Ambiguous Machine Architecture and Program Efficiency

Tanenbaum, A.S.

published in
Computer Architecture News
1977

document version
Publisher's PDF, also known as Version of record

citation for published version (APA)
Tanenbaum, A. S. (1977). Ambiguous Machine Architecture and Program Efficiency. *Computer Architecture News*, *6*(3), 11-13.

Ambiguous Machine Architecture and Program Efficiency

Andrew S. Tanenbaum
Computer Science Group
Mathematics Department
Vrije Universiteit
Amsterdam, The Netherlands

It is usually accepted as an article of faith among machine architects and systems programmers that the exact format and operation of all machine instructions should be defined down to the bit level. Assembly language programmers have been known to get quite agitated when the manufacturer's manuals are too vague about what happens to some obscure flip flop when a certain peculiar instruction encounters some exceptional condition. Although we certainly do not intend to defend fuzzy manufacturer's manuals, it is the purpose of this paper to show how deliberate ambiguity can enhance execution efficiency in certain circumstances.

It is common practice for computer manufacturers to define some common machine architecture to be used on a family of machines that span a considerable range of price and performance. By machine architecture we mean those aspects of the hardware visible to the programmer-e.g. the number of general registers, memory addressing, instruction set, etc., but not, for example, whether the logic is hardwired or microprogrammed, whether ECL or TTL is used, etc. This common architecture is then supported by interpretation on a variety of different (and incompatible) computers, IBM 370 "machine language" is a well known example of a common architecture, which has been implemented on a range of totally different computers ranging from a 20 bit machine (used for the 370/115) to a 108 bit machine (used for the 370/168).

It is our contention that such common machine architectures are overspecified, and by intentionally leaving certain aspects undefined, more efficient implementations would be possible in many circumstances, without impairing portability between models in any significant way. This point will now be illustrated by means of several examples.

All computers have an instruction used to call subroutines. This instruction deposits the return address-specifically defined as the address of the first instruction to be executed after the subroutine returns- in a register, in memory, or on the stack. This is an example of overspecification. As long as subroutines return using the standard RETURN instruction, there is no reason why the CALL instruction could not deposit return address + 4 or 2* return address or any other function of it.

The exact bit pattern deposited is none of the programmar's business; all (s)he has a right to demand is that RETURN returns properly. Programs that actually inspect the bit pattern (as opposed to merely moving it around) are obscene and should not be pandered to.

To see where this overspecification hurts, consider a common machine architecture in which addresses 0,1,2... address consecutive 16 bit words, not 8 bit bytes. Word addresses may be either odd or even. The smallest model of this family may be implemented on an LSI microcomputer whose memory is byte oriented i.e. consecutive addresses refer to consecutive 8 bit bytes, not consecutive 16 bit words. For efficiency, the interpreter may keep track of the program counter by the physical address in its own memory where the next instruction lies. If the target program storage area begins at location 1000 in the LSI host computer's memory, when target instruction 30 is the next to be executed, the internal pc will be 1060.

If this next instruction is a two word CALL instruction, technically the return address, 32, should be deposited somewhere. The RETURN instruction would then multiply the return address by 2 and add 1000 to find out where it should continue executing. Obviously it would be much more efficient if the interpreter were free to simply deposit 1064 instead of 32, since that speeds up both the CALL and return instructions. On other models, however, different functions of the true pc might be more appropriate.

As long as the program does not attemt to perform arithmetic on the return address, this proposed ambiguity will not impair portability at all. The idea here is somewhat reminiscent of abstract data types, where the internal representation is not defined, but only the effects of operations on them are.

The same kind of arguments as given above can be used to justify not pinning down the representation of any other essentially internal registers that must occasionally be stored in memory, but whose bit pattern is of no consequence to the programmer. This argument is doubly valid for machines programmed largely in high level languages, since there the programmer is not even technically able to commit improprieties with the bits.

Another example is the type of instruction that tests or compares operands and leaves TRUE or FALSE somewhere. By not defining the numerical value of TRUE and FALSE in the common architecture, the designers leave each implementer the choice of the most efficient values to use. Of course there must be instructions of the form JUMP TRUE and JUMP FALSE, so the machine can be programmed without knowledge of the numerical values.

Yet another example is the overspecification of condition code settings after instructions. After compare and move type instructions, the condition code settings are important of course, but after many instructions no one really cares. By officially announcing that the condition codes are undefined after certain instructions, programmers will not be tempted into doing tricky things with them, and the interpreters on the smaller models will not be forced to go to the trouble of setting them.

Creative ambiguity can also be used at the level of symbolic assembly language. The previous examples had to do with transporting object programs between members of a series. Now we are concerned with transporting symbolic assembly language programs, possibly compiler output, and reassembling on the new machine. In particular we consider the possibility that the new machine has a superset of the old machine's instruction set (as is commonly the case), or equivalently that a machine is upgraded in the field by the addition of new instructions.

The most useful ambiguity at this level would be for instruction lengths to be indeterminate. Among other effects, constructions such as JUMP *+20 (* means this location) would be ambiguous (since the programmer would have no way of knowing which instruction was at *+20) hence forbidden. Likewise JUMP LABEL+20 would be forbidden for the same reason. Jumps could only have explicitly labeled instructions as the destination address.

One use to which this ambiguity could be put would be to have multiple machine language instructions corresponding to a given mnemonic. Load constant (move immediate) instructions might have a format with a 16 bit constant field on all models, but also 4 and 8 bit versions on some models (presumably the larger ones, since this addition improves performance but might cost more money for additional interpreter storage). The assembler would then choose the most appropriate format. If the programmer is allowed to make assumptions about instruction length, this idea does not work. Note that few compilers would suffer from such a restriction; only "tricky" assembly language programmers would be inconvenienced.

Another use of indeterminate instruction lengths is to allow the assembler to make certain optimizations, e.g. replace the sequence LOAD X; ADD 1; STORE X by the instruction INCREMENT X. (The INCREMENT instruction might have been added later, or only be present on some models). If the programmer is free to jump to an arbitrary instruction, such optimizations are impossible. By forcing the programmer, in effect, to label all instructions referenced elsewhere in the program, the assembler can see whether or not such optimizations are safe. The value of putting this kind of optimization in the assembler, is that every compiler can then use it as the last pass, eliminating the neef for putting the same optimization code in each compiler.

It should be pointed out, however, that introducing intentional ambiguity may make assembly language programming more difficult. However considering the decreasing importance of assembly language programming, this is a minor objection.