractical. The arithmetic and quantitative relationships on pairs of numbers are good examples.

In order that the matching concept of the ASP language apply to such relationships, the associated relations must be generated by the hardware of the device, or by a subroutine, when a control structure specifies that they be located in the data base. A relation which is generated by hardware must employ a reserved-meaning item or link label (which corresponds in function somewhat to the operation code of an instruction in a conventional computer) in order to initiate the generation process. A relation which is generated by a subroutine must employ an item or link label which a programmer has previously specified as the name of this subroutine.

Typical of the relations which are generated in the ASP concept are SUM, DIFFERENCE, COUNT (the "count" of the data items which match to a control structure item), and GREATER THAN. Two of the most important types of relations which are generated are DECONC, which relates a simple item to a sequence of its component characters, and CONC, which relates a sequence of simple items to the single item which is their concatenation. These latter two relation types make it possible to apply the full power of the ASP concept to problems involving string manipulations.

**Input and output**

Input and output operations are specified by using relations with the special link labels INPUT and OUTPUT in the replacement structure. The values of variables, or the matching data relations located in the data base by the control structure, can be output by an OUTPUT relation in the replacement structure of that same instruction. (An example is given in the next section.)

Specifying an input device in an INPUT relation will cause relations to be read in from that device and stored in the data base.

**Programming examples**

Most complex queries and data modifications can be specified in a single ASP instruction. Furthermore, there is a very close structural similarity between such an ASP instruction and an equivalent query or modification statement as it would be expressed in an English-like query/maintenance language. This suggests that the task of translating from such a language to the ASP language should be relatively straightforward.

An example of a powerful one-instruction ASP program and its English-like equivalent is shown in Figure 3. This ASP instruction specifies a modification in a
hypothetical military information retrieval system. The specific modification is in the status of all examples of airmen (X1) which meet several criteria, including that they be stationed at airbases located in Europe. Whatever their status (X3), was previous to execution of this instruction, it is changed to ALERT. The correspondence between ASP relations and phrases in the sentences is shown by the circled numbers.

Note that the only difference between the control and replacement structures is in the "status" relation. The other relations are required to select the desired values of X1, but there is no intention, in this example, to alter them. Note also that it would be simple to augment this same instruction to specify the retrieval of information, as for example, the names of the affected airmen. This would be accomplished by adding the relation (X1, OUTPUT, TYPEWRITER) in the replacement structure.

An ASP program can be written to perform any process which can be described by an algorithm. ASP programs may include arithmetic operations, character manipulations, list processing, etc. Some of these operations, which are called out by reserved-meaning ASP symbols, involve indirect matches. Several of these symbols are introduced and explained in the example program which is discussed next.

Before discussing the program example it should be mentioned, explicitly, that variables are defined only within individual instructions. The values of variables are determined when the control structure containing them is processed, and new relations can be created with these variables only by the replacement structure of that same instruction. The values of variables appearing in another instruction are determined with no reference to variable values of other instructions, when that instruction's control structure is processed.

Figure 4 shows an ASP program for performing a complex search on data and outputting the results. The function of this program is to retrieve the names of all documents, published in the U.S.A. after 1958, whose principal topic is ARTIFICIAL INTELLIGENCE, and which have as descriptors at least three of the following: HEURISTIC, PROBLEM-SOLVING, THEOREM PROVING, INDUCTIVE INFERENCE, DEDUCTIVE INFERENCE.

The program begins by selecting those documents which satisfy all necessary criteria (i.e., published in U.S.A. after 1958, principal topic is ARTIFICIAL
INTELLIGENCE) and putting all of them in a particular relation (that of being an example of a "candidate"), after first deleting any other relations (X, example of, CANDIDATE) which may have already existed in data. All of this is accomplished in the first instruction (INSTR 1) of the program.

In addition, the first instruction creates a relation on each document which assigns to it a tally with an initial value of zero. Any old tally relations are deleted. Note the modifier "NON-TEST" which is appended to the link labels of two of the relations in the control structure of the instruction. These are the relations which identify any previous examples of candidates or values of tally which may have been left in data after a previous execution of the program. The NON-TEST modifier specifies that the pattern match shall not fail if no data relations corresponding to the relation it modifies are found.

---

**Figure 4 - Example of an ASP program**

From the collection of the Computer History Museum (www.computerhistory.org)
The second instruction locates all of those documents which, in addition to satisfying the requirements of the first instruction (they are "candidates"), have HEURISTIC as a descriptor. The instruction also locates the "value" of the tally for each such document and locates (indirectly, by virtue of the indirect match involving the reserved symbol SUM) the sum of one plus the value of the tally. The replacement structure then specifies, in effect, that this incremented tally value replace the previous tally value for all documents located by the control structure. The syntax in which SUM is used is fixed, so that the hardware of a processor will have predefined internal "locations" from which to fetch the numbers to be added, and a predefined location for inserting the resulting sum.

The third through the sixth instructions, which are denoted only by the three dots following the second instruction, are like the second, except that the item HEURISTIC is replaced, successively, by the items PROBLEM-SOLVING, THEOREM-PROVING, INDUCTIVE INFERENCE, and DEDUCTIVE INFERENCE.

After the execution of the sixth instruction, those documents which satisfy at least three of the five criteria given in the five previous instructions will have a tally of 3 or greater. The second to last instruction, INSTR 7, locates the documents satisfying this criteria (those having a tally value which is greater or equal to 3), and outputs their names to a device called PRINTER 1. Two reserved-meaning symbols, GTR (for "greater than") and OUTPUT appear in that instruction. The context-address logic permits the data base to be searched for matches to these interrelated relations which are predefined location for inserting the resulting sum.

The third through the sixth instructions, which are denoted only by the three dots following the second instruction, are like the second, except that the item HEURISTIC is replaced, successively, by the items PROBLEM-SOLVING, THEOREM-PROVING, INDUCTIVE INFERENCE, and DEDUCTIVE INFERENCE.

After the execution of the sixth instruction, those documents which satisfy at least three of the five criteria given in the five previous instructions will have a tally of 3 or greater. The second to last instruction, INSTR 7, locates the documents satisfying this criteria (those having a tally value which is greater or equal to 3), and outputs their names to a device called PRINTER 1. Two reserved-meaning symbols, GTR (for "greater than") and OUTPUT appear in that instruction. The context-address logic permits the data base to be searched, in turn, with each relation in the context address, and automatically outputs the relation (to PRINTER 1) of all values of X1.

The final instruction is the output instruction in case of the failure of any document to have a tally of 3 or greater (in which case there are no documents satisfying the search criteria). It outputs the relation (QUERY, result, NULL), in unformatted string form, as an ordered triple. If more elaborate formats are desired, the output function may be specified by a special subroutine, rather than by the instruction or program defining the query.

PART II ASP MACHINE

The ASP machine organization described in this part of the paper is radically different from that of a conventional digital computer. This is a result of the fact that it was designed specifically to implement the ASP language. This ASP machine organization permits the entire data base to be processed in parallel by the ASP instruction. As a result, the instruction execution time is dependent almost solely on the complexity of the control and replacement structures, but is nearly independent of the amount of data to be searched or altered by the instruction.

The ASP machine organization achieves its high speed, parallel processing capability by taking advantage of the low cost of logic implemented in large scale integrated (LSI) circuitry. The LSI circuitry is used to implement an iterative-cell, distributed logic memory. This memory provides the parallel search features of an associative memory, plus an extremely powerful inter-cell communication feature. It is this latter feature which enables the ASP machine to parallel-process the entire data base for the more complex operations involved in the instruction execution.

The key element in achieving high-speed execution of the ASP instruction is the matching of the control structure relations to the data base. These relations may be divided into two categories, relations which can be matched independently of each other and relations which are interrelated.

A control structure relation which contains no variable (X or Y) which is common to another relation can be matched independently. With the data base stored in an associative memory, it can be quickly searched in parallel for all matching relations.

However, if (as in Figure 2a) there is a variable which is common to a set of two or more control structure relations, these relations cannot be matched independently. Matching sets of data relations with common values for this variable must be found. This process can be quite time consuming, even with an associative memory, because of the need to check the individual matches to each relation for this common value condition. To perform this function in a highly parallel fashion, a new memory function called "context-addressing" was implemented by adding logic to the word cells of an associative memory structure.

The "context-address" of a variable is defined as the set of control structure relations which contain this variable, and thereby specify the context of data relations in which values for the variable must be found. The context-addressing memory function permits the memory to be searched, in turn, with each relation in the context address, and automatically tags the item/link label values which meet the criterion of being in each relation. As long as at least one variable value is found, the relations in the context address are known to have at least one match in the data base.

The context-address logic permits the data base to be searched for matches to these interrelated relations without having to check the individual matches
to each relation. This check is automatically accomplished within the memory by simultaneous communication between pairs of word cells. Because of the significance of this function, the memory of the ASP machine has been given the name "context-addressed memory."

With the intercell communication capability provided by the context-address logic, a relatively small amount of additional logic permits the parallel implementation of two other significant operations in the execution of an ASP instruction. One of these operations is the matching of control structure relations which involve two variables. The other operation is the writing of the set of relations specified by a replacement structure relation involving a variable.

**General machine organization**

The ASP machine organization is shown in Figure 5. The dominant element in this organization is the context-addressed memory. This memory stores both ASP data and programs, and, as suggested by its name, provides the capability to identify, in parallel, unknown items (and link labels) by specifying the context of relations in which the unknowns appear. This memory, its design, and the functions it performs are discussed in detail in the following sections.

Associated with the context-addressed memory are a compare register and a mask register. These registers perform functions which are similar to the functions performed by such registers in an associative memory. The compare register holds the word used to content search the memory, and also serves as a memory data register. The mask register holds the word used to mask specified bit positions so that they will not be considered in a content search of the memory.

A relatively small read-only memory is employed to store a micro-program for executing the ASP instructions. The micro-instructions include eight functions which are performed by the context-addressed memory (e.g., search, read, context address, etc.), plus several control functions which are performed external to the memory. The micro-program retrieves the components of the current ASP instruction from the context-addressed memory, executes this instruction on the data stored in that same memory, selects the next instruction to be executed, and then recycles and repeats the process. The one micro-program executes all ASP instructions, regardless of their complexity.

An arithmetic unit performs the wired-in functions (e.g., SUM, etc.) when the micro-program encounters relations in an ASP instruction which call out these functions. The complexity of this unit depends upon the complexity of the wired-in functions desired for a particular application. It could be as simple as an accumulator, or as complex as a stored-program digital computer. It is also possible to include basic parallel arithmetic capabilities in the cells of the context-addressed memory, using techniques which have been developed for associative memories, and thereby eliminate the external arithmetic unit entirely.

**Context addressed memory**

The context-addressed memory consists of a square array of identical storage cells which are interconnected both globally and locally. Each cell contains both memory and logic circuitry. The memory circuitry stores either an item, link label, or a relation, plus tag bits. The main purpose of the logic circuitry is to perform the comparison operations which are required to implement global searches of the array and local inter-cell communication.

The two types of cell interconnections are diagrammed in Figure 6. The contents of the compare and mask registers and the output of the micro-instruction decoder are distributed to all cells on the global busses. The signals involved in the inter-cell communication are propagated from cell-to-neighboring cell via the local lines. As indicated in the Figure 6b, each cell can propagate a signal only to its immediate neighbors to the north and to the west. The cell at the top of each column is connected to the cell at the bottom of that column, and the cell at the west end of each row is connected to the cell at the east end of that row. As a result, any cell can communicate, through a chain of intermediate cells, to any other cell in the array.

**Cell memory**

The memory of the storage cell is divided into five equal-length fields. Three of these fields are used to store ASP data and instruction information, and the

![Figure 5 - An ASP machine organization](image-url)
The directed graph shown below the memory array represents the same data as is shown stored in the array. As an example of the correspondence between representations, consider the relation 

\[(A, R1, C)\]

This relation is said to be stored in cell 1, 2. The actual contents of this cell are:

- \(A = 1, 1\) = address of cell containing "A"
- \(R1 = 1, 4\) = address of cell containing "R1"
- \(C = 4, 2\) = address of cell containing "C"

As another example, the CS relation between \(I\) and the compound item is 

\[(A, R1, X)\]

is stored in cell 4,1.

**Cell logic**

The cell logic performs comparison operations, and generates and propagates the inter-cell communication signals. In addition, it records the results of a comparison and performs Boolean operations on the results of successive comparisons.

Comparison operations are performed by a single comparison network. This network is used to compare the global input signal to the contents of any one of the cell's five memory fields or to the cell's wired-in address. The same network is also used to compare the local input signal to the cell's wired-in address.

---

From the collection of the Computer History Museum (www.computerhistory.org)
The result of a comparison operation performed within the cell is recorded in a match flip-flop. A Boolean function of successive comparison operations can be recorded in a second flip-flop, called the "sequence" flip-flop. The Boolean function is obtained by taking a logical combination of the sequence and match flip-flops, and storing the result in the sequence flip-flop.

The initial generation and the propagation of inter-cell communication signals is accomplished by a pair of switching networks. One network can gate a signal which originates within the cell, or which enters the cell on the local lines (either from the south or from the east) out on to the local lines to the north. The other network can gate a signal which enters the cell on the local lines from the east out on to the local lines to the west. In addition to the switching networks, a pair of buffer registers are required to temporarily store the signals entering a cell on the input local lines.

**Inter-cell communications**

The local signals are addresses of cells. Signals are initially generated as a result of a global command which is responded to by all cells which have a particular tag set. These cells simultaneously output any one of their three stored addresses (assuming they store relations or compound items) or their own address, as specified in the command, on their inter-cell lines to the north.

The address signals then propagate from cell to cell in synchronism with a global clock, until they reach a cell with a matching wired-in address. A signal always travels north initially until it reaches the row which it specifies. It then turns left and travels west along that row until it reaches the cell which it specifies.

The direction of propagation is determined by the comparisons performed within the storage cells. A cell compares the row components of an incoming north-bound signal with its own address. If there is a match, the cell outputs the address signal to its neighbor to the west; otherwise the address signal is propagated to its neighbor to the north. A cell compares the column components of an incoming west-bound signal with its own address. If there is a match, the signal sets a flip-flop; otherwise the signal is propagated to the neighbor on the west.

Storage cells are designed to simultaneously process address signals going south-to-north and east-to-west. However, if a cell simultaneously receives an east-to-west signal and a signal from the south which wants to turn west, a conflict, called a "blockage," occurs. The cell cannot transmit two signals to the west simultaneously. The signal coming from the east is given priority and is sent on to the west. The signal from the south is blocked from entering the row and is transmitted to the north.

This blocked signal will then automatically travel around the entire array column and return to the row it wishes to enter. With the introduction of routing cells and the express routes which they control (discussed in a later section), this signal blockage does not cause excessive delay in the communication process. When a high percentage of the cells transmit signals, it is possible to have a signal blocked more than once. Preliminary analysis indicates that even in such cases the delay is not excessive.

**The memory functions**

The distributed logic and memory provided by the storage cells is designed to implement eight powerful memory functions. These functions (plus several executed external to the memory) enable the microprogram to execute ASP instructions of any complexity upon large quantities of data within the memory.

Each of the memory functions manipulates sets of storage cells. This set manipulation characteristic is suggested by the nature of the ASP instruction. The control structure can specify the selection of a set of values for a variable \( X \) which is defined by a conjunction of relations, and the replacement structure can specify the creation of a relation for each value in the set. Since the number of values for a variable may be quite large, it is imperative to parallel process the set of values without having to retrieve them from the memory.

The memory functions specify sets of storage cells either by the data or the tags in a cell memory, or by the state of a "match flip-flop" in the cell logic. Data and tags are written into cell memories by the write function. A set of dissimilar data relations can be simultaneously written into a set of cells, in many cases, by a mass write function.

The match flip-flop can be set in cells selected by the search, context address, or box-car functions, or by any Boolean combination of these functions. The pulse function is used to perform the Boolean selection. The match flip-flop can be reset in all cells by the reset function.

The remainder of this section discusses each of the eight functions. Reference is made to Figure 8, which diagrams the signal flows involved in six of these functions. The next section gives a detailed example of the context-address function.
Search

The search function is identical to the content-addressing function implemented in conventional associative memories. This function selects, in parallel, all cells whose contents match the contents of the compare register, in the field specified by the mask register. As indicated in Figure 8a, this function employs the global lines and no local lines. The match flip-flop is set in all matching cells (indicated by # in the diagram).

Context address

This function selects, in parallel, all cells which are specified in a particular data field of a set of transmitting cells. The main use of this function is to select tentative values of variables in the pattern matching of control structure relations. The set of transmitting cells contain matches to a control structure relation, and the selected set of cells contain tentative values for the variable in this relation.

The context address function is always preceded by operations which select the transmit cells, set their transmit tag, and store the specification of the field to be transmitted.

When the context address function is called out, all cells whose transmit tags are set respond by transmitting the address contained in the specified field to their neighbors to the north. Each of these address flows for the box-car operation. Each of a set of relations involving tentative values of each of the variables. (This is explained below with an example.)

The box-car function is preceded by operations which select the set of transmitting cells, set their transmit tag, and store the specification of the field to be transmitted.

When the box-car function is called out, all cells whose transmit tags are set respond by transmitting the address contained in the specified field to their neighbors to the north. In this respect the function is similar to the context address function. However, in this case, as the address signal is initially transmitted from each cell, the address of the transmitting cell is “attached” to it.

Refer to the address contained in the designated field as “A” and to the address of the transmitting cell as “B.” As A travels from cell to cell, first north and then west (until it reaches the cell whose address is “A”), B follows it, somewhat as a box-car following a locomotive.

The A signal finally reaches the cell whose address is “A.” The fate of the B signal, which is attached to the A signal, depends upon the state of the “box-car” tag in the cell. If the tag is reset (indicating that the cell had not been selected on the search which preceded this box-car operation), the B signal is destroyed. However, if the tag is set, the cell will retransmit the incoming B signal. The B signal will then travel from cell to cell until it returns to the cell whose address is “B.” When it reaches that cell it sets the cell’s match flip-flop.

The diagram of Figure 8c indicates typical signal flows for the box-car operation. Each of a set of relation cells whose transmit tags are set (marked with #) simultaneously transmits, via local lines, to a cell in another set. If a receiving cell has its box-car tag set (indicated by a * for one of the two receiving cells) the box-carred address is retransmitted. This retransmitted signal must travel across the array, back to the cell which initiated it. In the diagram only one of the original transmitting cells receives back a retransmitted signal.

As an example of the application of the box-car function, assume that the data base is being searched

---

**Figure 8—The principal memory functions**

**Box-car**

This function selects, in parallel, all members of a transmitting set of cells which specify, in a particular data field, members of a second set of cells. The main use of this function is in the pattern matching of control structure relations which contain two variables. After the sets of tentative values have been obtained for these two variables (using the context address function), the box-car function is employed to select the set of cells containing relations involving tentative values of each of the variables. (This is explained below with an example.)

The box-car function is preceded by operations which select the set of transmitting cells, set their transmit tag, and store the specification of the field to be transmitted.

When the box-car function is called out, all cells whose transmit tags are set respond by transmitting the address contained in the specified field to their neighbors to the north. In this respect the function is similar to the context address function. However, in this case, as the address signal is initially transmitted from each cell, the address of the transmitting cell is “attached” to it.

Refer to the address contained in the designated field as “A” and to the address of the transmitting cell as “B.” As A travels from cell to cell, first north and then west (until it reaches the cell whose address is “A”), B follows it, somewhat as a box-car following a locomotive.

The A signal finally reaches the cell whose address is “A.” The fate of the B signal, which is attached to the A signal, depends upon the state of the “box-car” tag in the cell. If the tag is reset (indicating that the cell had not been selected on the search which preceded this box-car operation), the B signal is destroyed. However, if the tag is set, the cell will retransmit the incoming B signal. The B signal will then travel from cell to cell until it returns to the cell whose address is “B.” When it reaches that cell it sets the cell’s match flip-flop.

The diagram of Figure 8c indicates typical signal flows for the box-car operation. Each of a set of relation cells whose transmit tags are set (marked with #) simultaneously transmits, via local lines, to a cell in another set. If a receiving cell has its box-car tag set (indicated by a * for one of the two receiving cells) the box-carred address is retransmitted. This retransmitted signal must travel across the array, back to the cell which initiated it. In the diagram only one of the original transmitting cells receives back a retransmitted signal.

As an example of the application of the box-car function, assume that the data base is being searched
for matches to a control structure containing the relation \((X_1, R_1, X_2)\). Tentative values are found for \(X_1\) and then for \(X_2\) by performing context-address functions with all relations involving these two variables. However, the constraint that values of \(X_1\) must be \(R_1\)-related to value of \(X_2\) is not applied with the context address function.

To apply this constraint, the box-car function is employed. The transmit tag is set in all relations containing the link label \(R_1\). The box-car tag is then set in each cell which is tagged as containing a value of \(X_1\). Then those cells which have their transmit tags set are commanded to transmit box-car signals with their left field address as the "locomotive." At the conclusion of the box-car operation, the only cells which have received retransmitted signals are those which contain a tentative value of \(X_1\) as the left item and \(R_1\) as the link label.

A second box-car operation is then performed with the relation cells selected on the first operation and with the cells tagged as containing tentative values of \(X_2\). At the conclusion of this box-car operation, the only cells which receive retransmitted signals are those which contain a tentative value of \(X_1\) as the left item, link label \(R_1\), and a tentative value of \(X_2\) as the right item.

A pair of context-address operations is then performed with the selected relation cells to establish the values of \(X_1\) and \(X_2\) which meet the condition of being related to each other by \(R_1\). The significance of this box-car operation is that it performs the final selection of the values of interrelated variables in a highly parallel manner, without having to process the values individually.

Read

The read function causes the contents of the compare register to be replaced by the contents of a memory cell which has its match flip-flop set. (The case in which more than one cell has its match flip-flop set is discussed below.) Only one specified field may be read out at a time.

As illustrated in Figure 8d, the read function is implemented with the local inter-cell lines. The global read command gates the north-bound local lines from storage cell 1,1 into the compare register. The read command also forces every cell whose match flip-flop is set to initiate a box-car type of operation. The "locomotive" signal for every cell is the address 1,1. The box-car signal is the field specified for read out.

When cell 1,1 receives an input it discards the locomotive signal and outputs the box-car signal, as though it were doing a box-car operation. Since the north-bound local lines from this cell have been gated to the compare register (see above), this retransmitted signal is gated directly into the compare register.

If more than one storage cell has its match flip-flop set previous to a read operation, the micro-program may either:

(a) accept only the first cell’s contents which reach the compare register, by terminating the read command at this point, or

(b) accept a stream of cell contents, passing the data through the compare register to a peripheral device.

In some cases, the micro-program must determine merely if at least one match flip-flop is set in the memory. A read function is then called out, but the micro-program only senses whether something has entered the compare register.

Write

This function writes the contents of a field of the compare register into the corresponding field of all cells whose match flip-flop is set. As indicated in Figure 8e, the compare register communicates to the cells via the global busses. The field is specified by the mask register. This function is used to write new data, and to set and reset the various tag bits (including the transmit and box-car tags). It is also used to write the field specification involved in the context address, box-car, and read functions.

Mass write

This function simultaneously writes the addresses of one set of cells into another set of cells which are tagged as being available for storing new data. The function is used when storing the set of relations specified by a replacement structure relation containing an \(X\) variable.

When the mass write function is called out by the micro-program, every cell which has its match flip-flop set transmits its own address. Each of these signals travels on local lines to the closest cell which is available for storing new data. When a signal reaches an available cell it is written into all three data fields. (This is a very powerful operation, as it simultaneously writes many unique data fields into unique cells.)

The implementation requires that the available tag of each cell (a tag bit which, when set, means the cell is available to store new data) be OR'ed with the available tag of every other cell in the same row, to form an "available line."

The mass write function is used when it is desired to write a set of relations such as (value of \(X_2\), \(R_1\), \(A\)). Previous to the mass write operation, every cell...
containing a value of $X_k$ is tagged. At entry into the mass write function, the tagged item cells send their addresses north. Each signal proceeds north until it encounters a row with a TRUE available line. It is forced to turn west here. It then proceeds along the row until it finds an available cell. The signal is then written into all three fields of this cell, the cell's available tag is reset, and its match flip-flop is set. If another signal is also on that row when the available line goes FALSE, it is forced north again, to find the next row with an available cell.

Note that for this instruction the comparison function of the storage cells is disabled, and the only controlling factor on a signal is the state of the available lines.

The diagram of Figure 8f indicates typical signal flows for the mass write operation. The two cells marked with a # symbol have been tagged to transmit one of their own addresses. The two cells marked with an * symbol have their available tag set. Both transmitted signals begin to travel towards the available cell in the middle row. However, after one signal reaches this cell the other signal turns north again, traveling towards an available cell in the next row.

The other two data fields are then written by a normal write function into all of the cells whose match flip-flops have been set. In the example above, R1 and A are written into the link label and right item fields of these cells.

**Reset**

This function resets the match flip-flop in every cell. It is used, at the beginning of almost every microprogram subroutine, to preset the memory. It also sets the “sequence” flip-flop, discussed below, in every cell.

**Pulse**

This function resets a “sequence” flip-flop in the cell’s logic if the match flip-flop is not set, and simultaneously resets the match flip-flop. The pulse function is used to perform AND and OR functions of a sequence of states of the match flip-flop.

The most significant use of this function is during the selection of tentative values for a variable which is contained in two or more control structure relations. Here it is necessary to perform the context-address function with each relation in turn and to conclude with sequence flip-flops set in those cells which received an input signal at every context address operation. At the end of each context address operation, those cells which have been addressed will have their match flip-flop set. The pulse function AND’s the match and sequence flip-flops and stores the result in the sequence flip-flop.

**Example of the context address function**

The implementation of the context addressing function may be explained with the aid of an example. Figure 9a shows a $6 \times 5$ segment of a memory array which stores the data structure shown in b of that figure. In this example, the data base will be context-addressed with the single control structure relation shown in c of the figure.

From the discussion of the ASP language it is apparent that pattern matching the relation in c to the data in b must result in selecting A, B, C, and D as the four values for the $X$ variable.

The context addressing operation is begun by searching for matches to the relation shown in c.

The field containing the $X$ is masked off during this search. Since relations are represented in memory by the addresses of their component link label and items, the memory must actually be reached for link field $= 5, 1$
right item field $= 5, 4$.

Since compound items are also stored in terms of addresses, this search criterion will already be in this form when it is retrieved from memory as the link and right item fields of a compound item linked to an I item.
When searched with the relation of c, four cells in the memory are found to contain matches: cells 5,5; 3,4; 4,3; and 5,2. A global write command is issued causing a transmit tag to be set in all matching cells. A two-bit tag field is simultaneously set in all of the matching cells to record the designation of the field containing the variable which is being context addressed (the left item field, in this example).

Next, the reset function is called out to tag all cells as containing tentative values of the variable, \( X \), which is being context addressed. (All match flip-flops are reset and all sequence flip-flops are set.) The actual context address command is now issued to all cells. This global command causes each cell which has its transmit tag set to output the specified address signal to its neighbor on the north. Then the signals travel from cell to cell, on successive clock cycles, until each signal reaches the cell which it specifies. When a signal reaches the cell which it specifies it sets the match flip-flop in that cell.

The paths taken by the signals in this example have been drawn on the array in Figure 9a. The numbers shown alongside of the paths indicate the clock cycles on which the signal transfer occurs. It has been assumed here that the array segment shown in the figure is the upper left corner of a 1024 \( \times \) 1024-cell array. (It has also been assumed that no routing cells, discussed below, are used in the array. With routing cells the number of clock cycles required in this example would be 100 cycles.)

Several aspects of the inter-cell communication are illustrated in this example:

1. The signal initiated at cell 5,2 is the address of B:6,5. Since cell 6,5 is to the south-east of the initiating cell, the signal must travel north around the array and then west around the array.

2. Two signals pass through cell 2,2 on cycle 3. No conflict occurs because one signal is going north and the other signal is going west.

3. The signal initiated at cell 4,3 attempts to turn west in cell 2,3 on clock cycle 2, but it is blocked by the signal entering this cell at this time from the east. The blocked signal is forced to travel north around the array, to finally turn west on cycle 1027.

If the variable, which is being addressed, is contained in more than one relation, the above process is repeated for each relation, starting each time with the tentative values for the variable which have been selected up to that point and using the pulse function to accomplish the AND function of successive context address operations. It can be seen that the tentative values for the variable never have to be processed individually during the context addressing of this variable.

The number of clock cycles required to context address a variable with a single relation depends upon the greatest "distance" (number of cells) which must be traversed by any one signal and by the delay through a cell. This distance depends upon the array size, upon whether or not routing cells are used (see below), and upon the number of times that a single signal may be blocked. The number of times that a single signal is blocked depends upon the percentage of cells which are transmitting signals.

**Routing cells**

The storage cell which originates a signal and the storage cell which the signal specifies may be any "distance" apart, because there is no constraint on where data is stored in the memory array. For a square array of C cells, the maximum "distance" across the array is \( 2\sqrt{C} \) cells.

An order of magnitude reduction in this maximum can be affected by the addition of a relatively small number of a second type of cell, called a "routing" cell. The function of the routing cells is to route a signal around a block of storage cells when the specified cell is beyond that block.

The routing cells are arranged in a grid pattern, dividing the array of storage cells into square sub-arrays. (See Figure 10.) Each routing cell in a grid row communicates via "local" lines to the storage cell immediately above it and to the storage cell immediately below it. It communicates via "express" lines to the routing cell directly above it and below it in the adjacent grid rows. Routing cells in grid columns are similarly connected to adjacent storage and routing cells to their left and their right. These lines are indicated for one column in Figure 10.

The routing cell performs a comparison operation upon an incoming address signal to determine whether the signal should be output onto the local line or the express line. A row routing cell will output an incoming signal onto the local line only if that signal is going to a row of storage cells which are between this routing cell and the next one to the north. The column routing cells perform the analogous function with their incoming signals, using the horizontal component, or column, addresses.

A conflict may occur when a routing cell receives both a local input and an express input simultaneously. The cell can transmit only one of these input signals out on the local output line and one out on the express output line. Therefore, if both input signals request transmission on the same output line, only one request can be honored.
If both inputs request to be output on the express line from the routing cell, the express input is given priority over the local input. As a result, the local input must "plod" along through another subarray of storage cells until it reaches the next routing cell.

If both inputs request to output on the local line from the routing cell, the local input is given priority. As a result, the signal on the express line is forced to stay on the express line. This signal will, as a result, travel completely around the memory column or row via the express route. It will return to the same routing cell \(\sqrt{C/s}\) (where \(s\) is the subarray dimension) cycles later, at which time it will most likely be able to get on the local line.

For a memory array of 1024 x 1024 storage cells the optimal subarray size has been determined to be 32 x 32 storage cells. This reduces the maximum distance that a context addressing signal must travel (assuming no blockages), by an order of magnitude, to 186 cells. The number of routing cells required is 6 percent of the number of storage cells.

REFERENCES

1. D A SAVITI H H LOVE R E TROOP
   Association-storing processor study
   Defense Documentation Center June 1966
2. D G BOBROW B RAPHAEL
   A comparison of list processing computer languages
   Comm ACM Vol 7 No 7 pp 231-240 April 1964
3. V H YNGVE
   COMIT
   Comm ACM Vol 6 No 3 pp 83-84 Mar 1963
4. D J FARBER ET AL
   SNOBOL, a string manipulation language
   Journal ACM Vol 11 No 2 pp 21-30 Jan 1964
5. P L GARVIN Technical note no 6
   Inductive methods in semantic analysis
   AD296428 Defense Documentation Center January 1963
6. R F SIMMONS
   Answering English questions by computer: a survey
   Comm ACM Vol 8 No 1 pp 53-70 Jan 1965
7. M KOCHEM ET AL
   Adaptive man-machine concept-processing
   AFRL-62-397 AD288145 Defense Documentation Center June 1962
8. R F SIMMONS J F BURGER R E LONG
   An approach toward answering English questions from text
   AFIPS conference proceedings
   Vol 2 1966 Fall Joint Computer Conference pp 357-363
   Spartan Books Washington DC 1966
9. B RAPHAEL
   SIR: a computer program for semantic information retrieval
   AFIPS conference proceedings
   Vol 26 Part 1 1964 Fall Joint Computer Conference pp 577-589
   Spartan Books Washington DC 1964
10. J R KISEDA H E PETERSEN W C SEELBACH M TEIG
    A magnetic associative memory
    IBM Journal of Research and Development Vol 5 pp 106-121 April 1961
11. P M DAVIES
    A superconductive associative memory
    AFIPS conference proceedings
    1962 Spring Joint Computer Vol 21 pp 79-88
    The National Press Palo Alto California 1962

Figure 10—Routing cells in the memory array