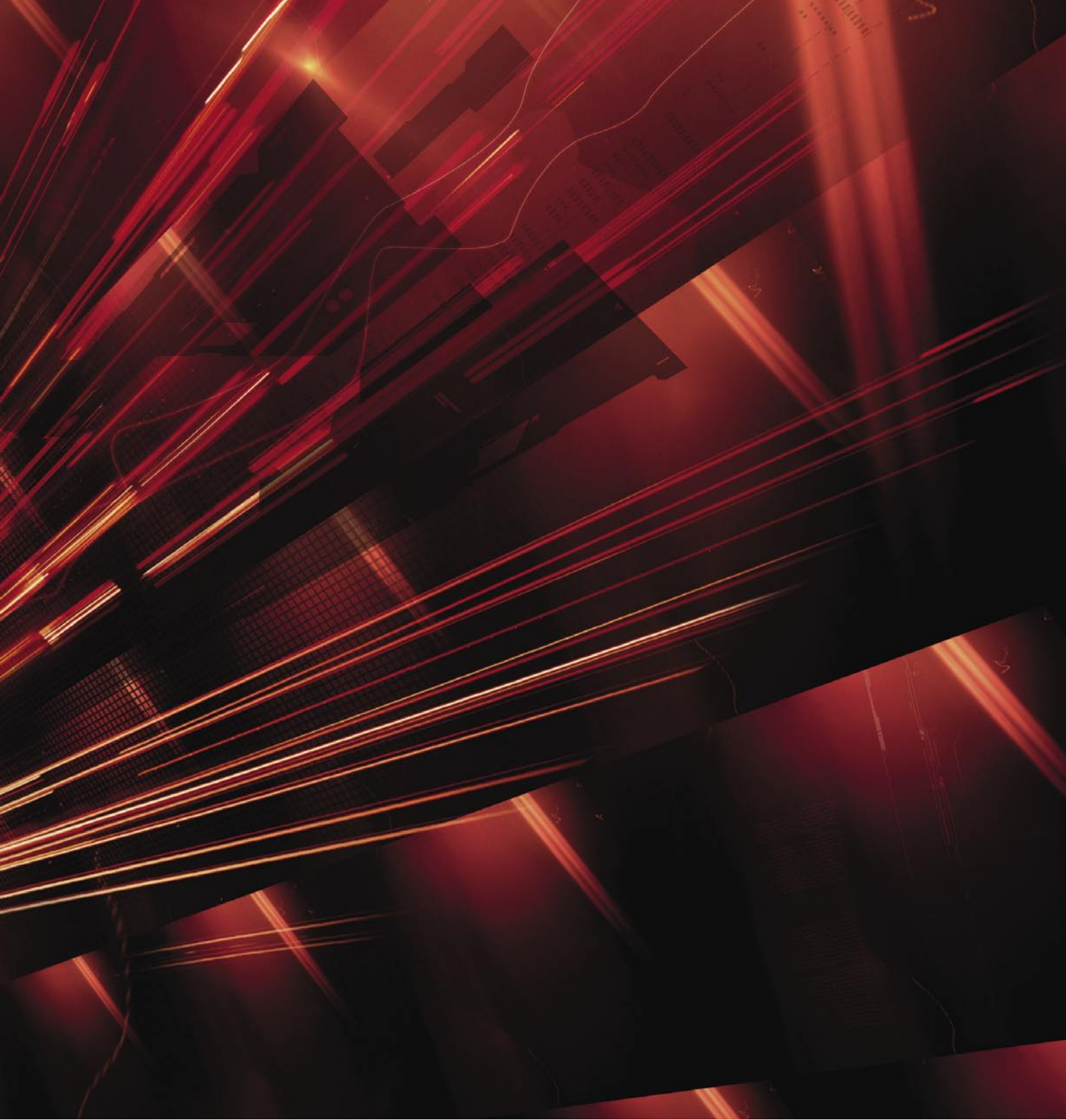


# THE LONG ROAD TO 64 BITS

JOHN R. MASHEY, TECHVISER

Double, double, toil and trouble—Shakespeare, *Macbeth*



Shakespeare's words (*Macbeth*, Act 4, Scene 1) often cover circumstances beyond his wildest dreams. *Toil and trouble* accompany major computing transitions, even when people plan ahead. To calibrate "tomorrow's legacy today," we should study "tomorrow's legacy yesterday." 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-bit systems. 32-bit systems 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,

# THE LONG ROAD TO 64 BITS

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.

## FUNDAMENTAL PROBLEM (LATE 1980S)

Running out of address space is a long tradition in computing, and often quite predictable. Moore's law increased the size of DRAM approximately four times every three to four years, and by the mid-1990s, people were able to afford 2 to 4 GB of

memory for midrange microprocessor systems, at which point simple 32-bit addressing (4 GB) 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 the following:

- Some unfortunate decisions may be driven by real constraints (1970: PDP-11 16-bit).
- Reasonable-at-the-time decisions turn out in 20-year retrospect to have been suboptimal (1976-77: 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 transitioning to 64/32 bits).

**Constraints.** Hardware people needed to build 64/32-bit CPUs at the right time—neither 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 expensive—the first commercial 64-bit micro's 64-bit data path added at most 5 percent 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?"<sup>1</sup>

We managed to survive the Y2K problem—with 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.

## 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, seg-



mented 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 micros—for 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 it 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.<sup>2</sup> 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 you need a 2N-bit operation (add, multiply, etc.), each can be done in one instruction on a 2N-bit CPU, but longer sequences are required 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.

**Addressing—virtual and real—in 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 2 GB. 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  $PM \leq VL$ .

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,  $VL \geq RL \geq AM$ . Some programs burn virtual address space for convenience and actually perform acceptably when  $PM \gg AM$ : I've seen cases where 4:1 still worked, as a result of good locality. File mapping

# MORE

## Related articles in ACM's Digital Library:

Bell, G., Strecker, W. D. 1998. Retrospective: what have we learned from the PDP-11—what we have learned from VAX and Alpha. In *25 Years of the International Symposia on Computer Architecture (selected papers)* (August).

Coady, Y., Kiczales, G. 2003. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development* (March).

Gifford, D., Spector, A. 1987. Case study: IBM's system/360-370 architecture. *Communications of the ACM* 30(4).

Van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N. 2001. Taming architectural evolution. *ACM SIGSOFT Software Engineering Notes* 26(5).

These articles will be available online at [www.acmqueue.com](http://www.acmqueue.com) for an eight-week period beginning Oct. 1. Want full access? Join ACM today at [www.acm.org](http://www.acm.org) and sign up for the Digital Library.

# THE LONG ROAD TO 64 BITS

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  $VL \geq RL \geq AM$ , and then  $AM$  grows, and perhaps  $RL$  is increased in ways visible only to the operating system, at which point  $VL \ll AM$ . 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 4 MB, but people would like the same code to run for  $M=1,000$  (4 GB), or 6,300 (1,000 GB). A few such systems do exist, although they are not cheap. I once had a customer complain about lack of current support for 1,000 GB 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 10,000-GB 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, etc.

Running out of address bits has a long history.

MAINFRAMES, MINICOMPUTERS, MICROPROCESSORS  
The oft-quoted George Santayana is appropriate here: “Those who cannot 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 16 MB of core memory. This was considered immense at the time. A “large” mainframe offered at most 1 MB of memory, although a few “huge” mainframes could provide 6 MB. 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 16-MB 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 16-MB limit grew inadequate. IBM 370-XA 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 2006—in 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 2-GB 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 extra headaches?

**Minicomputers.** In the 16-bit DEC PDP-11 mini-computer family (1970), a single task addressed only 64 KB, or in later models (1973), 64 KB of instructions plus 64 KB 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.”<sup>3</sup>

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



## Chronology: Multiple Interlocking Threads

- 1964 **IBM S/360**: 32-bit, with 24-bit addressing (16 MB total) of real (core) memory.
- 1968 **Algol 68**: includes long long.
- 1970 **DEC PDP-11/20**: 16-bit, 16-bit addressing (64 KB total).  
**IBM S/370 family**: virtual memory, 24-bit addresses, but multiple user address spaces allowed.
- 1971 **IBM 370/145**: main memory no longer core, but DRAM, 1 Kbit/chip.
- 1973 **DEC PDP-11/45**: separate instruction+data (64 KI + 64 KD); 248 KB maximum real memory.  
**Unix**: PDP-11/45, operating system rewritten in C; IP16.  
**C**: integer data types: int, char; C on other machines (36-bit Honeywell 6000, IBM 370, others).
- 1975 **Unix**: sixth edition, 24-bit maximum file size (16 MB).
- 1976 **DEC PDP-11/70**: (64 KI + 64 KD), but larger physical memory (a huge 4 MB).  
**C**: short, long added (partly from doing C for XDS Sigma, although long was 64 bits there).
- 1977 **Unix**: ported to 32-bit Interdata 8/32.  
**C**: unsigned, typedef, union; 32-bit long used to replace int[2] in lseek, tell on 16-bit PDP-11; IP16L32.  
**DEC VAX-11/780**: 32-bit, 32-bit addressing (4 GB total, 2 GB per user process).  
**C**: **PDP-11**: I16LP32; **VAX** (other 32-biters): ILP32.
- 1978 **Unix**: 32V for VAX-11/780; C is ILP32.  
**C**: *The C Programming Language*, Brian Kernighan and Dennis Ritchie (Prentice-Hall).  
**Intel 8086**: 16-bit, but with user-visible segmentation.
- 1979 **Motorola MC68000**: 32-bit ISA, but 24-bit addressing (e.g., S/360).
- 1982 **C**: I16LP32 on MC68000 in Bell Labs Blit terminal.  
**Intel 80286**: allows 16 MB of real memory, but restrictions keep most systems at 1 MB.
- 1983 **IBM 370/XA**: adds 31-bit mode for user programs; 24-bit mode still supported.  
**C**: Unix workstations generally use ILP32, following Unix on VAX systems.
- 1984 **Motorola MC68020**: 32-bit; 32-bit addressing.  
**C**: Amdahl UTS (32-bit S/370) uses long long, especially for large file pointers.  
**C**: Convex (64-bit vector mini-supercomputer) uses long long for 64-bit integers.
- 1986 **Intel**: 80386, 32-bit, with support for 8086 mode.
- 1987 **Apple Mac II**: MC68020's 32-bit addressing causes trouble for some MC68000 software.
- 1988 **IBM ESA/370**: multiple 31-bit address spaces per user, although complex; 24-bit still there.
- 1989 **ANSI C ("C89")**: effort had started in 1983, ANSI X3J11.
- 1992 **SGI**: ships first 64-bit micro (MIPS R4000); still running 32-bit operating system.  
**64-bit C working group**: discusses various models (LP64, ILP64, LLP64), with little agreement.  
**DEC**: ships 64-bit Alpha systems, running 64-bit operating system; LP64.
- 1994 **SGI**: ships IRIX 6 (64/32 operating system; ILP32LL + LP64) on Power Challenge; customers buy 4 GB+ memory, use it.  
**DEC**: ships 4 GB+ in DEC 7000 SMPs (may have been slightly earlier).
- 1995 **Sun UltraSPARC**: 64/32-bit hardware, 32-bit-only operating system.  
**HAL Computer's SPARC64**: uses ILP64 model for C.  
**Large file summit**: codifies 64-bit interface to files >2 GB, even in 32-bit systems (ILP32LL+LP64).  
**Aspen group**: supports LP64 model for C so that Unix vendors are consistent.
- 1996 **HP**: announces PA-RISC 2.0, 64-bit.
- 1997 **HP**: UP/UX 11.0 is 64/32-bit OS; ILP32LL + LP64.  
**IBM**: RS64 PowerPC, AIX 4.3; ILP32LL + LP64.
- 1998 **Sun**: 64/32 Solaris 7 released; ILP32LL + LP64.
- 1999 **C**: ISO/IEC C (WG14's "C99"); includes long long, at least 64 bits.
- 2001 **IBM**: 64-bit zSeries (S/360 descendant); 24-bit addressing still supported.  
**Intel**: 64-bit Itanium.
- 2002 **Microsoft**: Windows 64-bit for Itanium.
- 2003 **AMD**: 64-bit X86 (now called AMD64).
- 2004 **Intel**: 64-bit X86 (called EMT64), compatible with AMD.
- 2005 **Microsoft**: Windows XP Professional x64 for X86; LLP64 (or IL32LLP64).

# THE LONG ROAD TO 64 BITS

would fall. In any case, this lasted a long time—the 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 use only 64 KI (instruction) + 64 KD (data) of a maximum of 4 MB, so that individual large programs required unnatural acts to split code and data into 64 KB 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 65-KB address space, and perhaps more importantly, others whose programming is appreciably simplified by having a large address space.”<sup>4</sup>

**Microprocessors.** The Intel 8086’s 16-bit ISA seemed likely to fall prey to the PDP-11’s issues (1978). It did provide user-mode mechanisms for explicit segment manipulation, however. This allowed a single program to access more than 64 KB 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 (1987).

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.

## 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.

## THE 32- TO 64-BIT PROBLEM IN THE LATE 1980S

By the late ’80s, Moore’s law seemed cast in silicon, and it was clear that by 1993-94, midrange microprocessor servers could cost-effectively offer 2-4 GB 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 1993-94, and real trouble by 1995. As I wrote in *BYTE* in September 1991:<sup>5</sup>

“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 16-MB DRAM chips, but increasing physical memory slightly beyond 32 bits is definitely a goal. With 16-MB 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 actually 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 20 years 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 clean-up. 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 EMT64a (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.

In 2006, 4 GB of DRAM, consisting of 1-GB DRAMs, typically uses four DIMMs and can cost less than \$400. At least some large notebooks support 4 GB, and 300-GB disks are widely available for less than \$1 per gigabyte, so one would expect mature, widespread support for 64 bits by now. 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 surely well known.

#### 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



# THE LONG ROAD TO 64 BITS

translation, since much of the installed base remains 32-bit, and many 32-bit programs are perfectly adequate and can remain so indefinitely.

**SOFTWARE IS HARDER: OPERATING SYSTEM, COMPILERS, APPLICATIONS, USERS, STANDARDS** Building a 64-bit CPU is not enough. Embedded-systems markets can move more easily 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-bit 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 may be 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 issues—Fortran 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-bit versions 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 a while. 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.

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.

The next section traces 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. Interested readers should consult references 6-9. Masochistic readers might search newsgroup comp.std.c for “64-bit” or “long long”.

## 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 table 1, 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 table 1.

**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-bit systems (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-bit systems 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 a 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,” etc., 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 countercultural, 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 “`int16`” would have been viewed with disfavor. C compilers still had to run on 64 KI + 64 KD 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 3.3 MB of memory and 1.9 GB 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-bit. 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 in 1991/1992.** The MIPS R4000 and DEC Alpha were announced in the early 1990s. E-mail discussions were rampant among various companies during 1990-92 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

# TABLE 1

Common C Data Types

int	long	ptr	long long	Label	Examples
16	--	16	--	IP16	PDP-11 Unix (early, 1973)
16	32	16	--	IP16L32	PDP-11 Unix (later, 1977); multiple instructions for long
16	32	32	--	I16LP32	Early MC68000 (1982); Apple Macintosh 68K Microsoft operating systems (plus extras for X86 segments)
32	32	32	--	ILP32	IBM 370; VAX Unix; many workstations
32	32	32	64	ILP32LL or ILP32LL64	Amdahl; Convex; 1990s Unix systems Like IP16L32, for same reason; multiple instructions for long long.
32	32	64	64	LLP64 or IL32LLP64 or P64	Microsoft Win64
32	64	64	*(64)	LP64 or I32LP64	Most 64/32 Unix systems
64	64	64	*(64)	ILP64	HAL; logical analog of ILP32

*\*In these cases, LP64 and ILP64 offer 64-bit integers, and long long seems redundant, but in practice, most proponents of LP64 and ILP64 included long long as well, for reasons given later. ILP64 uniquely required a new 32-bit type, usually called `_int32`.*

# THE LONG ROAD TO 64 BITS

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 system 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` inevitable—or 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 2 GB, 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-bit more explicitly than in Unix code. That meant that changing its size would tend to break more code than in Unix. This seemed plausible. Every choice broke some code; 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.<sup>6</sup>

Differing implicit assumptions about the 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 hard to think ahead 20 years to 64-bit micros—we 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.



## LESSONS

- Standards often get created in nonstandard ways, and often, de facto standards long precede official ones.
- Reasonable people can disagree, especially when looking at different sets of data.
- Sometimes one has to work with competitors to make anything reasonable happen.
- Programmers take advantage of extra bits or ambiguity of specification. Most 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-biters are many years away, if ever (maybe 2020-2040), because we've just multiplied our addressing size by 4 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-bit product lines. 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.

## CONCLUSIONS

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 at least people have stopped arguing about whether there should be 64-bit micros or not. ☹

## REFERENCES

1. Mashey, J. 2004-05. Languages, levels, libraries, longevity. *ACM Queue* 2 (9): 32-38.
2. Mashey, J. 1991. 64-bit computing. *BYTE* (September): 135-142. The complete text can be found by searching Google Groups comp.arch: Mashey BYTE 1991.
3. Bell, C. G., Mudge, J. C. 1978. The evolution of the PDP-11. In *Computer Engineering: A DEC View of Computer System Design*, ed. C. Gordon Bell, J. Craig Mudge, and John E. McNamara. Bedford, MA: Digital Press.
4. Strecker, W. D. 1978. VAX-11/780: A virtual address extension to the DEC PDP-11 family. In *Computer Engineering: A DEC View of Computer System Design*, ed. C. Gordon Bell, J. Craig Mudge, and John E. McNamara. Bedford, MA: Digital Press.
5. See reference 2.
6. Rationale for International Standard—Programming Languages—C; <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf> (or other sites).
7. Aspen Data Model Committee. 1997-1998. 64-bit programming models: Why LP64? [http://www.unix.org/version2/whatsnew/lp64\\_wp.html](http://www.unix.org/version2/whatsnew/lp64_wp.html).
8. Josey, A. 1997. Data size neutrality and 64-bit support; <http://www.usenix.org/publications/login/standards/10.data.html>.
9. Adding support for arbitrary file sizes to the single Unix specification; <http://www.unix.org/version2/whatsnew/lfs20mar.html>.

## LOVE IT, HATE IT? LET US KNOW

[feedback@acmqueue.com](mailto:feedback@acmqueue.com) or [www.acmqueue.com/forums](http://www.acmqueue.com/forums)

**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. He spent 10 years at Bell Laboratories, working on the Programmer's Workbench version of Unix. He later managed development of Unix-based applications and software development systems. After Bell Labs, he was one of the designers of the MIPS RISC architecture, one of the founders of the SPEC benchmarking group, and chief scientist at Silicon Graphics. His interests have long included interactions between hardware and software. He received a B.S. in mathematics, and M.S. and Ph.D. in computer science from Pennsylvania State University.

© 2006 ACM 1542-7730/06/1000 \$5.00