PRACTICE

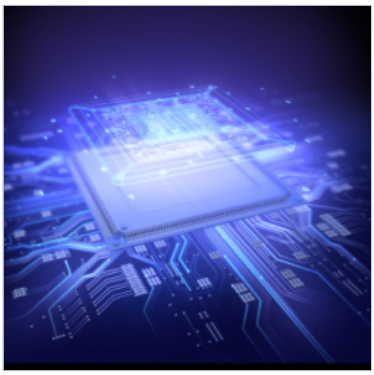
The Long Road to 64 Bits

By John Mashey

Communications of the ACM, January 2009, Vol. 52 No. 1, Pages 45-53
10.1145/1435417.1435431

Comments

VIEW AS: SHARE:



iStockPhoto

Shakespeare's words often cover circumstances beyond his wildest dreams. *Toil and trouble* accompany major computing transitions, even when people plan ahead. Much of tomorrow's software will still be driven by decades-old decisions. Past decisions have unanticipated side effects that last decades and can be difficult to undo.

For example, consider the overly long, often awkward, and sometimes contentious process by which 32-bit microprocessor systems evolved into 64/32-bitters needed to address larger storage and run mixtures of 32- and 64-bit user programs. Most major general-purpose CPUs now have such versions, so bits have "doubled," but "toil and trouble" are not over, especially in software.

This example illustrates the interactions of hardware, languages (especially C), operating system, applications, standards, installed-base inertia, and industry politics. We can draw lessons ranging from high-level strategies down to programming specifics.

Back to Top

Fundamental Problem (late 1980s)

Running out of address space is a long tradition in computing, and often quite predictable. Moore's Law grew DRAM approximately four times bigger every three to four years, and by the mid-1990s, people were able to afford 2GB to 4GB of memory for midrange microprocessor systems, at which point simple 32-bit addressing (4GB) would get awkward. Ideally, 64/32-bit CPUs would have started shipping early enough (1992) to have made up the majority of the relevant installed base before they were actually needed. Then people could have switched to 64/32-bit operating systems and stopped upgrading 32-bit-only systems, allowing a smooth transition. Vendors naturally varied in their timing, but shipments ranged from "just barely in time" to "rather late." This is somewhat odd, considering the long, well-known histories of insufficient address bits, combined with the clear predictability of Moore's Law. All too often, customers were unable to use memory they could easily afford.

Some design decisions are easy to change, but others create long-term legacies. Among those illustrated here are:

- Some unfortunate decisions may be driven by real constraints (1970: PDP-1116-bit).
- Reasonable-at-the-time decisions turn out in 20-year retrospect to have been suboptimal (19761977: usage of C data types). Some better usage recommendations could have saved a great deal of toil and trouble later.
- Some decisions yield short-term benefits but incur long-term problems (1964: S/360 24-bit addresses).
- Predictable trends are ignored, or transition efforts underestimated (1990s: 32-> 64/32).

Constraints. Hardware people needed to build 64/32-bit CPUs at the right timeneither too early (extra cost, no market), nor too late (competition, angry customers). Existing 32-bit binaries needed to run on upward-compatible 64/32-bit systems, and they could be expected to coexist forever, because many would never need to be 64 bits. Hence, 32 bits could not be a temporary compatibility feature to be quickly discarded in later chips.

Software designers needed to agree on whole sets of standards; build dual-mode operating systems, compilers, and libraries; and modify application source code to work in both 32- and 64-bit environments.

Numerous details had to be handled correctly to avoid redundant hardware efforts and maintain software sanity.

Solutions. Although not without subtle problems, the hardware was generally straightforward, and not that expensivethe first commercial 64-bit micro's 64-bit data path added at most 5% to the chip area, and this fraction dropped rapidly in later chips. Most chips used the same general approach of widening 32-bit registers to 64 bits. Software solutions were much more complex, involving arguments about 64/32-bit C, the nature of existing software, competition/cooperation among vendors, official standards, and influential but totally unofficial ad hoc groups.

Legacies. The IBM S/360 is 40 years old and still supports a 24-bit legacy addressing mode. The 64/32 solutions are at most 15 years old, but will be with us, effectively, forever. In 5,000 years, will some software maintainer still be muttering, "Why were they so dumb?"[4]

We managed to survive the Y2K problemwith a lot of work. We're still working through 64/32. Do we have any other problems like that? Are 64-bit CPUs enough to help the "Unix 2038" problem, or do we need to be working harder on that? Will we run out of 64-bit systems, and what will we do then? Will IPv6 be implemented widely enough soon enough?

All of these are examples of long-lived problems for which modest foresight may save later toil and trouble. But software is like politics: Sometimes we wait until a problem is really painful before we fix it.

Back to Top

Problem: CPU Must Address Available Memory

Any CPU can efficiently address some amount of virtual memory, done most conveniently by flat addressing, in which all or most of the bits in an integer register form a virtual memory address that may be more or less than actual physical memory. Whenever affordable physical memory exceeds the easily addressable, it stops being easy to throw memory at performance problems, and programming complexity rises quickly. Sometimes, segmented memory schemes have been used with varying degrees of success and programming pain. History is filled with awkward extensions that added a few bits to extend product life a few years, usually at the cost of hard work by operating-system people.

Moore's Law has increased affordable memory for decades. Disks have grown even more rapidly, especially since 1990. Larger disk pointers are more convenient than smaller ones, although less crucial than memory pointers. These interact when mapped files are used, rapidly consuming virtual address space.

In the mid-1980s, some people started thinking about 64-bit microsfor example, the experimental systems built by DEC (Digital Equipment Corporation). MIPS Computer Systems decided by late 1988 that its next design must be a true 64-bit CPU, and announced the R4000 in 1991. Many people thought MIPS was crazy or at least premature. I thought the system came just barely in time to develop software to match increasing DRAM, and I wrote an article to explain why.[5] The issues have not changed very much since then.

N-bit CPU. By long custom, an N-bit CPU implements an ISA (instruction set architecture) with N-bit integer registers and N (or nearly N) address bits, ignoring sizes of buses or floating-point registers. Many 32-bit ISAs have 64- or 80-bit floating-point registers and implementations with 8-, 16-, 32-, 64-, or 128-bit buses. Sometimes marketers have gotten this confused. I use the term 64/32-bit here to differentiate the newer microprocessors from the older 64-bit word-oriented supercomputers, as the software issues are somewhat different. In the same sense, the Intel 80386 might have been called a 32/16-bit CPU, as it retained complete support for the earlier 16-bit model.

Why 2N-bits? People sometimes want wider-word computers to improve performance for parallel bit operations or data movement. If one needs a 2N-bit operation (add, multiply, and so on), each can be done in one instruction on a 2N-bit CPU, but requires longer sequences on an N-bit CPU. These are straightforward low-level performance issues. The typical compelling reason for wider words, however, has been the need to increase address bits, because code that is straightforward and efficient with enough address bits may need global restructuring to survive fewer bits.

Addressingvirtual and realin a general-purpose system. User virtual addresses are mapped to real memory addresses, possibly with intervening page faults whereby the operating system maps the needed code or data from disk into memory. A user program can access at most VL (virtual limit) bytes, where VL starts at some hardware limit, then sometimes loses more space to an operating system. For example, 32-bit systems easily have VLs of 4, 3.75, 3.5, or 2GB. A given program execution uses at most PM (program memory) bytes of virtual memory. For many programs PM can differ greatly according to the input, but of course PM VL.

The RL (real limit) is visible to the operating system and is usually limited by the width of physical address buses. Sometimes mapping hardware is used to extend RL beyond a too-small "natural" limit (as happened in PDP-11s, described later). Installed AM (actual memory) is less visible to user programs and varies among machines without needing different versions of the user program.

Most commonly, VL RL AM. Some programs burn virtual address space for convenience and actually perform acceptably when PM >> AM: I've seen cases where 4:1 still worked, as a result of good locality. File mapping can increase that ratio further and still work. On the other hand, some programs run poorly whenever PM > AM, confirming the old proverb, "Virtual memory is a way to sell real memory."

Sometimes, a computer family starts with VL RL AM, and then AM grows, and perhaps RL is increased in ways visible only to the operating system, at which point VL << AM. A single program simply cannot use

easily buyable memory, forcing work to be split and making it more difficult. For example, in Fortran, the declaration REAL X (M, M, M) is a three-dimensional array. If M = 100, *X* needs 4MB, but people would like the same code to run for M = 1,000 (4GB), or 6,300 (1,000GB). A few such systems do exist, although they are not cheap. I once had a customer complain about lack of current support for 1,000GB of real memory, although later the customer was able to buy such a system and use that memory in one program. After that, the customer complained about lack of 10,000GB support...

Of course, increasing AM in a multitasking system is still useful in improving the number of memory-resident tasks or reducing paging overhead, even if each task is still limited by VL. Operating system code is always simplified if it can directly and simply address all installed memory without having to manage extra memory maps, bank registers, and so on.

Running out of address bits has a long history.

Back to Top

Mainframes, Minicomputers, Microprocessors

The oft-quoted George Santayana is apropos here: "Those who do not remember the past are condemned to repeat it."

Mainframes. IBM S/360 mainframes (circa 1964; see accompanying Chronology sidebar) had 32-bit registers, of which only 24 bits were used in addressing, for a limit of 16MB of core memory. This was considered immense at the time. A "large" mainframe offered at most 1MB of memory, although a few "huge" mainframes could provide 6MB. Most S/360s did not support virtual memory, so user programs generated physical addresses directly, and the installed AM was partitioned among the operating system and user programs. The 16MB limit was unfortunate, but ignoring (not trapping) the high-order 8 bits was worse. Assembly language programmers cleverly packed 8-bit flags with 24-bit addresses into 32-bit words.

As virtual addressing S/370s (1970) enabled programs that were larger than physical memory allowed, and as core gave way to DRAM (1971), the 16MB limit grew inadequate. IBM 370XA CPUs (1983) added 31-bit addressing mode, but retained a (necessary) 24-bit mode for upward compatibility. I had been one of those "clever" programmers and was somewhat surprised to discover that a large program (the S/360 ASSIST assembler) originally written in 1970 was still in use in 2006in 24-bit compatibility mode, because it wouldn't run any other way. Compiler code that had been "clever" had long since stopped doing this, but assembly code is tougher. ("The evil that men do lives after them, the good is oft interred with their bones."Shakespeare, again, *Julius Caesar)*

Then, even 31-bit addressing became insufficient for certain applications, especially databases. ESA/370 (1988) offered user-level segmentation to access multiple 2GB regions of memory.

The 64-bit IBM zSeries (2001) still supports 24-bit mode, 40-plus years later. Why did 24-bit happen? I'm told that it was all for the sake of one low-cost early model, the 360/30, where 32 bits would have run slower because it had 8-bit data paths. These were last shipped more than 30 years ago. Were they worth the decades of headaches?

Minicomputers. In the 16-bit DEC PDP-11 minicomputer family (1970), a single task addressed only 64KB, or in later models (1973), 64KB of instructions plus 64KB of data. "The biggest and most common mistake that can be made in computer design is that of not providing enough address bits for memory addressing and management, " C. Gordon Bell and J. Craig Mudge wrote in 1978. "The PDP-11 followed this hallowed tradition of skimping on address bits, but was saved on the principle that a good design can evolve through at least one major change. For the PDP-11, the limited address space was solved for the short run, but not with enough finesse to support a large family of minicomputers. This was indeed a costly oversight."[2]

To be fair, it would have been difficult to meet the PDP-11's cost goals with 32-bit hardware, but I think DEC did underestimate how fast the price of DRAM memory would fall. In any case, this lasted a longtimethe PDP-11 was finally discontinued in 1997!

The PDP-11/70 (1976) raised the number of supportable concurrent tasks, but any single program could still only use 64KI + 64KD of a maximum of 4MB, so that individual large programs required unnatural acts to split code and data into 64KB pieces. Some believed this encouraged modularity and inhibited "creeping featurism" and was therefore philosophically good.

Although the 32-bit VAX-11/780 (1977) was only moderately faster than the PDP-11/70, the increased address space was a major improvement that ended the evolution of high-end PDP-11s. VAX architect William Strecker explained it this way: "However, there are some applications whose programming is impractical in a 65KB address space, and perhaps more importantly, others whose programming is appreciably simplified by having a large address space."[7]

Microprocessors. The Intel 8086's 16-bit ISA seemed likely to fall prey to the PDP-ll's issues (1978). It did provide user-mode mechanisms for explicit segment manipulation, however. This allowed a single program to access more than 64KB of data. PC programmers were familiar with the multiplicity of memory models, libraries, compiler flags, extenders, and other artifacts once needed. The 80386 provided 32-bit flat addresses (1986), but of course retained the earlier mechanisms, and 16-bit PC software lasted "forever." The intermediate 80286 (1982) illustrated the difficulty of patching an architecture to get more addressing bits.

The 32-bit Motorola MC68000 (1979) started with a flat-addressing programming model. By ignoring the high 8 bits of a 32-bit register, it exactly repeated the S/360 mistake. Once again, "clever" programmers found

uses for those bits, and when the MC68020 (1984) interpreted all 32, some programs broke (for example, when moving from the original Apple Macintosh to the Mac II).

Fortunately, 64-bit CPUs managed to avoid repeating the S/360 and MC68000 problem. Although early versions usually implemented 40 to 44 virtual address bits, they trapped use of not-yet-implemented high-order v bits, rather than ignoring them. People do learn, eventually.

Back to Top

Lessons

Even in successful computer families created by top architects, address bits are scarce and are totally consumed sooner or later.

Upward compatibility is a real constraint, and thinking ahead helps. In the mainframe case, a 24-bit "first-implementation artifact" needed hardware/software support for 40-plus years. Then a successful minicomputer family's evolution ended prematurely. Finally, microprocessors repeated the earlier mistakes, although the X86's segmentation allowed survival long enough to get flat-address versions.

Back to Top

The 32- to 64-bit Problem in the Late 1980s

By the late 1980s, Moore's Law seemed cast in silicon, and it was clear that by 19931994, midrange microprocessor servers could cost-effectively offer 2GB-4GB or more of physical memory. We had seen real programs effectively use as much as 4:1 more virtual memory than installed physical memory, which meant pressure in 19931994, and real trouble by 1995. As I wrote in *Byte* in September 1991:[5]

"The virtual addressing scheme often can exceed the limits of possible physical addresses. A 64-bit address can handle literally a mountain of memory: Assuming that 1 megabyte of RAM requires 1 cubic inch of space (using 4-megabit DRAM chips), 2**64 bytes would require a square mile of DRAM piled more than 300 feet high! For now, no one expects to address this much DRAM, even with next-generation 16MB DRAM chips, but increasing physical memory slightly beyond 32 bits is definitely a goal. With 16MB DRAM chips, 2**32 bytes fits into just over 1 cubic foot (not including cooling)feasible for deskside systems...

"Database systems often spread a single file across several disks. Current SCSI disks hold up to 2 gigabytes (i.e., they use 31-bit addresses). Calculating file locations as virtual memory addresses requires integer arithmetic. Operating systems are accustomed to working around such problems, but it becomes unpleasant to make workarounds; rather than just making things work well, programmers are struggling just to make something work...."

So, people started to do something about the problem.

SGI (Silicon Graphics). Starting in early 1992, all new SGI products used only 64/32-bit chips, but at first they still ran a 32-bit operating system. In late 1994, a 64/32-bit operating system and compilers were introduced for large servers, able to support both 32-bit and 64-bit user programs. This software worked its way down the product line. A few customers quickly bought more than 4 GB of memory and within a day had recompiled programs to use it, in some cases merely by changing one Fortran parameter. Low-end SGI workstations, however, continued to ship with a 32-bit-only operating system for years, and of course, existing 32-bit hardware had to be supported...for years. For historical reasons, SGI had more flavors of 32-bit and 64-bit instruction sets than were really desirable, so it was worse than having just two of them.

This is the bad kind of "long tail"people focus on "first ship dates," but often the "last ship date" matters more, as does the "last date on which we will release a new version of an operating system or application that can run on that system." Windows 16-bit applications still run on regular Windows XP, 20years after the 80386 was introduced. Such support has finally been dropped in Windows XP x64.

DEC. DEC shipped 64-bit Alpha systems in late 1992, with a 64-bit operating system, and by late 1994 was shipping servers with memories large enough to need greater than 32-bit addressing. DEC might have requested (easy) 32-bit ports, but thinking long term, it went straight to 64-bit, avoiding duplication. It was expensive in time and money to get third-party software 64-bit clean, but it was valuable to the industry as it accelerated the 64-bit cleanup. DEC was probably right to do this, since it had no installed base of 32-bit Alpha programs and could avoid having to support two modes. For VMS, early versions were 32-bit, and later ones 64/32-bit.

Other vendors. Over the next few years, many vendors shipped 64-bit CPUs, usually running 32-bit software, and later 64/32-bit: Sun UltraSPARC (1995), HAL SPARC64 (1995), PA-RISC (1996), HP/UX 11.0 (1997), IBM RS64 and AIX 4.3 (1997), Sun Solaris 7 (1998), IBM zSeries (2001), Intel Itanium (2001), AMD AMD64 (2003), Intel EM-T64a (2004), Microsoft Windows XP x64 (2005). Linux 64-bit versions appeared at various times.

Most 64-bit CPUs were designed as extensions of existing 32-bit architectures that could run existing 32-bit binaries well, usually by extending 32-bit registers to 64 bits in 64-bit mode, but ignoring the extra bits in 32-bit mode. The long time span for these releases arises from natural differences in priorities. SGI was especially interested in high-performance technical computing, whose users were accustomed to 64-bit supercomputers and could often use 64 bits simply by increasing one array dimension in a Fortran program and recompiling. SGI and other vendors of large servers also cared about memory for large database applications. It was

certainly less important to X86 CPU vendors whose volume was dominated by PCs. In Intel's case, perhaps the emphasis on Itanium delayed 64-bit X86s.

By 2006, 4GB of DRAM, typically consisting of 1GB DRAMs, typically used four DIMMs and could cost less than $400, 300GB disks are widely available for less than $1 per GB, so one would have expected mature, widespread support for 64 bits by then. All this took longer than perhaps it should have, however, and there have been many years where people could buy memory but not be able to address it conveniently, or not be able to buy some third-party application that did, because such applications naturally lag 64-bit CPUs. It is worth understanding why and how this happened, even though the impending issue was well known.

Back to Top

Lessons

For any successful computer family, it takes a very long time to convert an installed base of hardware, and software lasts even longer.

Moving from 32 to 64/32 bits is a long-term coexistence scenario. Unlike past transitions, almost all 64-bit CPUs must run compatible existing 32-bit binaries without translation, since much of the installed base remains 32-bit and a significant number of 32-bit programs are perfectly adequate and can remain so indefinitely.

Back to Top

Software is Harder

Building a 64-bit CPU is not enough. Embedded-systems markets can move easier than general-purpose markets, as happened, for example, with Cisco routers and Nintendo N64s that used 64-bit MIPS chips. Vendors of most 32-bit systems, however, had to make their way through all of the following steps to produce useful upward-compatible 64-bit systems:

1. Ship systems with 64/32 CPUs, probably running in 32-bit mode. Continue supporting 32-bit-only CPUs as long as they are shipped and for years thereafter (often five or more years). Most vendors did this, simply because software takes time.
2. Choose a 64-bit programming model for C, C++, and other languages. This involves discussion with standards bodies and consultation with competitors. There maybe serious consequences if you select a different model from most of your competitors. Unix vendors and Microsoft did choose differently, for plausible reasons. Think hard about inter-language issuesFortran expects INTEGER and REAL to be the same size, which makes 64-bit default integers awkward.
3. Clean up header files, carefully.
4. Build compilers to generate 64-bit code. The compilers themselves almost certainly run in 32-bit mode and cross-compile to 64-bit, although occasional huge machine-generated programs can demand compilers that run in 64-bit mode. Note that programmer sanity normally requires a bootstrap step here, in which the 32-bit compiler is first modified to accept 64-bit integers and then is recoded to use them itself.
5. Convert the operating system to 64-bit, but with 32-bit interfaces as well, to run both 64- and 32-bit applications.
6. Create 64-bitversions of all system libraries.
7. Ship the new operating system and compilers on new 64-bit hardware, and hopefully, on the earlier 64-bit hardware that has now been shipping for awhile. This includes supporting (at least) two flavors of every library.
8. Talk third-party software vendors into supporting a 64-bit version of any program for which this is relevant. Early in such a process, the installed base inevitably remains mostly 32-bit, and software vendors consider the potential market size versus the cost of supporting two versions on the same platform. DEC helped the industry fix 32- to 64-bit portability issues by paying for numerous 64-bit Alpha ports.
9. Stop shipping 32-bit systems (but continue to support them for many years).
10. Stop supporting 32-bit hardware with new operating system releases, finally.
11. Going from step 1 to step 6 typically took two to three years, and getting to step 9 took several more years. The industry has not yet completed step 10.

Operating system vendors can avoid step 1, but otherwise, the issues are similar. Many programs need never be converted to 64-bit, especially since many operating systems already support 64-bit file pointers for 32-bit programs.

Next, I trace some of the twists and turns that occurred in the 1990s, especially involving the implementation of C on 64/32-bit CPUs. This topic generated endless and sometimes vituperative discussions.

Back to Top

C: 64-bit Integers on 64/32-bit CPUs: Technology and Politics

People have used various (and not always consistent) notations to describe choices of C data types. In the accompanying table, the first label of several was the most common, as far as I can tell. On machines with 8-bit char, short is usually 16 bits, but other data types can vary. The common choices are shown in the table.

Early days. Early C integers (1973) included only `int` and `char`; then `long` and `short` were added by 1976, followed by `unsigned` and `typedef` in 1977. In the late 1970s, the installed base of 16-bit PDP-11s was joined by newer 32-bit systems, requiring that source code be efficient, sharable, and compatible between 16-bit systems (using I16LP32) and 32-bitters (ILP32), a pairing that worked well. PDP-11s still employed (efficient) 16-bit `int` most of the time, but could use 32-bit long as needed. The 32-bitters used 32-bit `int` most of the time, which was more efficient, but could express 16-bit via `short`. Data structures used to communicate among machines avoided `int`. It was very important that 32-bit `long` be usable on the PDP-11. Before that, the PDP-11 needed explicit functions to manipulate `int` [2] (16-bit `int` pairs), and such code was not only awkward, but also not simply sharable with 32-bit systems. This is an extremely important point`long` was not strictly necessary for 32-bit CPUs, but it was very important to enable code sharing among 16- and 32-bit environments. We could have gotten by with `char, short`, and `int`, if all our systems had been 32 bits.

It is important to remember the nature of C at this point. It took a while for `typedef` to become common idiom. With 20/20 hindsight, it might have been wise to have provided a standard set of `typedefs` to express "fast integer," "guaranteed to be exactly N-bit integer," "integer large enough to hold a pointer," and to have recommended that people build their own typedefs on these definitions, rather than base types. If this had been done, perhaps much toil and trouble could have been avoided.

This would have been very counter-cultural, however, and it would have required astonishing precognition. Bell Labs already ran C on 36-bit CPUs and was working hard on portability, so overly specific constructs such as "intl6" would have been viewed with disfavor. C compilers still had to run on 64KI+64KD PDP-11s, so language minimality was prized. The C/Unix community was relatively small (600 systems) and was just starting to adapt to the coming 32-bit minicomputers. In late 1977, the largest known Unix installation had seven PDP-11s, with a grand total of 3.3MB of memory and 1.9GB of disk space. No one could have guessed how pervasive C and its offshoots would become, and thinking about 64-bit CPUs was not high on the list of issues.

32-bit happy times. In the 1980s, ILP32 became the norm, at least in Unix-based systems. These were happy times: 32-bit was comfortable enough for buyable DRAM, for many years. In retrospect, however, it may have caused some people to get sloppy in assuming `sizeof(int) == sizeof(long) == sizeof(ptr) == 32`.

Sometime around 1984, Amdahl UTS and Convex added `long long` for 64-bit integers, the former on a 32-bit architecture, the latter on a 64-bitter. UTS used this especially for long file pointers, one of the same motivations for long in PDP-11 Unix (1977). Algol 68 inspired long long in 1968, and it was also added to GNU C at some point. Many reviled this syntax, but at least it consumed no more reserved keywords.

Of course, 16-bit `int` was used on Microsoft DOS and Apple Macintosh systems, given the original use of Intel 8086 or MC68000, where 32-bit `int` would have been costly, particularly on early systems with 8- or 16-bit data paths and where low memory cost was especially important.

64-bit heats up inl991/1992. The MIPS R4000 and DEC Alpha were announced in the early 1990s. Email discussions were rampant among various companies during 19901992 regarding the proper model for 64-bit C, especially when implemented on systems that would still run 32-bit applications for many years. Quite often, such informal cooperation exists among engineers working for otherwise fierce competitors.

In mid-1992 Steve Chessin of Sun initiated an informal 64-bit C working group to see if vendors could avoid implementing randomly different 64-bit C data models and nomenclatures. Systems and operating system vendors all feared the wrath of customers and third-party software vendors otherwise. DEC had chosen LP64 and was already far along, as Alpha had no 32-bit version. SGI was shipping 64-bit hardware and working on 64-bit compilers and operating system; it preferred LP64 as well. Many others were planning 64-bit CPUs or operating systems and doing portability studies.

Chessin's working group had no formal status, but had well-respected senior technologists from many systems and operating systems vendors, including several who were members of the C Standards Committee. With all this brainpower, one might hope that one clear answer would emerge, but that was not to be. Each of the three proposals (LLP64, LP64, and ILP64) broke different kinds of code, based on the particular implicit assumptions made in the 32-bit happy times.

Respected members of the group made credible presentations citing massive analyses of code and porting experience, each concluding, *"XXX* is the answer." Unfortunately, *XXX* was different in each case, and the group remained split three ways. At that point I suggested we perhaps could agree on header files that would help programmers survive (leading to `<inttypes.h>`). Most people did agree that `long long` was the least bad of the alternative notations and had some previous usage.

We worried that we were years ahead of the forthcoming C standard, but could not wait for it, and the C Standards Committee members were supportive. If we agreed on anything reasonable, and it became widespread practice, it would at least receive due consideration. Like it or not, at that point this unofficial group probably made `long long` inevitableor perhaps that inevitability dated from 1984.

By 1994, DEC was shipping large systems and had paid for many third-party software ports, using LP64. SGI was also shipping large systems, which supported both ILP32LL and LP64, with `long long` filling the role

handled by long in the late 1970s.

The DEC effort proved that it was feasible to make much software 64-bit-clean without making it 32-bit unclean. The SGI effort proved that 32-bit and 64-bit programs could be sensibly supported on one system, with reasonable data interchange, a requirement for most other vendors. In practice, that meant that one should avoid `long` in structures used to interchange data, exactly akin to the avoidance of `int` in the PDP-11/VAX days. About this time in 1995 the Large File Summit agreed on Unix APIs to increase file size above 2GB, using `long long` as a base data type.

Finally, the Aspen Group in 1995 had another round of discussions about the 64-bit C model for Unix and settled on LP64, at least in part because it had been proved to work and most actual 64-bit software used LP64.

During the 1992 meetings of the 64-bit C group Microsoft had not yet chosen its model, but later chose LLP64, not the LP64 preferred by Unix vendors. I was told that this happened because the only 32-bit integer type in PCs was `long`; hence, people often used `long` to mean 32-bitmore explicitly than in Unix code. That meant that changing its size would tend to break more code than in Unix. This seemed plausible to me. Every choice broke some codes, and thus people looked at their own code bases, which differed, leading to reasonable differences of opinion.

Many people despised `long long` and filled newsgroups with arguments against it, even after it became incorporated into the next C standard in 1999. The official rationale for the C standard can be consulted by anyone who wants to understand the gory details.[6]

Differing implicit assumptions about sizes of various data types had grown up over the years and caused a great deal of confusion. If we could go back to 1977, knowing what we know now, we could have made all this easier simply by insisting on more consistent use of `typedef`. In 1977, however, it would have been difficult to think ahead 20 years to 64-bit microswe were just getting to 32-bit minis!

This process might also have been eased if more vendors had been further along in 64-bit implementations in 1992. Many people had either forgotten the lessons of the PDP-11/VAX era or had not been involved then. In any case, the 64/32 systems had a more stringent requirement: 64-bit and 32-bit programs would coexist forever in the same systems.

Timing is important to creating good industry standards. Too early, standards may be written with insufficient practical experience because no one has real implementations from which to learn. Too late, and vendors may well have committed to numerous incompatible implementations. In between, a few groups have early implementations to help inform the standards. Perhaps by accident, the 64-bit C standards were reasonably well-timed.

Back to Top

Lessons

Standards often get created in non-standard ways, and often, de facto standards long precede official ones.

Reasonable people can disagree, especially when looking at different sets of data.

Sometimes one must work with competitors to make anything reasonable happen.

Programmers take advantage of extra bits or ambiguity of specification. Most of the arguments happen because application programmers make differing implicit assumptions.

Code can be recompiled, but once data gets written somewhere, any new code must still be able to describe it cleanly. Current software is rarely done from scratch but has to exist inside a large ecosystem.

We might never build 128-bit computers, but it would probably be good to invent a notation for 128-bit integers, whose generated code on 64-bit CPUs is about the same as 64-bit code is on 32-bit CPUs. It would be nice to do that long before it is really needed. In general, predictable long-term problems are most efficiently solved with a little planning, not with frenzied efforts when the problem is imminent. Fortunately, 128-bitters are many years away, if ever (maybe 20202040), because we've just multiplied our addressing size by four billion, and that will last a while, even if Moore's Law continues that long! In case 128-bit happens in 2020, however, it would be wise to be thinking about the next integer size around 2010.
Of course, when the S/360 was introduced, IBM and other vendors had 36-and 18-bitproductlines. In an alternate universe, had the S/360 been a 36-bit architecture with four 9-bit bytes/word, most later machines would have been 18- and 36-bit, and we would just be starting the 36-bit to 72-bit transition.

Hardware decisions last a long time, but software decisions may well last longer. If you are a practicing programmer, take pity on those who end up maintaining your code, and spend some time thinking ahead. Allow for natural expansion of hardware, hide low-level details, and use the highest-level languages you can. C's ability to make efficient use of hardware can be both a blessing and a curse.

Back to Top

Conclusion

Some decisions last a very long time. The 24-bit addressing of 1964's S/360 is still with us, as are some side effects of C usage in the mid-1970s. The transition to 64-bit probably took longer than it needed for a host of

reasons. It's too bad that people quite often have been unable to use affordable memory for solving performance problems or avoiding cumbersome programming.

It's too bad there has been so much "toil and trouble," but "double" for microprocessors has been accomplished, and "double" for software is at least under way, and people have stopped arguing about the need for 64-bit micros.

Back to Top

References

1. Aspen Data Model Committee. 64-bit programming models: Why LP64? (19971998); www.unix.org/version2/whatsnew/lp64_wp.html.

2. Bell, C.G. and Mudge, J.C. The evolution of the PDP-11, *Computer Engineering: A DEC View of Computer System Design*. CG. Bell, J.C. Mudge, and J.E. McNamara, Eds. Digital Press, Bedford, MA, 1978,

3. Josey, A. Data size neutrality and 64-bit support (1997); www.usenix.org/publications/login/standards/10.data.html.

4. Mashey, J. Languages, Levels, Libraries, Longevity. *ACM Queue* 2, 9 (20042005), 3238.

5. Mashey, J. 64-bit computing. *BYTE* (Sept. 1991), 135142. 9 (The compLete text can also be found by searching Google Groups comp.arch: Mashey BYTE 1991).

6. Rationale for International· StandardProgramming LanguagesC; www.open-std.org/jtcl/sc22/wg14/www/docs/n897.pdf (or other sites).

7. Strecker, W.D. VAX-11/780: A virtual· address extension to the DEC PDP-11 Family. *Computer Engineering: A DEC View of Computer System Design*. CG. Bell J.C. Mudge, and J.E. McNamara, Eds. Digital Press, Bedford, MA, 1978.

8. Unix specification; adding support for arbitrary file sizes; www.unix.org/version2/whatsnew/Lfs20mar.html.

Back to Top

Author

John Mashey is a consultant for venture capitalists and technology companies, sits on various technology advisory boards of startup companies, and is a trustee of the Computer History Museum. Along with spending 10 years at Bell Laboratories working on the Programmer's Workbench version of Unix, he was also one of the designers of the MIPS RISC architecture, one of the founders of the SPEC benchmarking group, and chief scientist at Silicon Graphics.

Back to Top

Footnotes

A previous version of this article appeared in the October 2006 issue of ACM Queue.

DOI: http://doi.acm.org/10.1145/1435417.1435431

Back to Top

Tables

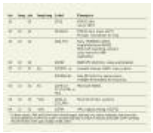


Table. Common C data types.

Back to top