2.18 Fallacies and pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the subsequent instruction until a counter steps down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times as fast. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast as the complex move instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand the concepts in COD Chapters 4 (The Processor) and 5 (Large and Fast: Exploiting Memory Hierarchy) thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is another situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some old C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore these hints, because the compiler does a better job at allocation than the programmer does.

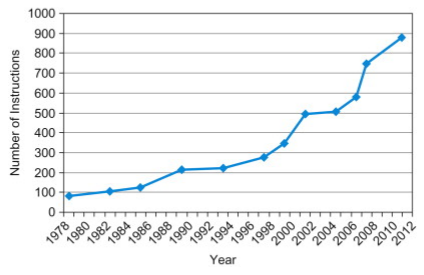
Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are the protracted time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C or Java. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and recent computers. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to forthcoming machines; it also makes the software easier to maintain and allows the program to run on more brands of computers.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

While backwards binary compatibility is sacrosanct, the figure below shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

Figure 2.18.1: Growth of x86 instruction set over time (COD Figure 2.44).

While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.



Pitfall: Forgetting that sequential word or doubleword addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word or doubleword can be found by incrementing the address in a register by one instead of by the word or doubleword size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in COD Figure 2.13 (Illustration of the stack allocation ...), the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

PARTICIPATION ACTIVITY 2.18.1: Pitfall: Pointing to a local variable.

Start 2x speed

Memory

```
int* GetElement()
{
   // Local array
   int array[3] = {15, 37, 42};

   // Point to array's first element
   int* p = &array[0];

   // Return pointer
   return p;
}


int main()
{
```

p ▶

```
    // Call procedure GetElement()
    int* arrayElement = GetElement();

    // Print value of array's first element
    printf("%d", *arrayElement);
}
```

p no longer points to array

PARTICIPATION ACTIVITY 2.18.2: Fallicies and pitfalls.

1) Using standard instructions can result in higher performance than using powerful instructions crafted specifically for an architecture.
   - ○ True
   - ○ False

2) Current C compilers tend to ignore coded hints regarding register allocation.
   - ○ True
   - ○ False

3) Writing code in a high-level language, such as C, results in more lines of code than writing in assembly language. Therefore, C programs take longer to write and debug.
   - ○ True
   - ○ False

4) Assembly language programs are not easily portable because of the many different architectures of current and future computers.
   - ○ True
   - ○ False

5) Assuming the addresses are in registers, if the size of a word is 4 bytes, the address of the next word can be found by adding 1 to the current word's address.
   - ○ True
   - ○ False

6) Memory containing variables local to a procedure may be reused as soon as the procedure returns.
   - ○ True
   - ○ False

Provide feedback on this section