Design of a high-level language machine*

by G. J. BATTAREL and R. J. CHEVANCE

Cii-Honeywell Bull
Louveciennes, France

INTRODUCTION

There is at present a broad consensus that a capability-based architecture is the best approach to safe systems.5,6,8 However, at the bare hardware level, a machine should provide for a set of basic mechanisms (e.g. processor assignment, manipulation of queues of processes, definition of and manipulation on domains) without prejudging of any policy to be applied to these elements. The internal structure of the objects referred to by capabilities, the way they are organized into capability lists and the evolution of these lists in relation to various events appearing throughout the life of a program and of its possible activations should appear at some higher level, so that they can be modified without disturbing the hardware. This demands flexibility (the capability for emulating, for instance), adaptability (to future modifications) or the ability to define a machine which could be orientated, on demand, toward a certain class of applications (see for instance, the Burroughs B1700 approach). In the following, we shall be concerned mainly with this level, which we shall call “virtual architecture level.” Considering a basic architecture, which we describe briefly, we define a virtual architecture for higher-level languages (namely PL/I, COBOL and FORTRAN and the system implementation language LIS,13) called the HLL-machine in this discussion. The objectives of the HLL-machine are the following:

• To define a virtual architecture as clean and homogeneous as possible, on which compilers may become more simple, and debugging aids more powerful.
• To define a minimum of intermediate languages, ideally, just one, for supporting PL/I, COBOL and FORTRAN.
• To define a set of run-time mechanisms enforcing safety.
• To obtain more compact object programs, as compared to third generation computer systems, so as to reduce the working sets of programs.
• To abide by the error confinement principle, which states that a procedure should not have at its disposal more capabilities than actually required for its execution.
• To improve overall system performance.

CONCEPTS AND TERMINOLOGY

All the system resources are defined as objects, an object being the unit of naming, sharing and protection. An object is referenced through a capability, which specifies access rights for this object and contains a reference to its realization. Capabilities cannot be created, modified or destroyed but at the basic architecture level; they must be protected against unauthorized modification and are grouped into C-lists, which are distinct from the objects, called data sets, which are accessible to the user.**

The addressing scheme of the basic architecture provides a process with a capability list, two call/return stacks—CS (Capability Stack) for capabilities and VS (Value Stack) for other information, and a set of base registers. A procedure is considered as composed of four parts:

1. Definition of a set of objects.
2. Definition of entry points into this procedure.
3. Instructions manipulating the objects defined in 1.
4. References to entry points of other procedures.

Some of the objects defined in a procedure are purely local to it (not accessible from the outside), while others (called external objects), may be shared among several procedures. A procedure, which is an object of the cataloging system, may be referenced by means of one of its entry points.

In the following, we define a domain as a connected partial subgraph of a graph G of the above kind. A domain is therefore a construction on procedures.

Example—Given four procedures P1, P2, P3, P4 and assuming P1 references P2 and P3, P2 references P3, P3 references P2 and P4, and P4 references P2 and itself, the graph G is as shown in Figure 1a. Possible domains for this graph are G itself, or G1 or G2 as shown in Figures 1b and

---

* This study was sponsored by the French DRME, Direction des Recherches et Moyens d’Essais, under contract 74/450.

** Due to considerations of data representation compatibility with other systems, the tagged architecture scheme was not retained, in spite of its merits. The partition principle has been retained.
A *program* is a domain in which an entry point of a specific node of the graph has been specified; this node then corresponds to the main procedure. A *process* is a particular activation of a program; control enters the main procedure specified by a program and walks along edges of the graph \( G \).

In a procedure, as it is constructed by a compiler, references to external objects or entry points are symbolic: actual access to an object implies that symbolic references be transformed into (virtual) addresses. Two extreme possibilities may be considered for these transformations:

1. Prior to any program specification, all references are transformed into addresses—the complete domain (corresponding to the graph \( G \) above) is constructed at one time; this corresponds to the well known static linking strategy.
2. The transformation of references into addresses is performed on demand—the domain is constructed edge-by-edge during the process activation. This corresponds to the dynamic linking strategy, as it is implemented in MULTICS, for instance.

Any linking policy appears as a particular way of constructing domains.

With the notion of domain in mind, a classification of objects can be issued according to their scope and lifetime. The *scope* of an object corresponds to the existence of an entry for this object in the cataloging system. It may be

- **System-wide**—For instance, a file which is accessible by several processes.
- **Domain-wide**—Local to a domain (e.g. a STATIC EXTERNAL variable in PL/I).
- **Internal**—I.e., local to a procedure.

The *lifetime* of an object is related to the existence of a realization for this object; inside its scope, an object may or may not have a realization.

**ADDRESS SPACE—ACCESS PATHS**

In order to achieve protection and error confinement, the informations which constitute the address space of a process have to be grouped inside distinct areas, according to three characteristics of the objects they are to contain:

- Scope
- Lifetime
- Access rights

In the first step, we define the areas which are necessary for a PL/I-like language, then we define relations between these areas—that is, the access paths to information.

**Addressing space areas**

The different areas are introduced in the order of their creation, from the compilation of a procedure until its execution in a specific domain.

- Procedure areas are defined at compile time. They consist of three parts:
  1. A CODE area containing object code instructions.
  2. A DC area containing descriptors and constants.
  3. A LINK area, a list of capabilities containing references to this procedure realization and entry points, and to the objects that this procedure may manipulate whose scope is not internal.

The first two areas are distinguished because their access attributes are different—CODE is 'execute-only,' whereas DC is 'read only.'

As long as the procedure is not destroyed, a single copy of CODE and DC areas will ever exist (object code is reentrant), whereas a new copy of the link area is created every time the procedure is linked to a new domain.

- Domain area. A domain must have its own (domain)
LINK area, which is a concatenation of copies of procedure LINK area. This is necessary because some objects referenced by a LINK are domain-wide, and the resolution of the reference to such objects in a LINK area is domain-dependent.

- Execution time areas. A distinction is introduced according to the lifetime of the objects.

Three kinds—known from PL/1—are introduced:

- Static. An object of this class has a single realization all over the process activation.
- Automatic. The lifetime of an object in this class is related to the activation of blocks and/or procedures.
- Controlled. The existence of a realization which may consist in several ‘generations’ is under programmer’s control.

The following areas are introduced:

- ST, static area, contains the realization of static objects whose scope is not larger than the domain. System-wide objects are realized in individual areas called DS (Data Sets). The ST area, created when a domain is activated, is enlarged every time a procedure is activated for the first time. As for the domain LINK area, only a part of the ST area—called slot—is accessible at a given time; it corresponds to the currently executing (external) procedure.
- Objects of the automatic class are realized in a value stack VS if they are local to the domain, or referenced from a capability stack CS if they are not (then their realization is in a DS area).
- Controlled objects cannot be system-wide; they are realized in a value heap area, VH, to which some garbage collection algorithm must be associated.

- Dangling references. A dangling reference is a reference to an object which currently has no realization in the address space. A safe implementation must detect such a circumstance, which may arise whenever the lifetime of an access path to an object is larger than the lifetime of this object. The problem is complicated since by means of pointers, parameters or overlay definitions, several access paths to an object may coexist. A solution consists of introducing tombstones indicating whether a realization for the object currently exists, and forcing all the access paths to the object to use this tombstone.

For automatic objects, tombstones may be defined on a block basis. As regards controlled objects, one tombstone per generation is required. A major characteristic of tombstones is that, once allocated, they can never be reused. Due to this, tombstones are realized in a particular area, called the tombstone space (TS).

Access paths

Three kinds of information are manipulated in the addressing space—addresses, descriptors and values. Values may appear in ST, VS, VH, or DS areas, according to their scope and lifetime.

For protection purposes, objects are associated with descriptors; these may be completely defined at compile time, then they are implemented in the read-only DC area, or they are not fully defined before execution, then they have to be constructed dynamically in VS or VH areas, according to the object class (automatic or controlled).

Access to an object is gained through indirections across the address space areas. These may communicate between each other according to certain restrictions. Three kinds of communications have to be considered—capabilities, addresses and pointer values. The Appendix exhibits the authorized communications between areas, from which address formats can be derived. Before describing these formats (in the third section) a few words about pointer values and program structure are necessary.

Pointer values

The use of pointer values, if they are to be implemented just as addresses, is a means of violating the rules of languages concerning the manipulation of variables in relation with their types. One way to avoid this is to associate not only a reference to an object, but also a descriptor of this object, to a pointer value. By this means, a descriptor is associated with every access path to a variable, and therefore control can be exercised on the type of access which is attempted to it whatever access path is used.

Program structure handling

Block structure is taken under consideration by means of activation records (see for instance, the BURROUGHS B6700) on capability and value stacks, CS and VS. An activation record contains the following information:

- Dynamic links (DC, DV) to the block which activated this block.
- Type—block, procedure.
- Lexicographic level.
- Exit information indicating through which statements control can leave the block.
- Reference to the tombstone associated with the activation record.
- Condition enabling information.
- Display information characterizing the activation records of the blocks which statically encompass this block.
- Loop control information.
- Parameters area (see below).
- Realization of automatic objects (values and/or descriptors).

As regards parameters, since formal parameters define a new access path to an actual parameter, they should be considered as pointer values, so that the protection requirements just defined can be met.
ADDRESS FORMATS DESCRIPTORS

Addresses

Three kinds of addresses are introduced, which allow a realization of the access paths in accordance with the restrictions defined in the appendix. Bit patterns are given as an indication.

- **Operand addresses** appear inside instructions, as shown in Figure 2, where
  
  AT: Specifies the area and the kind of address defined in AD—direct or indirect.
  AD: Is a displacement within the area or slot defined by AT. If AT specifies a stack, then AD is a pair (11, d) indicating the lexicographic level and a displacement inside the activation record.

- **Internal addresses** are purely dynamic and are used for special purposes (i.e. tombstones), as shown in Figure 3, where
  
  AT: Same as before.
  V: Validity field—If set to 0, indicates a dangling reference.
  d: Displacement in the area specified by AT.

- **General addresses** may be created at compile time in the descriptor area DC, or at a run-time, to implement pointer values in particular. They may contain up to three kinds of information:
  
  - The descriptor of an object or a reference to it (D).
  - A reference to the next area of the access path (R).
  - An additional displacement (P) used either when the reference R does not lead directly to the object, or for aggregate elements (see the following).

These three informations are not always required and four types of addresses are defined:

- **Type 0** contains field D.
- **Type 1** contains fields D and R.
- **Type 2** contains fields R and P.
- **Type 3** contains fields D, R and P. This type is used to represent pointer values.

Type 3 addresses are shown in Figure 4, where

T: Type of the address.
AT: Specifies the area and the kind of address. There is a particular AT value which indicates that R contains a value, and not an address; this is used for intermediate results.
V: Validity field; V=0 indicates a null pointer value.

Descriptors

The data structures which are taken under consideration are elementary items, arrays and aggregates.

- **Elementary item descriptors** appear in the D field of Type 1 or Type 3 general addresses. They specify the different attributes of the object (arithmetic, string...). For strings, they contain the length of the area allocated to the string. For arithmetic data, base, scale, mode and precision are encoded.
- **Array descriptors** define both the array structure and the array element. This allows the detection of out-of-bounds references, and unauthorized manipulations on an element. A descriptor for a two-dimensional array is represented in Figure 5, where
  
  N: Number of dimensions.
  RVO: Relative virtual origin, i.e. displacement to the (virtual) element with subscripts 0, 0, ..., 0.
  LB, UB: Lower and upper bounds of subscripts for this dimension.
  M: Displacement from one element to the next one in the same dimension.

- **Aggregate descriptors** consist of Type 0 addresses (corresponding to the main aggregate and any sub-aggregate) and a collection of Type 1 or Type 3 addresses (one for each terminal aggregate component). The Type 0 address contains the number of elementary components in the (sub-) aggregate, and a displacement to the descriptor of the first component. An example is shown in Figure 6. By this mean, an item is accessed in the same manner, whether or not it belongs to an aggregate. An array of aggregates is described as an aggregate of arrays.
INSTRUCTION SET

The first question in designing the instruction set concerns the number of intermediate languages—should one define a single intermediate language, as it is done in most systems, or a high-level language, as it appears in the Burroughs B1700? Considering microprogram memory size, maintenance problems, training of software people and development costs, the best solution is to have a single intermediate language. As regards PL/I, FORTRAN and COBOL, a design based on PL/I includes nearly all the language constructions of COBOL and FORTRAN. Only a few additional features are necessary to cope with particular constructions. It would be necessary to estimate the performance loss—if any—when executing COBOL or FORTRAN programs onto a PL/I-based architecture.

The following objectives must be kept in mind when designing an instruction set:

- **Safety**—The error detection must be as precise as possible, and appear as soon as possible.
- **Efficiency**—The most frequently used language constructions should be treated efficiently. Thus, the designer must have some knowledge about how the language is actually used, which implies run-time measurements in source programs.
- The instruction set should provide facilities for compiler designers, and for the implementation of high level debugging aids.
- **Generality**—A few general mechanisms should be used, rather than locally-optimal implementation tricks.

---

From the collection of the Computer History Museum (www.computerhistory.org)
Let us consider the various forms of instruction streams and related hardware architectures.

- **Stack machine and zero-address instructions.** This category may be illustrated by the Burroughs B6700—single or double precision data items are stacked, otherwise descriptors are stacked. Such a structure requires that operand values or descriptors be explicitly pushed on the top of the stack prior to any computation.

- **General registers and one-address instructions—IBM/370 or CDC 6600 can be found in this category.** Its major interest lies in an optimal handling of intermediate results. It requires that registers be explicitly loaded by instructions. In a descriptor-based architecture, it is mandatory that a description register be associated to each general (data) register. As a practical consequence, the complete register must be large enough to contain the largest data item, which may be too costly for small to medium machines.

- **Three-address machines** are reminiscent of the pioneer days; they may be illustrated by the Burroughs B3700 system. They are well suited to variable length items handling but two problems require special attention—intermediate result handling and array accessing.

Intermediate results or constants are represented as a special form of general addresses called IDV (Immediate Descriptor and Value). An IDV contains both the description and the value of an item, which saves one indirection. Some instructions may specify the creation of an IDV as the result of an operation; this is specified in the instruction opcode (I option). Source program measurements show (see, for instance, Reference 8) that the arithmetic expressions are generally extremely simple and do not involve any intermediate results. This is an argument in favor of three-address code, which generates instruction sequences which are more compact (and, therefore, interpreted more efficiently), as long as no intermediate result is required, with the counterpart of a higher cost for the execution of more complex expressions.

Again, according to program measurements, it appears that one reference among three is to array element; this means that array handling instructions should be given special care. On stack or general register machines, a rather long sequence of instructions is required. We suggest that two instructions be introduced:

- **build index list**
  \[ \text{BINDL } n, \text{OPA}_0, \text{OPA}_1, \ldots, \text{OPA}_n \]
  which builds a list of \( n \) index values with operands \( \text{OPA}_1, \ldots, \text{OPA}_n \) at location \( \text{OPA}_0 \), and

- **index**
  \[ \text{INDEX } \text{OPA}_1, \text{OPA}_2, \text{OPA}_3 \]
  which selects an element from the array defined by \( \text{OPA}_1 \) with the index list found in \( \text{OPA}_2 \), and stores it in \( \text{OPA}_3 \). The instruction also performs subscriptrange checking, according to the information found in the array descriptor. In the examples of instructions exhibited below, \( \text{OP} \) stands for \( \text{OPerand} \), \( \text{OPDA} \) for \( \text{OPerand \ Descriptor \ Address} \), and \( \text{OPA} \) for \( \text{OPerand \ Address} \). Instructions may be classified as follows:

- **Computational**
  \[ \text{Examples:} \]
  1. \text{ADDI } \text{OPDA}_0, \text{OPDA}_2, \text{OPA}_3 \quad (\text{OPA}_3 = \text{OPA}_1 + \text{OPA}_2) \]
  \( I \) in \text{ADDI} indicates that an intermediate result (IDV) is created at the location specified by \( \text{OPA}_3 \)
  2. \text{INC } \text{OPDA}_0, \text{OPDA}_2 \quad (\text{OPA}_2 = \text{OPA}_2 + \text{OPA}_3) \]
  3. \text{DECI } \text{OPDA}_1 \quad (\text{OPA}_1 = \text{OPA}_1 - 1) \]

- **Computational data movement**
  \[ \text{Examples:} \]
  1. \text{MVNZ } \text{OPDA}_1 \text{ moves numeric zero to } \text{OPA}_1 \]
  2. \text{String operations, which require the use of descriptors**}
    \[ \text{Examples:} \]
    1. \text{CAT } [ ] \text{OPDA}_0, \text{OPDA}_n \text{OPDA}_2 \text{ is the concatenation of two strings} \]
    2. \text{SUBSTR } [ ] \text{OPDA}_0, \text{OPDA}_1, \text{OPA}_2 \text{ selects a substring in } \text{OPA}_2 \text{ according to two integer values (origin and length) found in } \text{OPA}_2 \text{ and creates string descriptor for the string in } \text{OPA}_2 \]

- **String move instructions**
  \[ \text{Examples:} \]
  1. \text{MOVE } [ ] \text{OPDA}_0, \text{OPDA}_2 \quad (\text{OPA}_2 = \text{OPA}_0) \]
  2. \text{MVS } \text{OPDA}_0 \text{ sets string } \text{OPA}_0 \text{ to 'spaces'} \]

- **Index manipulation—see BINDL and INDEX above.**

- **Branching—several forms of branching are considered.** Relative to the current instruction, indirect (to cope with COBOL’s PERFORM and ALTER statements), local to a block, or outside the current block. Note that this last case implies a lot of housekeeping work, especially on the stacks. A typical branch instruction contains operand(s) address(es) and branching address.
  \[ \text{Examples:} \]
  1. \text{BRLESS } \text{OPDA}_0, \text{OPA}_3, \text{BADDR} \quad (\text{if } \text{OPA}_0 < \text{OPA}_3 \text{ then go to } \text{BADDR}) \]
  2. \text{BRS } \text{OPDA}_0, \text{BADDR} \quad (\text{OPA}_0 = \text{spaces then go to } \text{BADDR}) \]

- **Procedure and block control—Call instructions specify the address of a list of parameter descriptors represented as an aggregate, and the address of an entry descriptor. For safety purposes, leaving a block resets its activation record to zero. When entering a block, the activation record is initiated by means of computational data, or string, or address movement instructions.**

- **Address movement instructions are the only way to alter the contents of a pointer value.**
  \[ \text{Examples:} \]
  1. \text{SETPTR } \text{OPDA}_0, \text{OPDA}_2 \text{ forces pointer } \text{OPA}_2 \text{ to reference item } \text{OPA}_0 \text{. The pointer value inherits the item description, and the reference to the item may be indirect, if it is accessed through a tombstone (then the pointer will reference this tombstone).} \]
  2. \text{RESETPTR } \text{OPDA}_1 \text{ resets pointer } \text{OPA}_1 \text{ to the null value.} \]
• Miscellaneous instructions contain among other things allocate and free statements on the heap, and instructions for the dynamic construction of descriptors.

CONCLUDING REMARKS

The instruction set we have briefly presented leads to compact object programs—a ratio of two has been observed for typical COBOL programs, as compared with the IBM 370 code. Obviously, this instruction set should be tuned according to further measurement and to hardware and/or firmware requirements and constraints. The design of the HLL-machine has shown that a unique architecture may support PL/I, COBOL and FORTRAN. A great attention has been given to safety problems (e.g. dangling references) and programs could be run safely on this architecture. As regards efficiency, this architecture cannot be considered as realistic unless specially tailored hardware and/or firmware be defined for it.

REFERENCES

13. LIS, The System Implementation Language LIS, CL-HE Documentation, 4549 E/EN.

APPENDIX—COMMUNICATIONS BETWEEN ADDRESS SPACE AREAS

<table>
<thead>
<tr>
<th>FROM</th>
<th>DC</th>
<th>ST</th>
<th>LINK</th>
<th>VS</th>
<th>CS</th>
<th>TS</th>
<th>VH</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LINK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>VH</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0=forbidden; 1=allowed
Horizontal arrows indicate possible beginnings for an access path; vertical arrows indicate possible terminations