Just what is an op-code?
or a universal computer design

James Brakefield
San Antonio, Texas

Abstract:

The following computer architecture offers extensible machine language. That is it has an open-ended instruction set. It does this by using the same format for both op-codes and subroutine calls. Theoretically this means that the programmer can chose an instruction set for the application at hand. Ie., the machine language representation of a program can correspond to a high level textual representation of a program.

The mechanism of instruction processing for this architecture is essentially a way of encompassing the major categories of interpreters: Direct threaded code, indirect threaded code, and P-code or compressed threaded code. The mechanism is closest in spirit to that of Forth and can be considered a hardware implementation of Forth.

Due to the simplicity of instruction set representation, the design is not tied to any one word size. Thus, it is implementable on all the major word sizes (16, 24, 32, 36, 48, 60, or 64 bits).

I hope this is perceived as an elegant design.

The Text:

The address space is considered open ended. With current technology, an address of 16 to 64 bits can be envisioned. Within this address space reside two stacks, the value stack and the control stack. They are to function as in Forth. The top of the control stack is considered the "program counter." It serves to scan the code strings which are equivalent to instructions on other kinds of computers. Within a reserved portion of the address space are the I/O devices as on the PDP11 * series of computers. Also within the address space, usually starting at zero, are the "op-codes." This is new. Each reserved op-code address corresponds to a specific instruction. This paper does not consider which instructions should be included in the reserved set.

The instruction execution process is that the code scanner is advanced and the referenced code (address) is fetched. If it is a reserved op-code address, that op-code is executed. It it is not, an indirection takes place until a reserved op-code address is found. The reserved op-code is then executed.

The trail of indirect address is often useful. It is assumed that a specific indirect address within the trail can be retrieved at following op-codes.

Subroutines are programmed by preceding them with the "enter subroutine" op-code and following them by the "exit subroutine" op-code. To reference (i.e., to call) a subroutine, an indirect address trail must lead to the "enter subroutine" op-code. Usually the trail is but a single pointer to the subroutine.

The "enter subroutine" op-code pushes the address of the following memory location onto the control stack so that it

becomes the new code scanner. The previous code scanner is stacked and will pop-up to resume being the code scanner after the "exit subroutine" op-code is executed.

Within this mannner of executing instructions, several conveniences are possible. Referencing operands relative to the top of the value stack is handy for utilizing intermediate calculations. Referencing relative to the top of the control stack can be used to fetch loop counts. Referencing relative to the base of the value stack can be used for global values.

Frame relative references for local values and parameters is best done with a special register (frame pointer) for this purpose. Tracing indirect address trails is helped by registers which remember say the bottom of a trail and the first indirection of a trail. Often the first indirection is through the symbol table entry. With the address of the symbol table entry in a register, relative references can interrogate any facet of the symbol.

The normal "machine code" for this architecture is left Polish. I.e., operands are specified followed by the operator. The operators predominately use the value stack for operands and the result. Due to the ease of creating subroutines, an alternate form is possible. If each node of a parse tree is a separate routine, then the parse tree can be executed directly. This requires that the parse tree nodes consist of the "enter subroutine" code, pointers (addresses) to the operands, the operator code or pointer, and last the "exit subroutine" code. The enter and exit codes are the only overhead over normal parse tree representation.

An interesting duality exists in this design. The common routine for dealing with variables in Forth loads the address of the value of the variable. In this design, the op-code for this would precede the value and be referenced in a code string by address. The actual operation is to push the address of the next memory location onto the value stack. This is the dual of entering a subroutine, the mechanism is the same, only the stack pushed has changed.

There is much of this sort to be considered and reflected upon in this design.

Postlude:

The result of this design is an open ended stack machine: The control stack is not theoretically necessary. However, by virtue of it, control information is removed from the value stack leaving the subroutine interface much cleaner (parameters for the subroutine on entry to a subroutine, results of the subroutine on exit from the subroutine). The mapping of op-codes into reserved addresses makes user written subroutines (with non-reserved addresses) similar in representation to the built-in op-codes.

The net effect is that there is no distinction in the code-string between a user written subroutine and a built-in op-code. Both are represented by addresses and both use the two stacks in a similar fashion.

Implementation:

This is the template. Adapting it for a particular word size, a particular address space, and a particular set of built-in operators is straight forward. Usually the word size is sufficient to encompass the address. The value stack then (unless directed otherwise) operates on word size items. Unless the number of bits in a full address is greatly smaller than the number in a word, the control stack does also.

To save memory the more common addresses can be encoded. Typically, take the most common op-codes and encode them to be addresses Ø thru 127. This is 7-bits and with byte (8-bit) organized memory, the eighth bit serves as a flag saying that this is an 8-bit address. Similar encoding gives addresses Ø thru 16383 in two bytes, Ø thru 536870911 (2**29-1) in four bytes, and Ø thru 2**61 in eight bytes. In those situations where it is envisioned that user subroutine references will be very common, this encoding can be lookup-table driven. The codes Ø thru some limit, say 4Ø95, will be looked up in a read/write table. If the code is a builtin op-code, the table gives the micro-code starting point. If the code is a user subroutine, the table gives the full address of the subroutine.

For the 8, 16, 32, and 64-bit word size versions, it is envisioned that arithmetic would be provided for 8, 16, 32, and 64-bit signed and unsigned integers. Also arithmetic would be provided for 16, 32, 64, and 8Ø-bit floating point. Additional sizes can be supported if needed.

On machines supporting the larger address spaces, say 32-bits and above, some bits can be removed from the address and used to specify the nature and size of the object that the address references. Also one has the choice of bit addressing as opposed to byte or word addressing. Addresses so embellished will be called descriptors.

On the non-power of two word sizes with descriptors, the descriptor can be embellished to show the alignment of the operand. As an example, on a 48-bit word size, the descriptor can reference to the bit or to the 3-bit multiple. The power-of-two multiples of the bit reference are 1, 2, 4, 8, 16, 32, and 64-bit items. The power-of-two multiples of the 3-bit reference are 3, 6, 12, 24, and 48-bit items. This set of items would be a relatively complete set for most data processing.

In summary, the beauty of this design is that the instruction set is open ended by virtue of subroutine calls having the same format as op-codes. The use of two stacks gives all the elegance of the single stack design but with the duality of control and data. Control information and value information are separated, each with its own stack.

The process state is compact. This should make multi-processor versions simple to implement and control (especially if they share the same address space).

The built-in op-codes can be as minimal or as complete as desired. Vector and array processor versions need only use the value stack as necessary for parameters.

On this design upward, downward, and crossword compatability is automatic. The only requirement is to remap addresses to the new values. Non-builtin op-codes are simply replaced by the addresses of the emulation routines. Immediate data in code strings need also to be similarly remapped.

I look forward to using this design to facilitate the programming of complex programs of any size. The control and flexibility this design gives the programmer over algorithm decomposition will greatly facilitate large software packages. The programming model this design presents (two stacks, identity of op-codes and subroutines) is simple, complete, and conductive to well structured software.

* Trademark of Digital Equipment Corporation

References:

1. Moore, C. Aug. 1980. The evolution of FORTH--An unusual language. Byte 5:8:76-92.

2. Kogge, P. Mar. 1982. An architectural trail to threaded-code systems. IEEE Computer 15:3:22-32.

3. Brodie, L. 1981. Starting FORTH. Englewood Cliffs, N.J.: Prentice-Hall.

4. Brakefield, J. May 1982 (tenative). Interpreter mechanism for computers. 68 Micro-Journal.

5. Brakefield, J. Mar. 1982. DECUS Meeting: FORTH extensions. SIGPLAN NOTICES, pp. 20-21.

6. Brakefield, J. Mar. 1981. In the limit. IEEE Computer 14:3:88.

7. Brakefield, J. Oct. 1980. Is 32 bits of address too much? Computer Architecture News, pp. 39-40.

8. Brakefield, J. May 1980. A coding discipline for microprocessors. IEEE Computer 13:5:118-119.