"Any clod can have the facts, but having opinions is an *art*."

Charles McCabe,
San Francisco *Chronicle*

THE OPEN CHANNEL

The Open Channel is exactly what the name implies: a forum for the free exchange of technical ideas. Try to hold your contributions to one page maximum in the final magazine format (about 1000 words).

We'll accept anything (short of libel or obscenity) so long as it's submitted by a member of the Computer Society. If it's really bizarre we may require you to get another member to cosponsor your item.

Send everything to Jim Haynes, Computer Center, UC Santa Cruz, Santa Cruz, CA 95064.

Address space unification

Computers demonstrate man's ability to number things. They also show man's tendency to isolate things that are different in nature and later, as they are better understood, to unify these differences under a more general category.

An example of the latter tendency is address space unification. An address space is any numbered collection constituting "computer memory," such as registers, ROM, RAM, I/O ports, or disk sectors, with each collection having a separate space. Address space unification is a virtual address space encompassing two or more address spaces.

Address space unification is most useful on large systems devoted to the support of programming in general. As a result, the overriding factor is the uniformity and consistency of the programming environment. The abstract address space is partitioned among several computers and simulated by paging to and from disk. Program and data are usually somewhat separated; there are special locations for I/O devices and a special region for built-in op-codes. This approach is more useful than a variety of special op-codes, operating system calls, and subroutine calls.

Historically, at least, three address space unifications have taken place: program and data, main memory and I/O ports, and main memory and secondary or disk memory. Two further address space unifications are currently feasible, one of which provides virtual communications and the other, extensible machine language, provides the subroutine mechanism. Each of these unifications simplifies the set of data structures the programmer must deal with.

Virtual communications. If the address space is large enough and communications are fast enough, which is likely with 32-bit microprocessors and fiber optics, respectively, several computers can share the same address space.[1] Each computer owns a portion of the address space. Communication then consists of memory moves and moves with interlocks within the same address space, primarily to simplify the communications protocol associated with multiple computer systems.

Of course, for this to work in a reliable fashion, each computer must protect its own portion of the address space. Presumably a computer wishing to communicate would go through a handshake process to gain access to part of another computer's space. Thus there must be public regions of memory, regions for communications buffers, etc. The advantage of the common address space over traditional communications is that protocols can be individually arranged and programmed, so that those applications without full generality need not support a complicated protocol.

The tradeoff is that by making the communications standard more primitive (address and data versus a sender and receiver discipline), higher level disciplines become more flexible.

Extensible machine language. If computer op-codes look like subroutine calls, or vice versa, then a burden is lifted from the software compatibility issue.[2] An op-code not implemented on a given computer can be simulated by an equivalent subroutine call. This tends to do away with the philosophical difference between assembly language and high-level language programming, since high-level constructs can be implemented by subroutines that have the same user appearance as op-codes.

To have this equivalence of op-codes and subroutine calls, the computer architecture is somewhat constrained. First, the records or format structures of op-codes and subroutine calls must be made identical. The simplest way to do this is to have no record structure; i.e., the instruction has only the op-code and no register or memory addressing fields. The various threaded code systems have this property, so what follows has many similarities to Forth and interpretive systems.[3-5]

0018-9162/83/0200-0086$01.00 © 1983 IEEE

Two stacks for each process reside in a large, uniform address space. The stacks come complete with base or global pointers, top pointers, and level or frame pointers. One stack is used for values and is called the value stack (parameters, expression intermediaries, results, locals); the other is used for return addresses and looping information and is called the control stack. Conceptually, what corresponds to the program counter is considered to be the top of the control stack in the same way that working accumulators (registers) correspond to the top of the value stack.

The instruction processing cycle, or IPC, which is looking for an address, advances the program counter and examines the address at the referenced location. The equivalent of op-codes are addresses in a reserved section of memory—usually low memory. If the address examined by the IPC is one of these reserved addresses, the corresponding operation is performed. If not, indirection takes place without modifying the program counter until a reserved address is found. (The formats or recorded structures of op-codes and subroutine calls are those of a memory address and nothing more.)

Such indirect address trails may lead to a subroutine. Subroutines begin with the enter subroutine op-code (address) and exit with the exit subroutine op-code. Subroutine entry stacks the program counter on the control stack; subroutine exit unstacks it. While the subroutine executes, the new program counter scans it. Programs for this architecture are tree structures, and the IPC is a tree-scanning mechanism.

Thus the difference between an op-code and a subroutine is whether its address is in reserved memory. Indirection is the default and does not stop until a reserved memory location is referenced. A subroutine can consist of a single op-code address or an enter subroutine code, followed by the sequence of codes for the body of the subroutine, and the exit subroutine code.

On a stack-oriented machine, the parameters for both op-codes and subroutines are taken from the value stack, and the results are returned to it. Thus, both op-codes and subroutines, unlike all traditional architectures, have a similar interface with the two stacks. Even the single-stack machines such as ICL, Burroughs, HP, and DEC use different parameter mechanisms for op-codes and subroutine calls; subroutines use a frame pointer and op-codes use implied parameters on top of the stack.

The set of op-codes must include a set of primitives similar to those of Forth (see list below). Additional op-codes can be as complete or as specialized as required. Non-built-in op-codes are emulated by subroutines. The equivalent of the linker need only know the address of the subroutine or op-code and not which is which.

There is some advantage in coding immediate values within the code string, where they would be preceded by the utilizing op-code or subroutine call. This to some extent breaks up the consistency of code strings as simple address lists and may prove to be important.

Since op-code references should be more common than subroutine references, using a compaction scheme for op-code addresses will result in better memory utilization. Any variable-size encoding of addresses will do: typically, 2^7 addresses in a byte, 2^{14} in 16 bits, and 2^{29} in 32 bits.

The programmer's algebra. So how exactly do these methods translate into programming tasks? The data structures of programming are those of traditional mathematics: natural number, integer, real, complex, vector, array, and so on. To these are added the address (usually a variation on the natural number) and compositions of the above (records and files). With op-codes and subroutine calls mapped into addresses, no additional structures are required. The body of a subroutine is simply a list of addresses, i.e., an array or list of natural numbers.

With traditional op-codes, each instruction must be thought of as a record with certain special characteristics: the record is executable, and the fields of the record reference registers or other context-sensitive information. In many instruction sets, exceptions are more common than rules.

The above architecture provides a clean, convenient, and comprehensive programming environment (or virtual machine). In particular, subroutines are very compact and efficient, and this alone tends to promote well-structured programming. The emphasis, under the dual-stack discipline, is on the subroutine interface. Of course, as subroutines become shorter they also become more numerous. However, modern programming language practice (i.e., type checking) is adequate to handle the large number of subroutines.

Op-codes

Small constants (say 0-63)
Load and store (byte, word, double word, etc.)
- Immediate (from code string)
- Nearby (symbol table values, reached by indirection)
- Absolute (address on top of value stack)
- Relative (to base of stack, to top of stack, to frame pointer)

Arithmetic operators
String operators
Control operators
- Conditional and case
  - Ascend (pop control stack)
  - Descend (push control stack via subroutine entry)
  - Skip relative
- Looping
  - Entry (stack loopback address, loop count, loop delta, loop limit)
  - Exit (pop control stack)
  - Countdown (modify loop count and conditionally pop control stack)

Enter and exit subroutine
- Global and relative

James C. Brakefield
5803 Cayuga
San Antonio, Texas 78228

References

1. J. Brakefield, "In the Limit," *Computer*, Vol. 14, No. 3, Mar. 1981, p. 88.
2. J. Brakefield, "Just What is an Opcode?—or a Universal Computer Design," *Computer Architecture News*, Vol. 10, No. 4, 1982, pp. 31-34.
3. L. Brodie, *Starting FORTH*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
4. P. Kogge, "An Architectural Trail to Threaded-Code Systems," *Computer*, Vol. 15, No. 3, Mar. 1982, pp. 22-32.
5. J. Brakefield, "Talk on Interpreters," *Computer Architecture News*, Vol. 10, No. 6, 1982, pp. 21-28.