A big LITTLE Problem: A Tale of big.LITTLE Gone Wrong

> *hey guys, I have a library project that I'm using in my android app w/ `gomobile`, there's a `SIGILL` crash on exynos (samsung), so I'm looking for `.o` and/or `.sym` for the compiled binary, #general says this is the place to pose this question—shipit, on Gophers Slack*

The night of October 24, this question appeared on the #darkarts channel in the Gophers Slack (a group chat of ~33k Go programmers—sign up here).

This question seemed straightforward at first. It seemed safe to assume that shipit simply made a bug or false assumption.

As it turned out, shipit's code was correct. No bug there.

The code that shipit's code used was correct too. No bug there.

Android's code was correct too. No bug there.

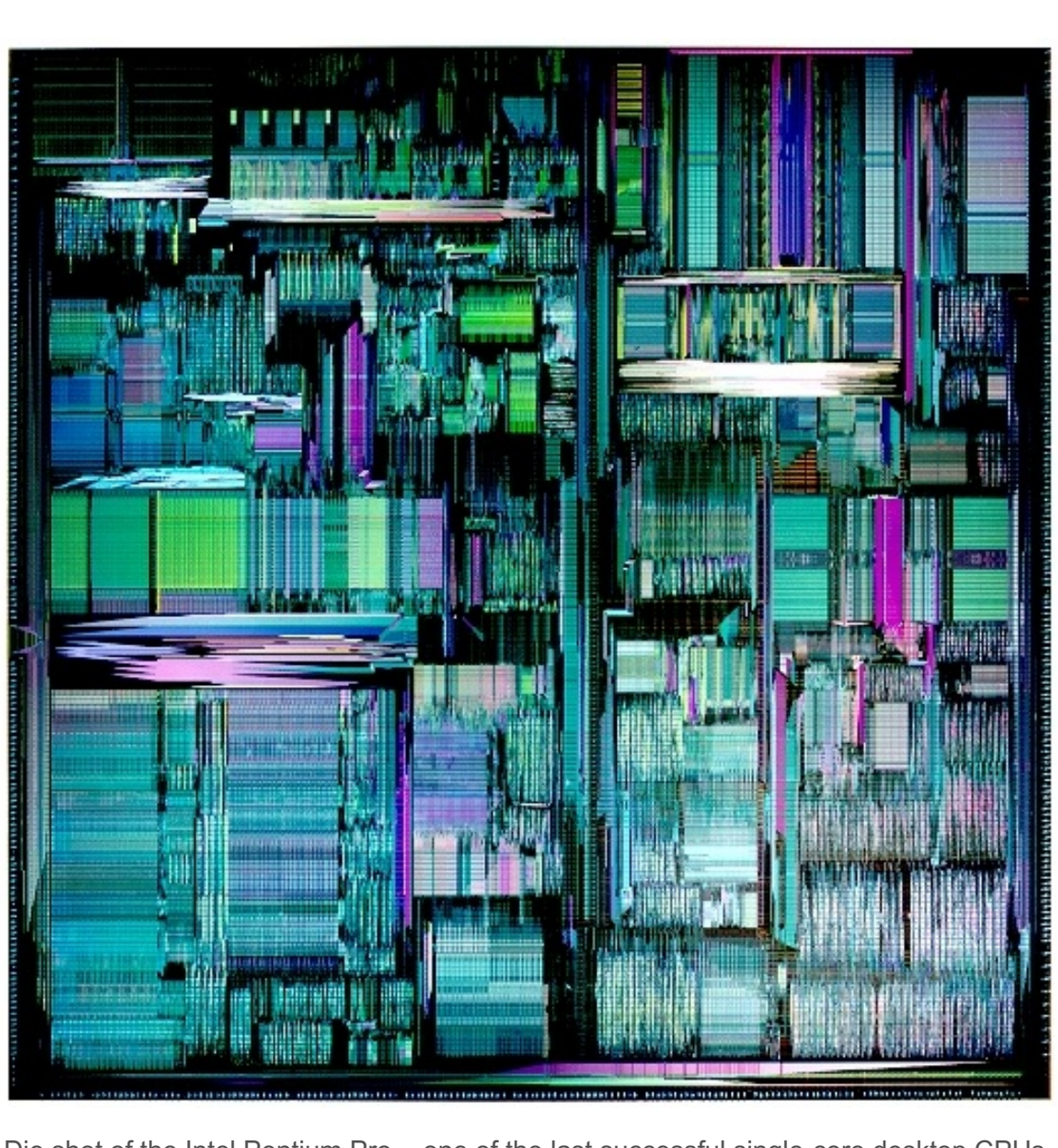
If all of the code is correct, then how could this be happening?

Sometimes, bugs occur in places you never consider. Usually, people will either blame the app or the operating system. This issue was far deeper.

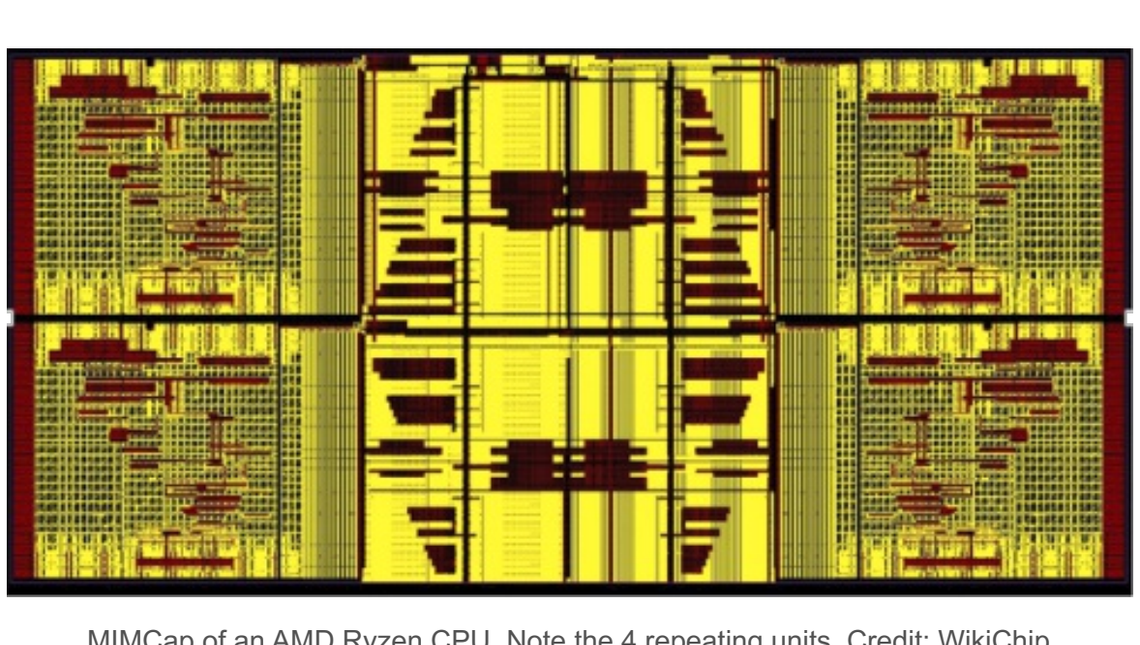
· · ·

Parallel Processing & big.LITTLE

The naïve explanation commonly used to explain how computers work is that they execute a stream of "instructions," each of which does some simple operation—addition, subtraction, etc. However, this explanation is very outdated. In the early 2000s, processor designers ran into limitations on how fast a single stream of instructions could be run—the faster they became, the less energy- and cost-efficient they became. Instead of continuing this pattern, processor designers moved the logic used to process a stream of instructions into a unit called a "core," and packed many of these "cores" into a single computer chip. Increasingly multitasking-focused systems could now run 8 tasks at a time instead of 1. Large tasks could also be broken into smaller parts, each part running on its own core.



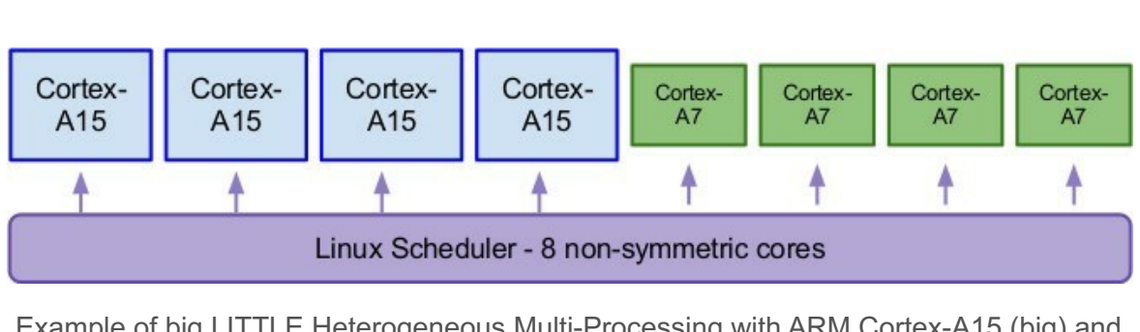
Die shot of the Intel Pentium Pro—one of the last successful single-core desktop CPUs (1995–2000). Credit: WikiChip



MIMCap of an AMD Ryzen CPU. Note the 4 repeating units. Credit: WikiChip

As the smartphone technology developed, ARM (the company that designs processors for smartphones, both Android and iPhone) realized that they were trying to optimize their core designs for 2 opposite kinds of tasks—light tasks (checking for notifications, updating the battery state indicator), and heavy tasks (intensive gaming, video playback). It was incredibly difficult to create a core design that both minimized power consumption on background tasks and maximized performance on mobile games.

ARM decided that instead of a one-size-fits-all design, building 2 kinds of cores would be the best way to solve this problem. Power-hungry "big" cores handle applications like games which demand speed at all costs, and low-power "LITTLE" cores handle light loads without draining the battery. This system is called "big.LITTLE" (and the modern variant of this is also called Heterogeneous Multi-Processing). It is transparent to applications—they start on a LITTLE core, and are moved to a big core as soon as the operating system detects that they are using a lot of processing power.



Example of big.LITTLE Heterogeneous Multi-Processing with ARM Cortex-A15 (big) and ARM Cortex-A7 (LITTLE). The A15 and A7 have been replaced by newer cores, but the concept remains the same. Credit: WikiMedia

Well, at least that is the theory. Nobody and nothing is perfect.

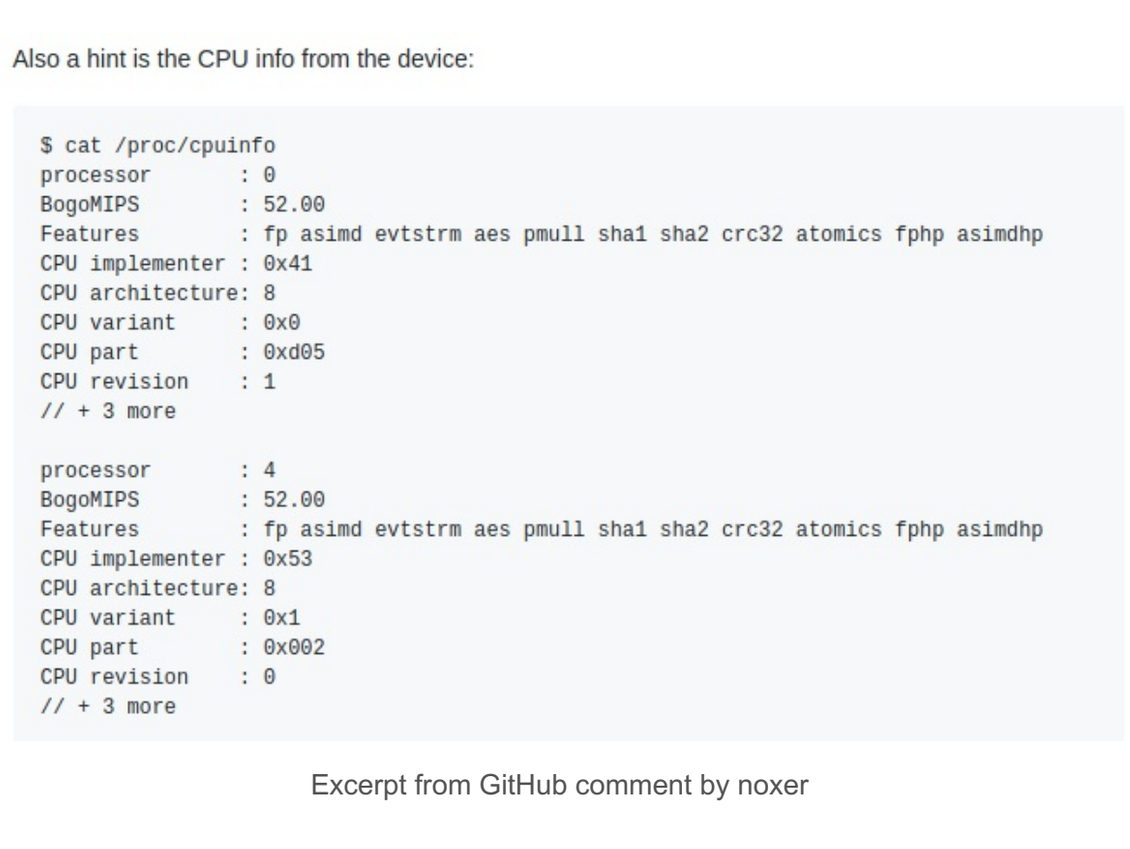
· · ·

Debugging

Wednesday October 24—The first occurrence of this issue (at least the first I know of) was found in shipit's Android App code written in the Go programming language. The program was killed by Android after it attempted to run an unsupported (illegal) instruction. This test only seemed to occur on Samsung Galaxy S9(+) smartphones.

The issue was bisected to commit 0a7ac93c27c9ade79fe0f66ae0bb81484c241ae5. This code revision modified the Go compiler (the system that converts human-readable code to machine instructions) to use the ARM `LDADDAL` atomic increment instruction—which quickly (98% faster than the old way) and safely increments a number that is shared between multiple cores. This was guarded by a check that the CPU supports the instruction—and a fallback instruction which would be run instead if atomic instructions (including `LDADDAL`) were not supported. This check was reviewed and was confirmed to be correct.

Then we had a breakthrough.



Also a hint is the CPU info from the device:

```
$ cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 52.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd05
CPU revision    : 1
// + 3 more

processor       : 4
BogoMIPS        : 52.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp
CPU implementer : 0x53
CPU architecture: 8
CPU variant     : 0x1
CPU part        : 0x002
CPU revision    : 0
// + 3 more
```

Excerpt from GitHub comment by noxer

There are a few key values to pay attention to here:
1. CPU architecture/variant
2. CPU implementer

The CPU architecture/variant pair indicates that cores 0–3 support ARMv8.0 instructions, and cores 4–7 support ARMv8.1 instructions.

I dove into the manual for ARMv8.1 and found the changes since ARMv8.0.
As I expected: added atomic increment instructions.

I suggested locking Go to individual cores to confirm whether this was the problem. Using taskset to force the task to run on specific cores, plusmid (Tim Scheuermann) confirmed that it worked correctly on some cores and failed on others.

```
$ go version
Illegal instruction
$ taskset 0x1 go version
go version go1.11.1 android/arm64
$ taskset 0x2 go version
go version go1.11.1 android/arm64
$ taskset 0x4 go version
go version go1.11.1 android/arm64
$ taskset 0x8 go version
go version go1.11.1 android/arm64
$ taskset 0x10 go version
Illegal instruction
$ taskset 0x11 go version
Illegal instruction
$
```

Go crashes on certain cores and works correctly on others.

Great! Now we know what the problem is at least! The big cores support atomics and the LITTLE cores don't!

That turned out to be incorrect, but was close enough to the truth that I now knew where to look.

· · ·

Peculiar Design Decisions in the Samsung Galaxy S9(+)

I began researching the hardware of the Samsung Galaxy S9 to try to figure out what exactly was happening. Strangely, there are 2 versions of the S9—one for North America, and one for the rest of the world. The North American version uses the Qualcomm Snapdragon 845 processor, and the other version uses the Samsung Exynos 9810 processor. The phone on which the bug was discovered was purchased in Germany, so I narrowed the problem down to the Exynos 9810.

On WikiChip I found the specifications for the Exynos 9810.

> *The processor is fabricated on Samsung's 10 nm process and features 8 cores in a DynamiQ configuration consisting of 4 Mongoose 3 big cores operating at 2.9 GHz and 4 Cortex-A55 little cores operating at 1.9 GHz.—WikiChip*

What? Mongoose 3 does not sound like a name ARM would use for their CPUs. Then I noticed that the CPU implementer IDs were different. Samsung designed the Mongoose 3 big core, and ARM designed the Cortex-A55 LITTLE core. Mongoose 3 uses ARMv8.0, and Cortex-A55 uses ARMv8.2. So my original determination that the big cores were 1 minor revision ahead was incorrect—the LITTLE cores were actually 2 minor revisions ahead.

· · ·

Solutions?

I spent several days trying to figure out a way that Go could work around this. My first thought was to lock to the current OS thread. That ended up not being an option—there is no reasonable way for Go to figure this out. Part of this problem is that Android makes the same assumption that Go does and returns the same feature detection values on all cores.

There was only one reasonable fix I could find. If Samsung patched their Android version to report lack of atomics on all cores, everything would work as expected.

I have been attempting to contact Samsung about this issue. Below is my corresponding Twitter thread.

Jaden Weiss

· · ·

Thanks to shipit for discovering this issue & plusmid for his debugging work!

· · ·

UPDATE: Apparently the Linux kernel does a sanity check and will disable incompatible cores. Samsung overwrote the check. See below.

Will Deacon
@WillDeacon
Replying to @CompuJad @SamsungMobile
Speaking as one of the maintainers of the arm64 Linux kernel, I'd like to point out that this isn't an issue with the upstream kernel, but appears to be caused by the

UPDATE: Go has added a temporary workaround that disables atomics on all Android phones. This will be released in Go 1.11.3 & 1.12, and is currently merged on master. See the related GitHub comment for more information.