

# Re: How to develop a random number generation device

---

*Source:* <http://sci.tech-archive.net/Archive/sci.electronics.design/2007-09/msg03439.html>

---

- *From:* David Brown <david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
  - *Date:* Tue, 18 Sep 2007 18:02:34 +0200
- 

John Larkin wrote:

On Mon, 17 Sep 2007 23:17:33 +0200, David Brown  
<david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

John Larkin wrote:

On Mon, 17 Sep 2007 14:54:38 -0500, Vladimir  
Vassilevsky  
<antispam\_bogus@xxxxxxxxxxx> wrote:

John Larkin wrote:

My  
point  
is  
that  
large  
numbers  
of  
CPU  
cores  
\*will\*  
become  
common  
and  
cheap,  
and  
we  
need  
a  
new  
type  
of  
OS

Re: How to develop a random number generation device

to  
take  
advantage  
of  
this  
new  
reality.  
Done  
right,  
it  
could  
be  
simple  
and  
astoundingly  
secure  
and  
reliable.

Dear John,

Try developing a perfect OS of your own. I did. That was a very enlightening experience of why certain things have to be done by the certain ways. Particular questions welcome.

I did write three RTOS's, one for the 6800, one for the PDP-11, one for the LSI-11. As far as I know, they were perfect, in that they ran damned fast and had no bugs. The 6800 version included a token ring LAN thing, which I invented independently in about 1974.

Well, I remember 64-bit static rams, and 256-bit DRAMS. I can't see any reason we couldn't have 256 or 1024 cpu's on a chip, especially if a lot of them are simple integer RISC machines.

1024 CPUs = 1048576 software interfaces  
and a hell of the bus arbitration.

No worse a software interface than if each process was running on a single shared CPU; much less, in fact, since irrelevant interrupts,

Re: How to develop a random number generation device

swapping, and context switches aren't going on. Each process absolutely owns a CPU and only interacts with other processes when \*it\* needs to, probably through shared memory and semaphores.

A shared memory interface for 1024 cpus? That's going to be absolutely vast, or have terrible latency.

Why would a \*system\* care about the latency of one processor accessing memory? The system only cares about net performance. As it stands now, only one \*process\* can access memory at a time (because all processes share a single CPU) and they all suffer from context switching overhead. Multiple CPUs never context switch, so \*must\* be overall faster.

It is certainly true that it matters little if a process is delayed because it is swapped out of the cpu, or because the cpu it is running on has slow access to memory. But unless your new architecture is an improvement in speed (since it is unlikely to be more power efficient, or cheaper, and is not inherently more reliable), then there is no point in making it.

There is no reason to suppose your massively multiple core will be faster. Your shared memory will be a huge bottleneck in the system – rather than processes being blocked by limited cpu resources, they will be blocked by memory access.

You also seem to be under the impression that context switches are a major slowdown – in most cases, they are not significant. On server systems with two or four cores, processes typically get to run until they end or block for some other reason – context switches are a tiny fraction of the time spent. If you want a faster system serving more web pages or database queries, you have to boost the whole system – more I/O bandwidth, more memory bandwidth (this is why AMD's devices scale much better than Intel's), more memory, etc. Simply adding extra cpu cores will make little difference beyond the first few. For desktops, the key metric is the performance on a single thread – dual cores are only useful (at the moment) to make sure that processor-intensive threads are not interrupted by background tasks.

For almost every mainstream computing task, it is more efficient to use fewer processors running faster (although it is seldom worth getting the very fastest members of a particular cpu family) – you can get more work out of 2 cores at 2 GHz than 4 cores at 1 GHz. In a chip designed around many simple cores, each core is going to be a lot slower than a few optimised fast cores can be.

I still don't understand why you think that interrupts or context switches are a reliability issue – processors don't have problems with them.

It's the OS's that we have problems with. Hardware is cheap and reliable; software is expensive and buggy. So we should shift more of the burden to hardware.

Re: How to develop a random number generation device

## Re: How to develop a random number generation device

If you shift the complexity to hardware, you'd get hardware that is expensive and buggy.

Have you anything to back up this belief in cheap and reliable hardware? Certainly some hardware is cheap and reliable, being relatively simple – but the same applies to software.

And I'd love to hear you explain to customers that while their web server has a load average of a couple of percent, they need to buy a second processor chip just to run an extra cron job. A single cpu per process will *\*never\** be realistic.

The IBM Cell structure is a hint of the future.

The Cell is a specialised device – only the one "master" cpu can run general tasks. The eight smaller cpu's are only useful for specialised dedicated tasks (such as the graphics processing in games). This is precisely as I have described – massively multi-core devices exist, but they are only suitable for specialised tasks.

As far as bus arbitration goes, they all just share a central cache on the chip, with a single bus going out to dram. Cache coherence becomes trivial.

"Just share a central cache?" It might sound easy to you, but I suspect it would be *\*slightly\** more challenging to implement.

Sure, but Moore's Law keeps going, in spite of a pattern of people refusing to accept its implications.

Moore's Law is not like the law of gravity, you know. You can't quote it as "proof" that a simple solution to your shared cache problem will be developed!

I'd  
be  
happy

Re: How to develop a random number generation device

to  
waste  
a  
little  
silicon  
if  
I  
could  
have  
an  
OS  
that  
doesn't  
crash  
and  
that  
doesn't  
go  
to  
sleep  
for  
seconds  
at  
a  
time  
for  
no  
obvious  
reason.

The weak link is a developer. It is obviously more difficult to develop multicore stuff; hence it is a higher probability of flaws.

Putting a few hundred RISC cores on a chip, connecting to a central cache, is easy. You only have to get it right once. In our world, incredibly complex hardware just works, and modestly complex software is usually a bag of worms. Clearly we need the hardware to help the software.

You are too used to solid, reliable, \*simple\* cores like the cpu32. Complex hardware is like complex software – it \*is\* complex software, written in design languages then "compiled" to silicon. Like software, big and complex hardware has bugs.

## Re: How to develop a random number generation device

Far fewer than software bugs. Hardware design, parallel execution of state machines, is mature, solid stuff. Software just keeps bloating.

It's just a matter of the costs of finding and fixing errors in hardware are so much higher than for software, so that more effort goes into getting it right in the first place. But the result is that a given feature is vastly more expensive to develop in hardware than software. A hardware version of the Vista kernel may well be more reliable than the software version – but it would take several centuries to design, simulate and test, and cost millions per device.

Multiple  
cores gives  
absolutely  
no benefits  
in terms of  
reliability or  
stability –  
indeed, it  
opens all  
sorts of  
possibilities  
for  
hard-to-debug  
race  
conditions.

Especially if you remember about the  
50–page silicon erratas for pretty much any  
modern CPU.

Intel, maybe. Are any of the RISC machines that bad? But  
my PC doesn't  
have hardware problems, it has software problems.

Yes, many RISC machines have substantial errata. The more complex you  
make the design, the more bugs you get.

So let's make it simple, waste some billions of CMOS devices, and get  
computers that always work. We'll save a fortune.

## Re: How to develop a random number generation device

What you seem to be missing is that although the cores on your 1K cpu chip are simple (and can therefore be expected to be reliable, if designed well), they don't exist alone. If you want them to support general purpose computing tasks, rather than a massive SIMD system, then you have a huge infrastructure around them to feed them with instruction streams and data, and enormous complications trying to keep memory consistent.

They don't if you insist on running a copy of a bloated OS on each. A system designed, from scratch, to run on a pool of cheap CPUs could be incredibly reliable.

What do you think in particular would be better for a typical desktop applications?

Oh, 256 CPUs and, say, 32 FPUs should be plenty.

My desktop machine might well run more than 256 processes. How does that fit in your device? But most of the time, there are only 2 or 3 processes doing much work – often there will be 1 process which should run as fast as possible, as single-thread performance is the main bottleneck for desktop cpus.

My XP is currently running about 30 processes, with a browser, mail client, a PDF datasheet open, and Agent running. A number of them are not really necessary.

How many on yours?

At the moment, I've got 51 processes and a total of about 476 threads (it's the number of threads that's important here) on my XP-64 desktop, excluding any that task manager is not showing. There is a svchost.exe service with 80 threads on its own, and firefox has 24 threads.

John