INTRODUCTION TO SPLM
A Direct-Execution Aerospace Computer Architecture

R. Hokom, A. Grebert, F. Gerbstadt, and R. Bock

This is an introduction to, and an overview of, the Space Programming Language Machine (SPLM), which is an architecture for a class of machines designed for efficient execution of on-board, aerospace computation functions. It directly executes software written in the SPLM Language (SPLML), which has been specially constructed as a complete, concise notation for the application area. The goal of the SPLM development effort is to design machines that are smaller, faster and easier to program than currently

This work was supported by the U.S. Air Force under contract F04701-71-C-0200, Space and Missile Systems Organization (SAMSO), U.S. Air Force Systems Command, Los Angeles, California 90045.
available aerospace computers. The specific features and characteristics of the SPLM may have general applicability, although this was not a design criterion.

Over the years, much has been written on the subject of computers that directly execute higher-level, problem-oriented languages and the expected advantages to be gained. The SYMBOL machine, which is described in reference (4), was actually implemented in 1970. The feasibility of employing this approach to the aerospace applications area was adequately established by a study performed in 1970 and reported on in reference (5).

The SPLML syntax and semantics have been specified, and a SPLML Translator and SPLM Semantic-level Simulator have been developed. The detailed design of the SPLM and the concurrent development of a SPLM Clock-Level Simulator is now in progress. In addition, an evaluation of program memory utilization has been performed: the SPLM was shown to require 45% less program storage for a sizeable sample taken from the Minuteman on-board software compared to the currently employed computer. This effort partially verified the achievement of a machine design that is smaller and easier to program than conventional computers.

Since the SPLM is a fairly complex machine which executes a language roughly the size of ALGOL 60, it is inappropriate to completely describe it here. Reference (6) is recommended to those interested in the full exposure. Here we will only cover key or otherwise interesting features and concepts, primarily by examining the language, SPLML.

Development Background

The SPLM is intended for on-board, aerospace applications, including avionic, ballistic and spaceborne systems. A salient characteristic is that the computer is part of one or more control loops within the system and thus has a real-time interface with response time requirements sometimes measured in microseconds. Typical functions include guidance and navigation, attitude control, command processing, system monitoring, display control, tracking, and so forth.

Since the computer is carried on-board, factors such as size, weight and power are extremely important. Advanced circuit technologies (i.e., microminiturization) were given impetus initially by the needs of these applications. This problem, along with cost considerations, has invariably led to the selection of computers that are undersized for the computational processes that should be done. This has led to very expensive, highly machine-oriented software, which does not contain all planned computational functions. This situation is compounded when new missions and requirements are considered: reference (7) discusses the case of the Space Transportation System as well as the general problem area.

The U.S. Air Force-sponsored SPL development was directed towards developing a programming language for this application, plus ground-based support software, that could be compiled for existing computers. A number of compilers have been built and several versions of SPL exist. SPLML is neither a subset nor a superset of SPL, but is a new language which incorporates most of the main capabilities of SPL along with several additional ones. Initially, an attempt was made to keep the two compatible, but this proved to be contradictory to achievement of the other goals of the effort.

Most languages, including SPL, contain elements that can be traced to either machine characteristics or compiler optimization. For example, the FORTRAN DO-loop is directly traceable to index-register addressing. This inherent machine-compiler dependence tends to produce ambiguous or peculiar semantics, as well as syntactical constructs which are unrelated to the applications area (e.g., compiler directives). SPLML was developed, however, without any particular machine design or characteristics in mind. It is based primarily on providing the essential features needed to conveniently program aerospace software.

The major emphasis in designing the language was on defining its semantics. The syntax and grammar were influenced primarily by considerations of clarity, conciseness and ease of translation. The SPLML Translator, which has been implemented in APL, is a one-pass processor that performs syntax analysis and compression of source text into an encoded machine representation. A major decision made fairly early was that the SPLM and the Translator were to be considered as a multi-level interpreter of SPLML, which permitted a great deal of flexibility in specifying program dynamics to match application needs. The semantics definition differentiates between those activities performed by the Translator and those to be done by the SPLM.

Language Features

Aerospace programmers must concern themselves with a special set of problems peculiar to their field. They must think in terms of the responses their system must give to a variety of stimuli. Programmers are therefore responsible for insuring that the computer reads all of the variables, provides proper analysis of these variables and issues appropriate control signals to keep the total system operating. Normally, reaction times are critical; taking action too slowly often means irrecoverable loss. The most powerful SPLML features which have been designed to aid the programmer with these problems will now be described.

SPLML programs are composed of program blocks with declarations of all local data, control and I/O objects in the block head. This block structure and associated rules of scope for identifiers, similar to those of ALGOL 60, permit dynamic allocation and de-allocation of storage for all data objects.

Data objects have declared attributes of type, dimension...
and length; all of which may be expressed dynamically. The type, which may be boolean, cardinal, integer or real, and length of a data object govern the values which the data object may assume. These values are summarized in Figure 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Representable Values for Declared Length n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>false and true</td>
</tr>
<tr>
<td>cardinal</td>
<td>integral values from 0 to $2^n - 1$</td>
</tr>
<tr>
<td>integer</td>
<td>integral values from $-2^{n-1}$ to $2^n - 1$</td>
</tr>
<tr>
<td>real</td>
<td>all values of the form $a \times 2^b$, where $a$ is an integer of length n and $b$ is an integer of length m, m being an implementation parameter.</td>
</tr>
</tbody>
</table>

Figure 1. SPLM Data Types

Variable length capabilities allow the programmer to control numerical accuracy and to conserve storage.

The dimension attribute permits the declaration of, and later access to, data arranged in multi-dimensional, rectangular arrays composed of elements of the same type and length. Data of dissimilar type, dimension and length can be declared as belonging to nameable data sets which are, in general, tree structures.

A reference to a scalar element within an array-structured or tree-structured variable is denoted by the structure identifier followed by a sufficient number of subscripts to access the desired element. For arrays, the number of subscripts needed is equal to the rank of the array; for data sets, the number of subscripts is variable depending on the depth of the element within the tree structure.

Access to any sub-array (a smaller array within an array) is denoted by NULL subscripts to select all elements along a dimension or by subscript pairs $a:b$ to select all elements with indices $a, a+1, \ldots b$.

Formulas in SPLML use data references and an extensive set of operators to compose computational rules. All operators have equal precedence which permits left-to-right evaluation of formulas without parentheses. Parentheses in formulas denote the postponement of the immediately preceding operator until after the evaluation of the parenthesized expression.

Because of the dynamic quality of the language, all operators are defined within an operand domain, which limits the type, length and dimension of the operands. Most operators, including all of the arithmetic, boolean, and relational operators, have domains which include any combination of scalars and arrays. An operation on two scalars yields a scalar result; on a scalar and an array yields an array result of the same dimension, given by performing the operation upon the scalar and all elements of the array; on two arrays of the same dimension yields an array result of the same dimension, given by performing the operation element-by-element.

The length of the result of an arithmetic dyadic operator is equal to the length of the longer operand. The programmer may further control the result length by a special operator to cause truncation to a given length. Certain operators restrict the operand type to be consistent with their function such as the boolean AND operator. However, all operators accept a special value, NULL, as an operand and yield the NULL value if all operands have NULL value.

Procedures consist of a block and a set of zero or more formal parameters. Procedures may be called with a set of zero or more actual parameters which correspond left-to-right with the formal parameters. Any excess actual parameters are ignored; any excess formal parameters are treated as having null as a value. All procedures yield a result derived by an explicit formula or the NULL value by default.

An interrupt handler may be declared to correspond with a particular interrupt and takes the form of a block. Interrupt masking statements in SPLML allow interrupts to be selectively initiated or enabled. An interrupt handler is called by an enabled interrupt. By inhibiting an interrupt this call is deferred until enabling again. The interrupt handler call also implies an inhibit of all interrupts to give the programmer complete control over interrupt service policies.

A control set is a list of statement labels and procedure identifiers in any order. Selection of a control set element by subscribing performs a branch in control if a label is selected or a procedure call if a procedure is selected. Thus, special program control can be performed by control set selection in multiple case situations.

Facilities in the language provide for recovery by a checkpoint and restart capability, using a state holding mechanism. On checkpoint, which is automatically performed as part of the semantics of certain instructions, the processor state is stored in a state variable. A restart restores the processor state which in effect is a branch in control to a previous processor state. The intent is that a restart may be performed after an otherwise disastrous external event such as power outage or fluctuation, etc.

The I/O provisions of SPLML allow the transfer of information between the computer and its environment through a designated I/O port. A port is assumed to be connected to a device which generates or accepts binary information of a fixed length at each access. For each port, a boolean or cardinal data object, called a stream, may be declared with any dimension and the characteristic port width. A stream may be specified in a declaration so that transfers are either buffered or non-buffered. Non-buffered streams give the programmer the ability to control time-dependent I/O by delaying program execution until data transfer is completed.

No special statements were included to initiate I/O transfers. An input transfer is denoted by any reference to the input stream; an output by assignment to the output stream. Otherwise accesses to streams are semantically identical to access to ordinary data objects of the same type, dimension and length.

All of the above are features which permit the construction of program executives varying in nature from simple, priority-driven response to scheduled periodic. Investigation showed that no single real-time interface and executive control scheme was suitable for all, or even most, applications. Instead, the SPLM contains the necessary basic elements that permit application-oriented schemes to be constructed.

Mechanization Concepts

The SPLM design is being produced by a somewhat iterative process in which each advance to a further level of detail is fully evaluated. Bindings, or mechanization decisions, are postponed as long as possible.

There is a one-to-one translation of language tokens (e.g., operators, data-references, and so forth) to machine
instructions. In other words, SPLML is the assembly language of the SPLM.

There is a general model of the machine which can be used for discussion, and which is the model upon which the language semantics are formally defined. This model corresponds to the data structures within the SPLM Semantics Simulator. Basically, the SPLM is a machine containing a sequentially stored program string, a variable stack, a single processor, and a single I/O unit.

Memory is typically the largest contributor to the size, weight and power characteristics of an aerospace computer. The portion of memory used for storing program instructions and constants is much larger than that used for storing variables (e.g., Minuteman III only 6% of memory is read/write). In order to achieve efficiency in the utilization of program memory, a minimal encoding of instructions based upon occurrence frequencies is utilized. The major factor in reducing memory, though, is that the SPLM instructions are data-independent and have a high semantic content. For example, the same ADD instruction is used to add scalars and/or arrays of any type or length. There is no need for things such as “half-word fixed add” or “double-precision floating add” in the SPLM. Figure 2 illustrates the power of this feature: the example is taken from the Minuteman III application software.

The stack is used for variable storage, formula evaluation, and block dynamics control. In the semantics model, all accesses within the stack are performed by scanning for marks. It is obvious that this would be much too slow, but “speed-up” mechanisms such as register-pointers will not be designed in until the machine characteristics can be evaluated and the true speed payoffs can be measured. Data descriptors, which contain information concerning type, length and structure, are co-located with the data value spaces for all variables. The descriptors are created when declarative statements are executed, and provide data attribute information to the processor when the variables are referenced as operands.

The I/O unit is connected with both the memory and the processor. Thus, there are two levels of interface: interleaved memory buffering on an asynchronous basis, and direct processor I/O on a synchronized basis. Except for timing, the two are semantically equivalent.

The processor scans the program string and executes all instructions. It utilizes the stack for storing of intermediate results. A basic design principle is that data values are not supplied to the processor (i.e., moved from memory) until actually needed as operands.

In mechanizing the processor, a primitive data structure was uncovered; certain basic registers that are absolutely required. Fifteen registers are necessary to either create or utilize data-descriptors; six are required to handle I/O and interrupts; and seven are necessary for general control (e.g., program scan, top-of-stack, and so forth). Since the Semantics Simulator utilizes APL operators to handle primitive data operations, the basic set of registers for arithmetic, boolean and relational functions has not been established as yet: it is anticipated that SPLM will match conventional machines in this area. Additional registers and data paths are being added during the detailed design to achieve speed balance.

It is appropriate, at this point, to discuss a basic concept behind the SPLM architecture; the design is for a class of machines, each member of which will have different timing characteristics. By judicious establishment of implementation parameters (e.g., register and data path sizes) and addition of “speed-up” hardware, an SPLM can be designed that is properly balanced for a specific application. A minimum hardware machine is the basic design goal in our present efforts: this is a design for the slowest machine that it is felt could be practically employed. Faster, and thus larger, machines are also being designed as secondary efforts for evaluation purposes.

There is one element of this concept which is of special interest. Except for a set of primitive operations, most of the SPLM monadic and dyadic operators are implementable via software. For instance, the add operation on scalars is primitive, but for arrays a trap to a software procedure can be employed. The minimum hardware machine utilizes software implementation extensively. If in a particular application, certain operators are utilized quite frequently, the speed of the SPLM can be dramatically altered by the selective addition of hardware to replace software.

This approach also leads us to view non-primitive monadic and dyadic operators as semantically equivalent to procedures. The only difference is that non-primitive operators enter software via hardware decoding and traps, while procedures are invoked by a call mechanism. There are two interesting aspects to this equivalence:

1) The set of non-primitive monadic and dyadic operators can be considered as open-ended. When designing an SPLM for a particular application, a tailored set can be chosen.

2) If unanticipated timing bottlenecks occur, the basic mechanism exists for integrating new operators. For long-term projects, the wise developer might allocate spare op-codes and micro-program storage to be utilized for speed gains tailored to the actual application software. This could also greatly increase machine life within systems whose requirements change.

Future Development

The SPLM architecture design, as well as the associated software and study documentation, is available to all. If the evaluation results bear out the intuitive estimates of its significant advantages over current aerospace computers, it is anticipated that normal competitive forces will compel manufacturers to seriously investigate SPLM as their next
A hardware breadboard is not useful at this phase of development. There is no advanced state-of-the-art hardware technology required. The ability to perform hardware-timing tradeoffs can be nicely accomplished with a good functional simulator, structured for machine parameterization.

The SPLM Clock-Level Simulator, currently in development, should prove to be extremely useful to the machine designer. It is expected to be used both as a tool for application tailoring and as a specification document for circuit design. It is implemented in APL and is intended for general usage. It is probably too detailed and slow, however, to be efficiently employed for evaluation of software samples of any appreciable size.

There are a number of study extensions such as self-test, fault-tolerance, multiprocessing, and so forth that might be pursued profitably. The manufacturing implications of the SPLM architecture seem most interesting:

1) Can hardware implementations be achieved that would permit manufacture of application-specific machines from a basic SPLM? In other words, can an off-the-shelf machine with built-in performance variability be developed?

2) Can the SPLM Clock-Level Simulator be directly interfaced with existing, or planned, computer-aided design systems?

To the authors, the most meaningful future activity will be evaluations of the SPLM as a candidate for specific applications, where the memory and timing estimates should be based upon actual coding of representative application software.

SPLM has the potential to significantly alleviate serious problems associated with aerospace computers. By efficient utilization of hardware, much more processing can be obtained for the same size, weight and power costs as today's machines. The programming language should ease the many problems associated with software development and modification.

References


Robert A. Hokom has 10 years experience in software systems, language, and computer architecture development. He was employed by the Aerospace Corporation as a technical consultant to the U.S. Air Force on the SPLM project. At present, he is a Senior Systems Planning Specialist with the Chase Manhattan Bank.

Alain Grebert, Fred Gerbstadt and Robert Bock are the principals of Systems Integration Associates, Inc., which was formed in 1970 for the purpose of pursuing the study of design methods for language oriented processors. Together they represent nearly 50 years of experience in the fields of application and systems hardware/software design and implementation with companies such as Burroughs, GE, IBM, Univac and ICT.