Symbolics Architecture

David A. Moon
Symbolics, Inc.

This architecture enables rapid development and efficient execution of large, ambitious applications. An unconventional design avoids trading off safety for speed.

What is an architecture? In computer systems, an architecture is a specification of an interface. To be dignified by the name architecture, an interface should be designed for a long lifespan and should connect system components maintained by different organizations. Often an architecture is part of a product definition and defines characteristics on which purchasers of that product rely, but this is not true of everything that is called an architecture. An architecture is more formal than an internal interface between closely-related system components, and has farther-reaching effects on system characteristics and performance.

A computer system typically contains many levels and types of architecture. This article discusses three architectures defined in Symbolics computers:

(1) System architecture—defines how the system appears to end users and application programmers, including the characteristics of languages, user interface, and operating system.

(2) Instruction architecture—defines the instruction set of the machine, the types of data that can be manipulated by those instructions, and the environment in which the instructions operate, for example subroutine calling discipline, virtual memory management, interrupts and exception traps, etc. This is an interface between the compilers and the hardware.

(3) Processor architecture—defines the overall structure of the implementation of the instruction architecture. This is an interface between the firmware and the hardware, and is also an interface between the parts of the processor hardware.

System architecture

System architecture defines how the system looks to the end user and to the programmer, including the characteristics of languages, user interface, and operating system. System architecture defines the product that people actually use; the other levels of architecture define the mechanism underneath that implements it. System architecture is implemented by software; hardware only sets bounds on what is possible. System architecture defines the motivation for most of the design choices at the other levels of architecture. This section is an overview of Symbolics system architecture.

The Symbolics system presents itself to the user through a high-resolution bitmap display. In addition to text and graphics, the display contains presentations of objects. The user operates on the objects by manipulating the presentations with a mouse. The display includes a continuously updated reminder of the mouse commands applicable to the current context. Behind the display is a powerful symbol processor with specialized hardware and software. The system is dedicated to one user at a time and shares such resources as files, printers, and electronic mail with other Symbolics and non-Symbolics computers through both local-area and long-distance networks of several types. The local-area network is integral to system operation.

The system is designed for high-productivity software development both in symbolic languages, such as Common Lisp[1] and Prolog, and in nonsymbolic languages, such as Ada and Fortran. It is also designed for efficient execution of large programs, particularly in symbolic languages, and delivery of such programs to end users. The system is intended to be especially suited to complex, ambitious applications that go beyond what has been done before; thus it provides facilities for exploratory programming, complexity management, incremental construction of programs, and so forth. The operating system is written in Lisp and the architec-

0018-9162/87/0100-0043$01.00 © 1987 IEEE

tural concept originated at the MIT Artificial Intelligence Laboratory. However, applications are not limited to Lisp and AI. Many non-AI applications that are complex enough to be difficult on an ordinary computer have been successfully implemented.

Meeting these needs requires an extraordinary system architecture—just another PC or Unix clone won't do. The intended applications demand a lot of processor power, main and virtual memory size, and disk capacity. The system must provide as much performance as possible without exceeding practical limits on cost, and computing capacity must not be diluted by sharing it among multiple users. These purely hardware aspects are not sufficient, however. The system must also improve both the speed of software production and the quality of the resulting software by providing a more complete substrate on which to erect programs than has been customary. Programmers should not be handed just a language and an operating system and be forced to do everything else themselves.

At a high level, the Symbolics substrate provides many facilities that can be incorporated into user programs, such as user-interface management, embedded languages, object-oriented programming, and networking. At a low level, the substrate provides full run-time checking of data types, of array subscript bounds, of the number of arguments passed to a function, and of undefined functions and variables. Programs can be incrementally modified, even while they are running, and information needed for debugging is not lost by compilation. Thus the edit-compile-test program development cycle can be repeated very rapidly. Storage management, including reclamation of space occupied by objects that are no longer in use, is automatic so that the programmer does not have to worry about it; incremental so that it interferes minimally with response to the user; and efficient because it concentrates on ephemeral objects, which are the best candidates for reclamation. The system never compromises safety for the sake of speed. (A notorious exception, the dynamic rather than indefinite extent of &rest arguments, is recognized as a holdover from the past that is not consistent with the system architecture and will certainly be fixed in the future.)

In an ordinary architecture, such features would substantially diminish performance, requiring the introduction of switches to turn off the features and regain speed. Our system architecture deems this unacceptable, because complex, ambitious application programs are typically never finished to the point where it is safe to declare them bug-free and remove run-time error-checking. We feel it is essential for such applications to be robust when delivered to end users, so that when something unanticipated by the programmer happens, the application will fail in an obvious, comprehensible, and controlled way, rather than just supplying the wrong answer. To support such applications, a system must provide speed and safety at the same time.

Symbolics systems use a combination of approaches to break the traditional dilemma in which a programmer must choose either speed or safety and comfortable software development:

- The hardware performs low-level checking in parallel with computation and memory access, so that this checking takes no extra time.
- Machine instructions are generic. For example, the Add instruction is capable of adding any two numbers regardless of their data types. Programs need not know ahead of time what type of numbers they will be adding, and they need no declarations to achieve efficiency when using only the fastest types of numbers. Automatic conversion between data types occurs when the two operands of Add are not of the same type.
- Function calling is very fast, yet does not lose information needed for debugging and does not prevent functions from being redefined.
- Built-in substrate facilities are already optimized and available for programmers to incorporate into their programs.
- Application-specific control of virtual-memory paging is possible. Prepaging, postpurging, multipage transfers, and reordering of objects to improve locality are supported.[2]

These benefits are not without costs:

- Both the cost and the complexity of system hardware and software are increased by these additional facilities.
- Performance optimization is not always automatic. Programmers still must sometimes resort to metering tools. Declarations are available to optimize certain difficult cases, but their use is much less frequent than in conventional architectures.

Why Lisp machines? This is really three questions:

(1) Why dedicate a computer to each user instead of time-sharing?

(2) Why use a symbolic system architecture?

(3) Why build a symbolic system architecture on unconventional lower-level architectures?

Why dedicate a computer to each user instead of time-sharing? This seemed like a big issue back in 1974 when Lisp machines were invented, but perhaps by now the battle has been won. A report from that era[3] states these reasons for abandoning time-sharing:

- Time-sharing systems degrade under heavy load, so work on large, ambitious programs could only be conducted in off-peak hours. In contrast, a single-user system would perform consistently at any time of day.
- Performance was limited by the speed of the disk when running programs too large to fit in main memory. Dedicating a disk to each user would give better performance.

The underlying argument was that increasing program size and advancing technology, making capable processors much less expensive, had eliminated the economy of scale of time-sharing systems. The original purpose of time-sharing was to share expensive hardware that was only lightly used by any individual user. The serendipitous feature of time-sharing was interuser communication. Both of these purposes are now served by local-area networking. Expensive hardware units are still shared, but the processor is no longer among them.

These arguments apply to all types of dedicated single-user computers, even PCs, not only to symbolic architectures.

Why use a symbolic system architecture? Many users who need a platform for efficient execution of large symbolic programs, a high-productivity software development environment, or a system for exploratory programming and rapid prototyping have found symbolic languages such as Lisp and symbolic architectures such as this one very beneficial. Programs can be built more quickly, and fit more smoothly into an integrated environment, by incorporating such built-in substrate facilities as automatic storage management and the flexible display with its presentation-based user interface. The full error-checking saves time when developing new programs. The programmer can concentrate on the essential aspects of the program without fussing about minor

mistakes, because the machine will catch them. The ability to change the program incrementally greatly speeds up development.

Once the initial exploration phase is over, it is possible to turn prototypes into products quickly. Good performance can be achieved without a lot of programmer effort and without sacrificing those development-oriented features that are also of value later in the program's life, during maintenance and enhancement.

Why build a symbolic system architecture on unconventional lower-level architectures? Conventional instruction architectures are optimized to implement system architectures very different from Symbolics'. For example, they have no notions of parallel error-checking and generic instructions; they often obstruct the implementation of a fast function call, especially one that retains error-checking, incremental compilation, and debugging information; and they usually pay great attention to complex indexing and memory addressing modes, which have little utility for symbolic languages. Implementing Symbolics' system architecture on a conventional instruction architecture would force a choice between safety and performance: we could not have both. The type of software we are interested in either could not run at all or would require much faster hardware to achieve the same performance. Later I will discuss the special aspects of Symbolics' instruction and processor architectures that make them more suitable to support a symbolic system.

Comparing the performance of machines with equivalent cycle times and different architectures can sometimes be illuminating. The 3640, VAX 11/780, and 10-MHz 68020 all have cycle times of about 200 ns. (The 68020 takes two clock cycles to perform a basic operation, so its 100-ns nominal cycle time is equivalent to the other two machines' 200 ns.) On a Fortran benchmark (single-precision Whetstone), the VAX is 1.8 times the speed of the 3640 (750 versus 400). With floating-point accelerators on each machine, the ratio is 2.1. On the Lisp benchmark Boyer,[4] the 3640 is 1.75 times the speed of the VAX running Portable Standard Lisp, 3.9 times the speed of the VAX running DEC Common Lisp, and 2.1 times the speed of the 68020 running Lucid Common Lisp. (The 68020 time at 10 MHz was estimated by multiplying its 16-MHz time by 1.6, no doubt an inaccurate procedure.) The VAX and 68020 programs were compiled with run-time error checking disabled and safety compromised, while the 3640 was doing full checking as always. Like any benchmark figures presented without a complete explanation of what was measured, how it was measured, what full range of cases was tested, and how it can be reproduced in another laboratory, these numbers should not be taken very seriously. However, they give some idea of the effect of optimizing the instruction architecture to fit the system architecture. One could say that the VAX is three times better at Fortran than at Lisp and that the 68020 and VAX are similar for Lisp. These figures also show the effect of different compiler strategies on identical hardware.

This comparison was scaled to remove the effect of cycle time and show only the effect of architecture. This is not completely fair to the conventional machines, because in general they can be expected to have faster cycle times than a symbolic machine. Running the 68020 at full speed and using a newer model of the VAX would have improved their times. Hardware technology of conventional machines will always be a couple of years ahead of symbolic hardware, in cycle time and price/cycle, because of the driving force of their larger market. It's interesting to note that this hardware advantage applies only to the processor, which usually contributes less than 25 percent of system cost. Power supplies, sheet metal, and disk drives don't care whether the architecture is symbolic; they cost and perform the same for equivalent configurations of either type of machine.

This comparison is not completely fair to the symbolic machine, either. Software exploiting the full capabilities of the symbolic machine should have been compared, but this software won't run at all on the conventional machines. Software technology on symbolic machines will always be a couple of years ahead of conventional machines, because it is built on a more powerful substrate using more productive tools.

Performance. The best published analysis of performance of Lisp systems appears in Gabriel's work.[4] The various 3600 models perform quite capably on these benchmarks, as can be seen from a perusal of the book. Some of the reasons for such good performance will become apparent as we proceed.

However, one must always ask exactly what a benchmark measures. A problem with Gabriel's benchmarks is that they are written in a least common denominator dialect that represents Lisp as it was in 1970. This makes it easier to benchmark a broad spectrum of machines, but makes the benchmarks less valid predictors of the performance of real-world programs. Since 1970, there have been many advances in the understanding of symbolic processing and in the range of its applications. The basic operations measured by these benchmarks, such as function calling, small-integer arithmetic, and list processing, are still important today, but many other operations not measured are of equal importance. These benchmarks do not use the more modern features of Common Lisp (such as structures, sequences, and multiple values), do not use object-oriented programming, and are generally not affected by system-wide facilities such as paging and garbage collection. As predefined, portable programs, these benchmarks cannot benefit from the unusual aspects of Symbolics system architecture, such as large program support, full run-time safety, efficient storage management, substrate facilities, support for languages other than Lisp, and faster development of efficient programs.

Instruction architecture

Symbolics' philosophy is that different levels of architecture should be free to change independently, to satisfy different goals and constraints. Users see only the system architecture, leaving the lower levels, such as the instruction architecture, free to change to utilize available technology, maximize performance, or minimize cost. Most other computer families allow users to depend on the instruction architecture and therefore are not free to change it. It tends to be optimized for only the first member of the family. Later implementations using newer technology, as well as implementations at the high or low extremes of the price/performance curve, are penalized by the need for compatibility with an unsuitable instruction architecture.

Symbolics system architecture has been implemented on three different instruction architectures. The LM-2 machine, based on the original MIT Lisp Machine,[3] was the first; it was discontinued in 1983. The 3600 family of machines uses a second instruction architecture and three different processor architectures. A third instruction architecture, appropriate for VLSI implementation, is being used in a line of future products now under development.

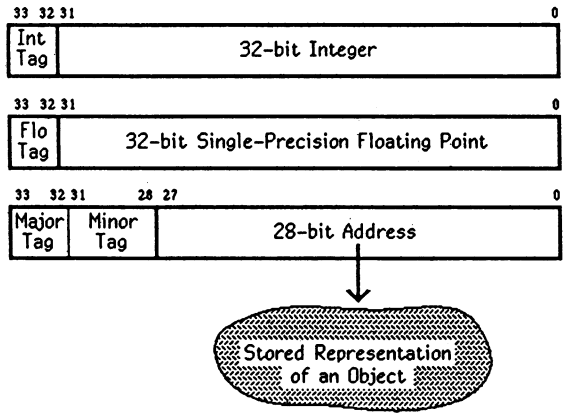

Figure 1. An object reference is a 34-bit quantity, consisting either of a 32-bit data word with a 2-bit data type tag, or of a 28-bit address with a 6-bit data type tag.

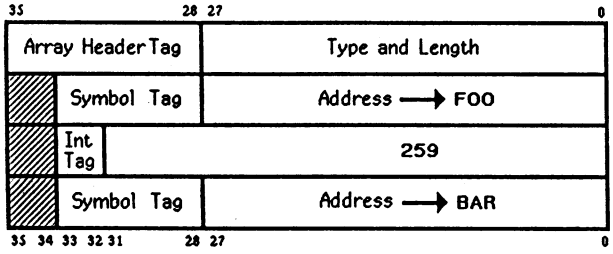

Figure 2. An array of three elements—FOO, 259, and BAR—consists of a header word defining the type and length of the array, followed by an object reference for each array element.

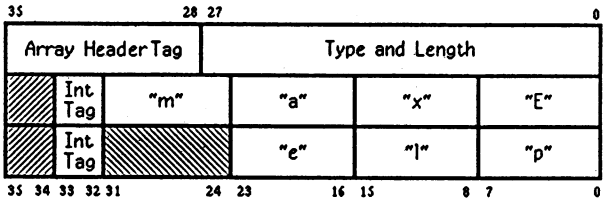

Figure 3. A string containing the seven characters "Example" stores each character in a single 8-bit byte. Bytes are packed into 32-bit integer objects.

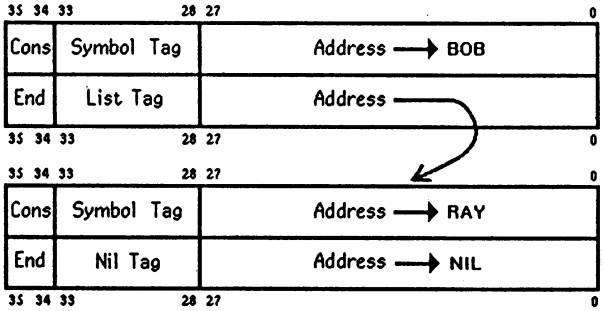

Figure 4. An ordinary list of two elements requires four words of storage. Unlike arrays, lists do not have headers.

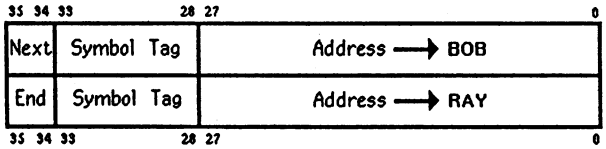

Figure 5. A compact list of two elements requires two words of storage. It uses the cdr code to eliminate two object references.

(Reprinted from "Architecture of the Symbolics 3600," *12th Int'l Symp. Computer Architecture*, © 1985 IEEE.)

The following sections summarize the instruction and processor architectures of the 3600 family, discuss some of the design tradeoffs involved, and show how these architectures are especially effective at supporting the desired system architecture. Further details can be found elsewhere.[5,6]

Data are object references. The fundamental form of data manipulated by any Lisp system is an *object reference*, which designates a conceptual object. The values of variables, the arguments to functions, the results of functions, and the elements of lists are all object references. There can be more than one reference to a given object. Copying an object reference makes a new reference to the same object; it does not make a copy of the object.

Variables in Lisp and variables in conventional languages are fundamentally different. In Lisp, the value of a variable is an object reference, which can refer to an object of any type. Variables do not intrinsically have types; the type of the object is encoded in the object reference. In a conventional language, assigning the value of one variable to another copies the object, possibly converts its type, and loses its identity.

A typical object reference contains the address of the object's representation in storage. There can be several object references to a particular object, but it has only a single stored representation. Side-effects to an object, such as changing the contents of one element of an array, are implemented by modifying the stored representation. All object references address the same stored representation, so they all see the side-effect.

In addition to such object references by address, it is possible to have an immediate object reference, which directly contains the entire representation of the object. The advantage is that no memory needs to be allocated when creating such an object. The disadvantage is that copying an immediate object reference effectively copies the object. Thus, immediate object references can only be used for object types that are not subject to meaningful side-effects, have a small representation, and need very efficient allocation of new objects. Small integers (traditionally called fixnums) and single-precision floating-point numbers are examples of such types.

In the 3600 architecture, an object reference is a 34-bit quantity consisting of a 32-bit data word and a 2-bit major data type tag. The tag determines the interpretation of the data word. Often the data word is broken down into a 4-bit minor data type tag and a 28-bit address (see Figure 1). This variable-length tagging scheme accommodates industry-standard 32-bit fixed and floating-point numbers with a minimum of overhead bits for tagging. Addresses are narrower than numbers to make additional tag bits available for the many types of objects that Lisp uses.

Addresses are 28 bits wide and designate 36-bit words in a virtual memory with 256-word pages. The address granularity is a word, rather than a byte as in many other machines, because the architecture is object-oriented and objects are always aligned on a word boundary. This results in one gigabyte of usable virtual memory. It is interesting to note that the 3600's 28-bit address can actually access the same number of usable words as the VAX's 32-bit address, because the VAX expends two bits on byte addressing and reserves three-fourths of the remaining address space for the operating system kernel and the stack (neither of which is large).

In addition to immediate and by-address object references, the 3600 also uses pointers, a special kind of object reference that does not designate an object as such. A pointer designates a particular location within an object or a particular instruction within a compiled function. Pointers are used primarily for system programming.[7]

Stored representations of objects. The stored representation of an object is contained in some number of consecutive words of memory. Each word may contain an object reference, a header, a special marker, or a forwarding pointer. The data type tags distinguish these types of words. For example, an array is represented as a header word, containing such information as the length of the array, followed by one memory word for each element of the array, containing an object reference to the contents of that element (see Figure 2). An object reference to the array contains the address of the first memory word in the stored representation of the array.

A *header* is the first word in the stored representation of most objects. A header marks the boundary between the stored representations of two objects. It contains descriptive information about the object that it heads, which can be expressed as either immediate data or an address, as in an object reference.

A *special marker* indicates that the memory location containing it does not currently contain an object reference. Any attempt to read that location signals an error. The address field of a special marker specifies what kind of error should be signalled. For example, the value cell of an uninitialized variable contains a special marker that addresses the name of the variable. An attempt to use the value of a variable that has no value provokes an error message that includes the variable's name.

A *forwarding pointer* specifies that any reference to the location containing it should be redirected to another memory location, just as in postal forwarding. These are used for a number of internal bookkeeping purposes by the storage management software, including the implementation of extensible arrays.

Some objects include packed data in their stored representation. For example, character strings store each character in a single 8-bit byte (see Figure 3). For uniformity, the stored representation of an object containing packed data remains a sequence of object references. Each word is an immediate object reference to an integer, whose 32 bits are broken down into packed fields as required, such as four 8-bit bytes in the case of a character string.

A word in memory consists of 36 bits, of which I have already explained 34. When a memory word contains a header or a machine instruction, the remaining two bits serve as an extension of the rest of the word. When a memory word contains an object reference, a special marker, or a forwarding pointer, the remaining two bits are called the cdr code. The representation of conses and lists (Steele, p. 26)[1] saves one word by using the cdr code instead of a separate header to delimit the boundaries of these small objects. In addition, lists are represented compactly by encoding common values of the cdr in the cdr code instead of using an object reference (see Figures 4 and 5).

Tagging every word in memory produces these benefits:

- All data are self-describing and the information needed for full run-time checking of data types, array subscript bounds, and undefined functions and variables is always available.
- Hardware can process the tag in parallel with other hardware that processes the rest of a word. This makes it possible to optimize safety and speed simultaneously.
- Generic instructions alter their operation according to the tags of their operands.
- Automatic storage management is simple, efficient, and reliable. It can be assisted by hardware, since the data structures it deals with are simple and independent of context. The details appear elsewhere.[8,5]
- Data use less storage due to compact representations. Programs use less storage due to generic instructions and because tag checking is done in hardware, not software.

The cost of tagging is that more main memory and disk space are required to store numerical information. Each main memory word includes 7 bits for error detection and correction, so the 4 tag bits add 10 percent. Each 256-word disk sector includes about 128 bytes of formatting overhead, so the 4 tag bits per word add 11 percent. We feel that the benefits amply justify these costs.

Instruction set. The 3600 architecture includes an instruction set produced by the compilers and executed by a combination of hardware and firmware. All instructions are 17 bits long, consisting of a 9-bit operation field and an 8-bit argument field. Instructions are packed two per word, which is important for performance in two ways:

(1) Dense code decreases paging overhead by making programs occupy fewer pages and

(2) simplifies the memory system by decreasing the ratio of required instruction fetch bandwidth (in words/second) to processor speed (in instructions/second).

Every instruction is contained in a compiled function, which consists of some fixed overhead, a table of constants, and a sequence of instructions (see Figure 6). The table of constants contains object references to objects used by the instructions, including locative pointers to definition cells of functions called by this function. Indirection through the definition cell ensures that if a function is redefined its callers are automatically linked to the new definition.

Instructions operate in a stack machine model: Many instructions pop their operands off the stack and push their results onto the stack. In addition to these 0-address instructions, there are 1-address instructions, which can address any location in the current stack frame. In this way the slots of the current stack frame serve the same purpose as registers. The

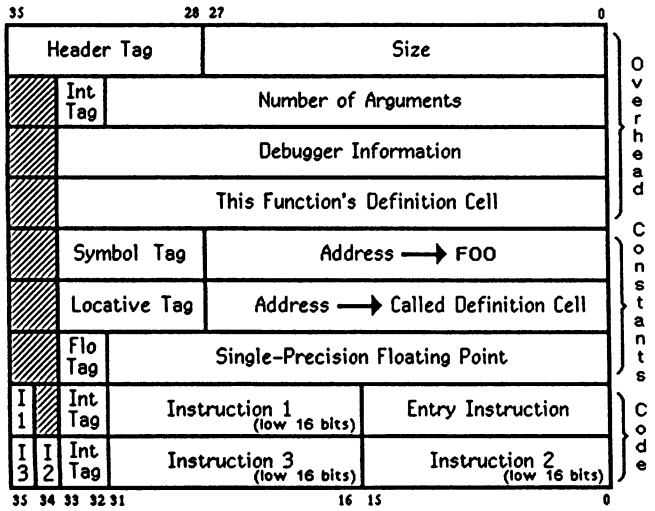

Figure 6. A compiled function consists of four words of overhead, a table of constants and external references, and a sequence of 17-bit instructions, packed two per word.

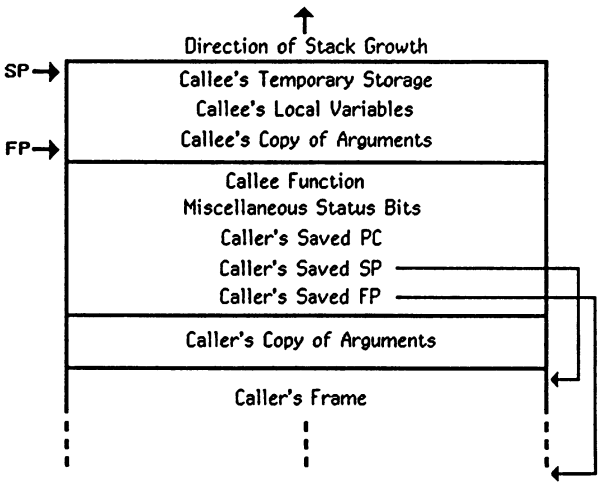

Figure 7. A stack frame consists of the caller's copy of the arguments, five header words, the callee's copy of the arguments, local variables, and temporary storage. The frame-pointer (FP) and stack-pointer (SP) registers address the current stack frame.

(Reprinted from "Architecture of the Symbolics 3600," *12th Int'l Symp. Computer Architecture*, © 1985 IEEE.)

1-address instructions include multi-operand instructions, which pop all of their operands except the last off the stack and take their last operand from a location in the current stack frame.

There are several ways an instruction can use its argument field. Table 1 lists the ways to develop the address of an operand in the stack or in memory by adding argument to a base address. Table 2 lists non-address uses of argument. Each individual opcode only uses argument in a single way; there are no addressing modes. The motivation for implementing this particular set of arguments is to provide for constants (including small integers as a special case), all types of Lisp variables (local and nonlocal lexical, special, structure slot, instance), branching, and byte fields. Byte fields were included because they are heavily used in system programming.

Many instructions are simply Lisp functions directly implemented by hardware and firmware, rather than built up from other Lisp functions and implemented as compiled instructions. These Lisp-function instructions are known as built-ins. They take a fixed number of arguments from the stack and from their argument field. They return a fixed number of values on the stack. Examples of built-ins are eq, symbolp, logand (with two arguments), car, cons, member, and aref (with two arguments).[1] The criterion for implementing a Lisp function as a built-in instruction is that hardware is only used to optimize key performance areas. When a Lisp function is not critical to system performance, or hardware implementation of it cannot achieve a major speedup, it remains in software where it is easier to change, to debug, and to optimize.

Using an instruction set designed for Lisp rather than adapting one designed for Fortran or for a hand-crafted assembly language enhances safety and speed. 3600 instructions always check for errors and exceptions, so programs need not execute extra instructions to do that checking. Instructions operate on tagged data, so extra instructions to insert and remove tags are not needed. Instructions are generic, so declarations are not needed to tell the compiler how to select type-specific instructions and translate between data formats. In contrast, Lisp compilers for conventional machines[9] must generate extra shifting or masking instructions to manipulate tags, must use multi-instruction sequences for simple arithmetic operations unless there are declarations, and are always having to compromise between safety and speed.

Unlike many machines, the 3600 does not have indexed and indirect addressing modes. Instead it has instructions that perform structured, object-oriented operations such as subscripting an array or fetching the car of a list. This fits the instruction set more closely to the needs of Lisp and at the same time simplifies the hardware by reducing the number of instruction formats to be decoded.

Function call. Storage whose lifetime is known to end when a function returns (or is exited abnormally) is allocated in three stacks, rather than in the main object storage heap, to increase efficiency. The control stack contains function-nesting information, arguments, local variables, function return values, and small stack-allocated temporary objects. The binding stack records dynamically bound variables.[1] The data stack contains stack-allocated temporary objects. This article concentrates on the control stack, which is the most critical to performance.

The protocol for calling a function is to push the arguments onto the stack, then execute a Call instruction that specifies the function to be called, the number of arguments, and what to do with the values returned by the function. When the function returns, the arguments have been popped off the stack and the values (if wanted) have been pushed on. Note the similarity in interface between functions and built-in instructions.

Every time a function is called, a new stack frame is built on the control stack. A stack frame consists of the caller's copy of the arguments, five header words, the callee's copy of the arguments, local variables, and temporary storage, including

arguments being prepared for calling the next function (see Figure 7). The current stack frame is delimited by the frame-pointer (FP) and stack-pointer (SP) registers, which are available as base registers in instructions that use their argument field to address locations in the current stack frame.

A compiled function starts with a sequence of one or more instructions known as the entry vector. The first instruction in the entry vector, the entry instruction, describes how many arguments the function accepts, the layout of the entry vector, and the size of the function's constants table (see Figure 6), and tells the Call instruction where in the entry vector to transfer control. The Call instruction and the entry vector cooperate to copy the arguments to the top of the stack (creating the callee's copy), convert their arrangement in storage if required, supply default values for optional arguments that the caller does not pass, handle the &rest and Apply features of Common Lisp, and signal an error if too many or too few arguments were supplied. The details are beyond the scope of this article.

Function return. A function returns by executing a Return instruction whose operands are the values to be returned. The value disposition saved in the frame header by Call controls whether Return discards the values, returns one value on the stack, returns multiple values with a count on the stack, or returns all the values to the caller's caller.

Return removes the current frame from the stack and makes the caller's frame current, by restoring the saved FP, SP, and PC registers. If the cleanup bits in the frame header are nonzero, special action must be taken before the frame can be removed. Return takes this action, clears the bit, and tries again. Cleanup bits are used to pop corresponding frames from the binding and data stacks, for unwind-protect,[1] for debugging and metering purposes, and for stack buffer housekeeping.

Motivations of the function call discipline. The motivations for this particular function-calling discipline are

- to implement full Common Lisp function calling efficiently,
- to be fast, so that programmers will write clear programs,
- to retain complete information for the Debugger, and
- to be simple for the compiler.

Table 1. Ways to develop an operand address.

Beginning of the current stack frame + offset
End of the current stack frame - offset
Lexical parent's environment captured in a closure + offset
Current function's constants table + offset (possibly with indirection)
An array used as a DEFSTRUCT structure + subscript
A flavor instance + offset (possibly with mapping-table indirection)
Address of the current instruction ± offset (for branching)

Table 2. Nonaddress uses of argument field.

A byte field any number of bits wide, positioned anywhere within a 32-bit word. This specification takes ten bits and hence overflows into two bits of operation.
An immediate integer between -128 and 127
An immediate integer between 0 and 255
Extended operation field, to allow more than 512 instructions

To implement full Common Lisp function calling efficiently requires matching the arguments supplied by the caller—with normal function calling or with Apply—to the normal, &optional, and &rest parameters of the callee, and generating default values for unsupplied optional arguments. The entry vector takes care of this. Common Lisp's &key parameters are implemented by accepting an &rest parameter containing the keywords and values, then searching that list for each &key parameter. Multiple values are passed back to the caller on the stack, with a count. The caller reconciles the number of values returned with the number of values desired.

Function calling historically has been a major bottleneck in Lisp implementations, both on stock hardware and on specially-designed Lisp machines. It is important for function calling to be as fast as possible. If it is not, efficiency-minded programmers will distort their programming styles to avoid function calling, producing code that is hard to maintain, and will waste a lot of time doing optimization by hand that should have been done by the Lisp implementation itself. The 3600's function call mechanism attains good speed (fewer than 20 clock cycles for a one-argument function call and return when no exceptions occur) by using a stack buffer to minimize the number of memory references required, by optimizing the stack frame layout to maximize speed rather than to minimize space, by arranging for the checks for slower exception cases to be fast (for example, Return simply checks whether the cleanup bits are nonzero), and by using the entry vector mechanism to simplify run-time decision-making.

The information that the debugger can extract from a stack frame includes the address of the previous frame (from the saved FP in the header), the function running in that frame (from the header), the current instruction in that function (from the PC saved in the next frame), the arguments (from the stack—the header specifies the argument count and arrangement), the local variables (from the stack), and the names of the arguments and local variables (from a table created by the compiler and attached to the function).

The compiler is simple because there is only a single calling sequence. Any call can call any function, and the argument patterns are matched up at run time. Everything is in the stack and no register-saving conventions are required, since there are no general-purpose registers.

The principal costs of this function-calling discipline are the five-word header in each frame and the copying of arguments to the top of the stack. The time to create the header is not a problem, because it is overlapped with necessary memory accesses, but the space occupied by the header and by the extra copy of the arguments is a substantial fraction of the typical frame size. This extra space is not a major problem because the stack buffer is large enough (1024 words) that it rarely overflows.

Argument copying is necessary because Common Lisp functions do not take a fixed number of arguments. In a function with &optional parameters, some of the arguments are supplied by the caller while the others are defaulted by the entry vector. The location in the stack frame of an argument must not depend on whether it was supplied or defaulted, since this varies from one call to the next, but the compiler must know the location in order to generate code to access the argument. The entry vector could not put default values in the standard location if the arguments were not at the top of the stack, because the frame header would be in the way. In a function with an &rest parameter, the caller can supply an arbitrary number of arguments. If these arguments were at the top of the stack, they would make it impossible for the compiler to know the locations of the local variables, which are pushed after the arguments.

Copying the arguments that are not part of an &rest parameter to the top of the stack solves both these problems. It gives the function complete control over the arrangement of its stack frame and makes the stack depth constant. Argument copying takes extra time, but typically only one clock cycle per argument, which is faster than the run-time decision-making that would otherwise be necessary to access an optional argument or a local variable.

Processor architecture

Three processor architectures are used in three representative models of the 3600 family: 3640, 3675, and 3620. Since they all implement the same instruction architecture, there are substantial similarities among their processor architectures. They differ due to implementation in different technologies and choices of different cost/performance tradeoffs, but this overview largely glosses over the differences.

The main goal of each of these processor architectures is to implement the instruction architecture described earlier with the highest performance achievable within its particular cost budget. The costs are generally higher than most workstations but lower than most minicomputers. For high performance the number of clock cycles required to execute an instruction must be minimized; the goal is to execute a new instruction every cycle. Because the system architecture specifies that safety and convenience must not be compromised to increase performance, instructions typically make many checks for errors and exceptions. Minimizing the cycle count demands that these checks be performed in parallel, not each in a separate cycle.

Adequate bandwidth for access to operands is also required. In the 3600 instruction architecture, a simple instruction can read two stack locations and write one stack location. One of these is a location in the current stack frame specified by an address in the instruction, while the other two are at the top of the stack. Operands are supplied by the stack buffer, a 1K-word memory that holds up to four virtual-memory pages of the stack. The stack buffer contains all of the current frame plus as many older frames as happen to fit. When the stack buffer fills up (during Call), the oldest page spills into normal memory to make room for the new frame. When the stack buffer becomes empty (during Return), pages move from normal memory back into the stack buffer until the frame being returned to is entirely in the buffer. The maximum size of a stack frame is limited to what will fit in the stack buffer. A second stack buffer contains an auxiliary stack for servicing page faults and interrupts without disturbing the primary buffer.

Associated with the stack buffer are the FP and SP registers, which point to the current frame and to the top of the stack, and hardware for addressing locations in the current stack frame via the argument field of an instruction, which calculates a read address and a write address every clock cycle. The third operand access is provided by a duplicate copy of the top location in the stack, in a scratchpad memory, which can be read and written every clock cycle. The SP register is incremented or decremented by instructions that push or pop the stack.

The stack buffer provides the same operand bandwidth, two reads and one write every clock cycle, as in a typical register-oriented architecture. It has the advantage that register saving and restoring across subroutine calls is not required, since all registers already reside in the stack. As in a register-window design, overhead occurs only when the stack buffer overflows or underflows and requires a block transfer between stack buffer and main memory. Another advantage is that each instruction contains only one address instead of three, making the instructions smaller (so that they can be fetched from main memory more quickly and processed with less hardware) and allowing more registers to be addressed. A disadvantage of a stack architecture is that it requires address-calculation hardware, including a 10-bit (for a 1K-word buffer) adder. Since each instruction contains only one address instead of three, extra instructions are sometimes required to move data to the top of the stack so they can be addressed.

Instructions are processed by a four-stage pipeline (see Figure 8) under the control of horizontal microcode. Microcode is used as an engineering technique, not to create a general-purpose emulator that could implement alternate instruction architectures. Knowledge of the instruction architecture is built into hardware wherever that achieves a substantial performance improvement.

To achieve full performance, instructions must be supplied to the processor at an adequate rate. Each processor model has a different design, with different tradeoffs.

The 3640 uses a four-instruction buffer. When the buffer is exhausted, or a branch occurs, microcode reads two words from memory and refills the instruction buffer. This design uses much less hardware than the other two, but provides lower performance. Refilling the buffer takes five clock cycles, so in the worst case the performance penalty is about a factor of two. With a typical instruction mix, the observed slowdown is about 35 percent, because complex instructions such as function calls and memory references spend more than one cycle in the execute stage.

The 3675 uses a 2K-instruction cache. Program loops that fit in the cache execute at full speed, with no instruction fetching overhead. An autonomous instruction prefetch unit fills the cache with instructions before they are needed, in parallel with execution. At the cost of a substantial increase in hardware complexity over the 3640, this design ensures that the pipeline almost never has to wait for an instruction.

The 3620 uses a six-instruction buffer. An autonomous instruction prefetch unit fills the buffer in parallel with execution. The 3620 instruction stage is a compromise between the other two designs. Straight-line code executes at full speed, but branches execute at 3640 speed because they must refill the buffer.

The datapath contains several units that function in parallel (see Figure 9). Simple instructions such as data movement, arithmetic, logical, and byte-field instructions execute in a single clock cycle. For example, when executing an Add instruction

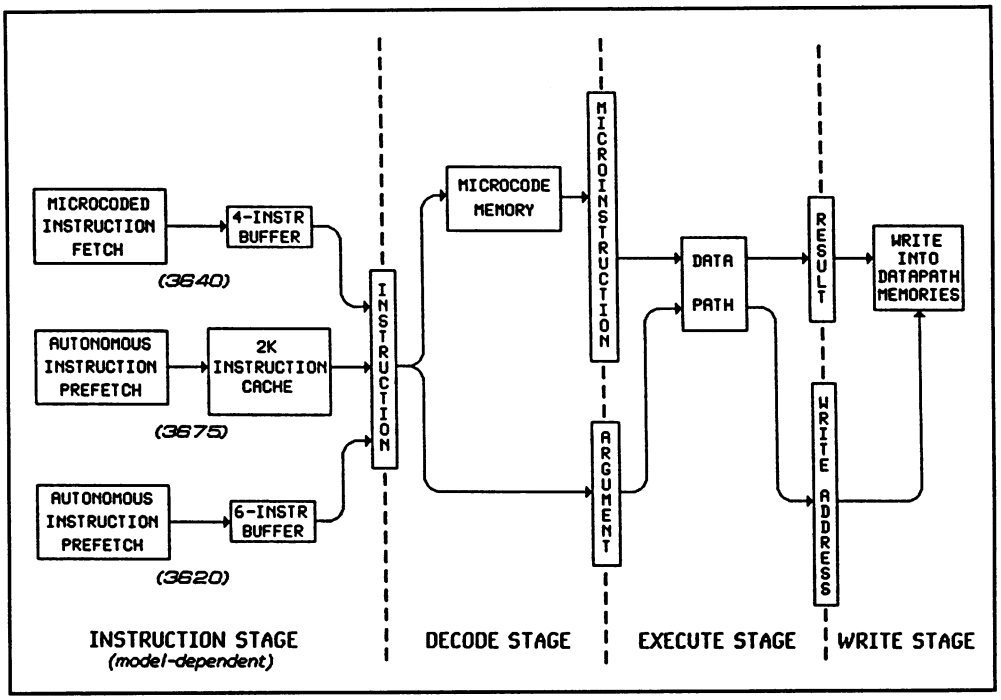

Figure 8. The instruction processing pipeline, with variations for three 3600 family models.

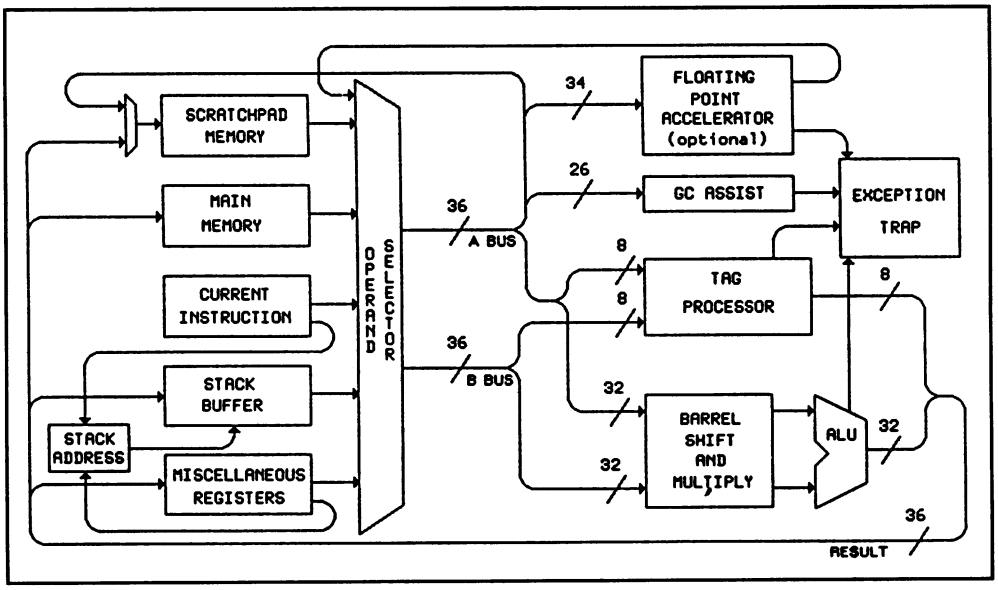

Figure 9. 3640 datapath, contained in the Execute and Write stages of the pipeline. Other 3600 family models have generally similar datapaths.

the following activities all take place in parallel:

- The stack buffer fetches the two operands, one from a calculated address in the stack buffer memory and the other from the duplicate top-of-stack in the scratchpad memory.
- The fixed-point arithmetic unit computes the 32-bit sum of the operands and checks for overflow. This result is only used if both operands are fixnums.
- The optional floating-point accelerator, if present, starts computing the sum of the operands and checking for floating-point exceptions. This result is only used if both operands are single-floats.
- The tag processor checks the data types of the operands.
- The stack buffer accepts the result from the fixed-point arithmetic unit, adjusts the stack pointer, and in the write stage stores the result at the new top of the stack.
- The decode stage decodes the next instruction and produces the microinstruction that will control its execution. If the type-checking unit or either arithmetic unit detects an exception, control is diverted to a microcode exception handler.

When the operands of Add are not both fixnums, executing the instruction takes more than one machine cycle and more than one microinstruction. In the case of adding two single-floats, the extra time is only required because the floating-point arithmetic unit is slower than the fixed-point arithmetic unit. In other cases, extra time is required to convert the operands to a common format, to perform double-precision floating-point operations, or to trap to a Lisp function to add numbers of less common types.

Memory-reference instructions such as the car and aref Lisp operations are limited

mainly by the speed of the memory. Car, for example, takes four clock cycles. Complex instructions such as Call, Return, and the Common Lisp member function invoke microcode subroutines. A wide microinstruction word and fast microcode branching minimize the number of microinstructions that need to be executed. Simple and memory-reference instructions can be discovered to be complex at run time because of an exceptional condition such as the data type of the operands.

I have described here an unusual system architecture and presented an overview of the underlying architectures that implement it. When considering the type of applications that this system architecture targets, note how important to their success it is that we compromise neither safety nor speed. With this in mind, some of the unconventional design choices in these architectures were made based on rationales with varied benefits and costs. For example, a close fit between processor, instruction, and system architectures improves performance, but allowing users to depend on details of the instruction architecture can interfere with this. The lack of this close fit dissipates the hardware price/performance advantage of conventional architectures when measuring system-level performance on software suited to symbolic architectures. □

References

1. G. L. Steele, *Common Lisp*, Digital Press, Burlington, MA, 1984.
2. D. L. Andre, *Paging in Lisp Programs*, Master's thesis, University of Maryland, 1986.
3. R. D. Greenblatt et al., "The LISP Machine," *Interactive Programming Environments*, eds. D. R. Barstow, H. E. Shrobe, and E. Sandewall, McGraw-Hill, Hightstown, NJ, 1984.
4. R. P. Gabriel, *Performance and Evaluation of Lisp Systems*, The MIT Press, Cambridge, MA, 1985.
5. D. A. Moon, "Architecture of the Symbolics 3600," *12th Int'l Symp. Computer Architecture*, 1985, pp. 76-83.
6. *Symbolics Technical Summary*, Symbolics Inc, Cambridge, MA, 1985.
7. *Symbolics Common Lisp: Language Dictionary*, Symbolics Inc, Cambridge, MA, 1986.
8. D. A. Moon, "Garbage Collection in a Large Lisp System," *Proc. 1984 ACM Symp. Lisp and Functional Programming*, pp. 235-246.
9. R. A. Brooks et al., "Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp," *Proc. 1986 ACM Conf. Lisp and Functional Programming*, pp. 67-85.

David A. Moon is a technical director at Symbolics, Inc. Previously, he was a hardware designer, microprogrammer, and writer of manuals at Symbolics. His interests include advanced software development and architectures for symbolic processing.

Moon received the BS degree in mathematics from MIT in 1975.

Readers may write to the author at Symbolics, Inc., 11 Cambridge Center, Cambridge, MA 02142. His e-mail address is Moon@Stony-Brook.SCRC.Symbolics.COM on the ARPA Internet.