Simulation to Aid Developing Software for Hardware

31-January-2016

Ian Romanick

FOSDEM 2016

Agenda

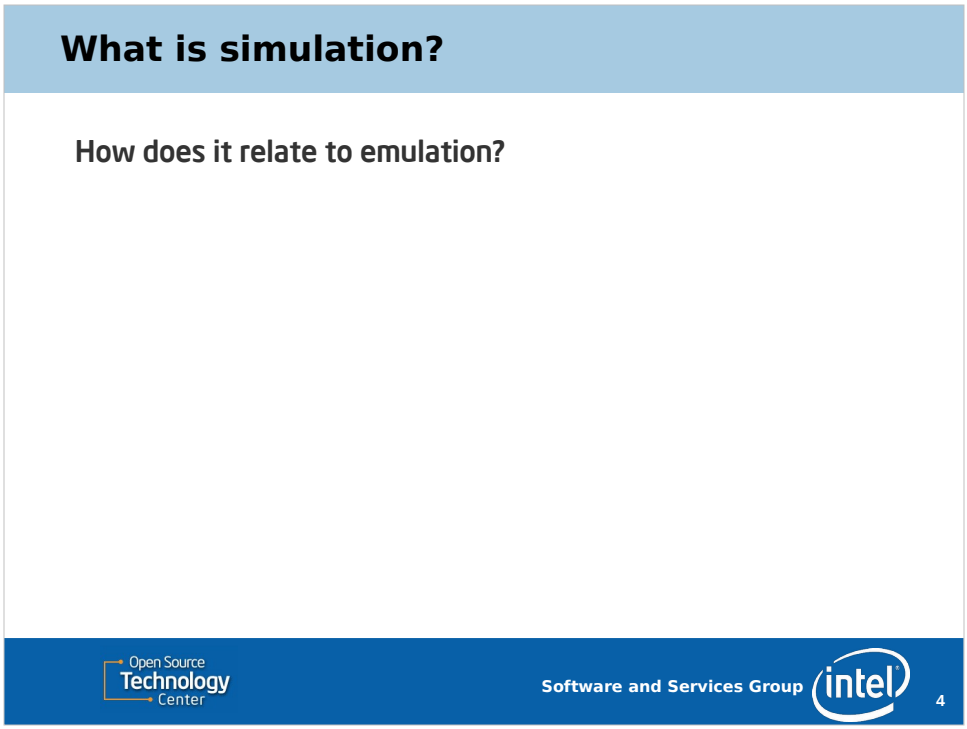
What

Why

How

Not so much who, when, or where

What is simulation?



People often use “emulation” and “simulation” interchangeably, but I believe they are very different beasts. Emulation, especially of vintage computer systems is quite popular these days.

Let's start by thinking about emulation, and then think about how simulation differs.

What is emulation?

Wikipedia says:

> ...imitation of a computer or other electronic system with the help of another type of computer/system.

Not terribly helpful

This is a completely accurate definition. It is perhaps a bit too broad, and it really fails to capture the essence of what emulation is.

What is emulation?

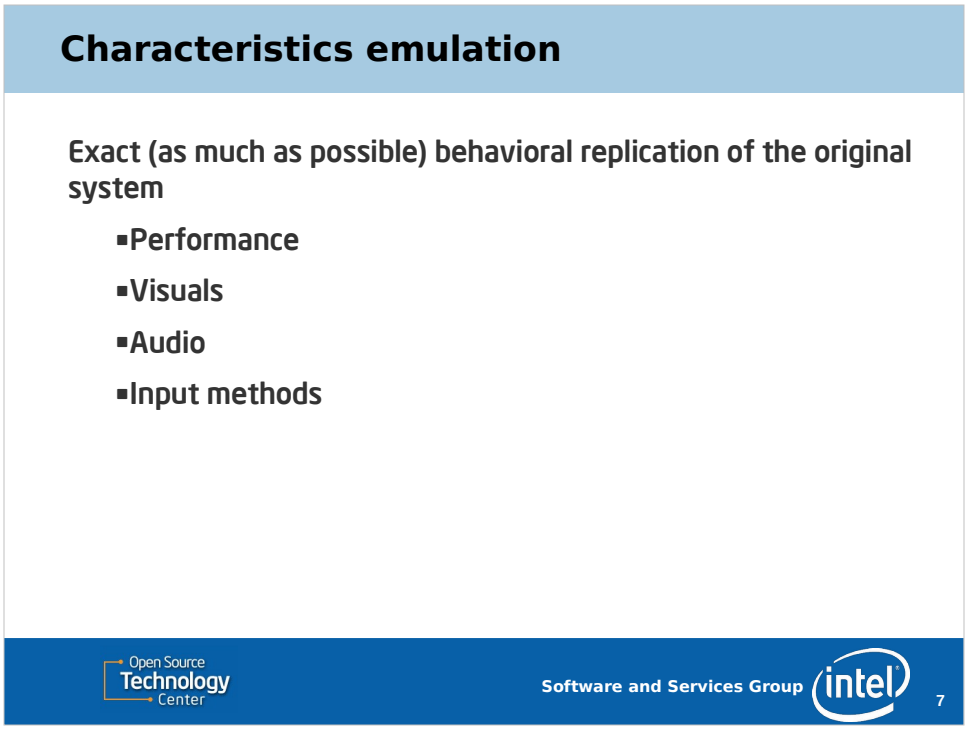
Wikipedia says:

> ...imitation of a computer or other electronic system with the help of another type of computer/system.

More useful:

> Use of one type of computer / system to replicate the user experience of another type of computer / system.

The fundamental focus of many emulators is to reproduce the user experience of some other computer or electronic system. Our definition really needs to embody that.



I believe these are the four most important areas that an emulator must faithfully reproduce.

For many applications performance is of equal or even greater importance than other concerns. If a game system emulator fails to hit the target framerate, it doesn't matter how faithfully the graphics or sound is reproduced. The system has failed.

Canonical example: MAME

MAME is probably the quintessential emulator.

It also provides some representative examples of the difficulty of “exact behavioral replication” of the original.

The classic Star Wars arcade game is difficult in two ways. First, it uses a vector display. Now vector displays are all but extinct. Unlike a CRT that scans the electron beam in a left-to-right-top-to-bottom fashion, a vector display scans in arbitrary direction to create the display.

In addition, the game uses a flight yoke for input. There is nothing quite like this available for a modern PC.

So... you have to try to recreate the experience with the I/O technologies available. Input systems are remapped, and great care is taken to reproduce the look of a vector display on a CRT or LCD. MAME developers have put quite a bit of effort into this.

What is simulation?

Wikipedia says:

> ...an attempt to model a real-life of hypothetical situation on a computer so that it can be studied to see how the system works.

This definition is really has things like simulating the formation of the universe or diffusion of gasses in mind, but it's spot on for our purposes.

We want to study the hardware / software system to understand how it works... or often during development, how it fails to work,

What is simulation?

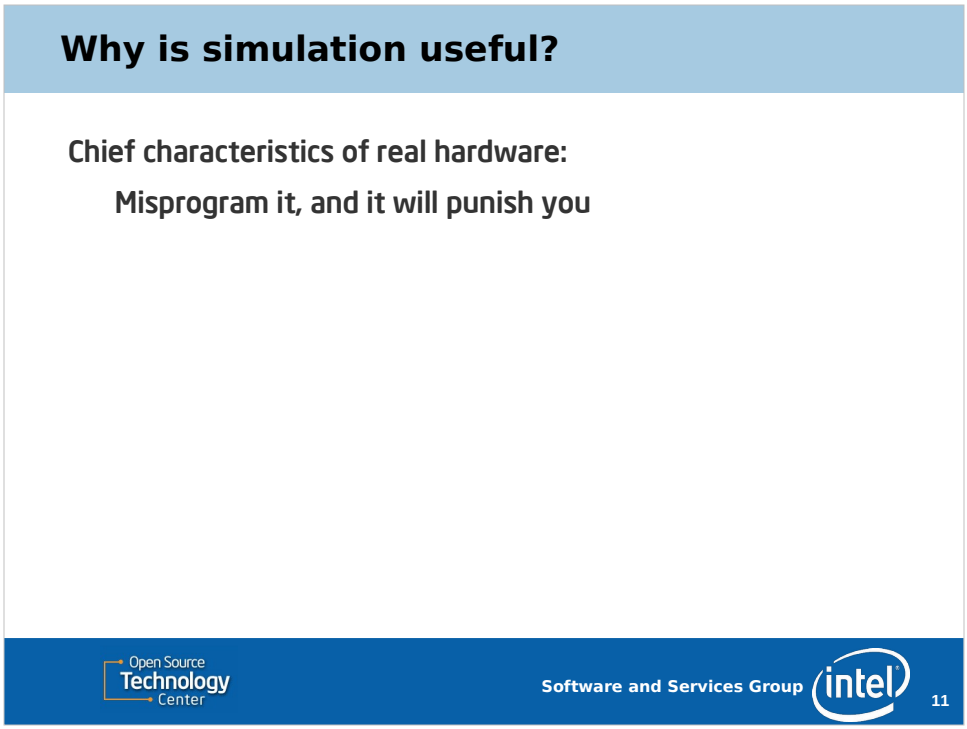
Chief characteristic of a simulator:

- Introspection

Fundamentally, simulation and emulation differs in the target audience: developers or end users.

What's important to note is we don't really care about the user experience. Performance and reproduction of kooky I/O systems aren't even a concern.

Instead, we want to peer behind the curtain to see the inner clockwork.



Programming hardware is fundamentally hard. You do a bunch of work to make an offering to the dragon, and it either continues sleeping or it burns your village down.

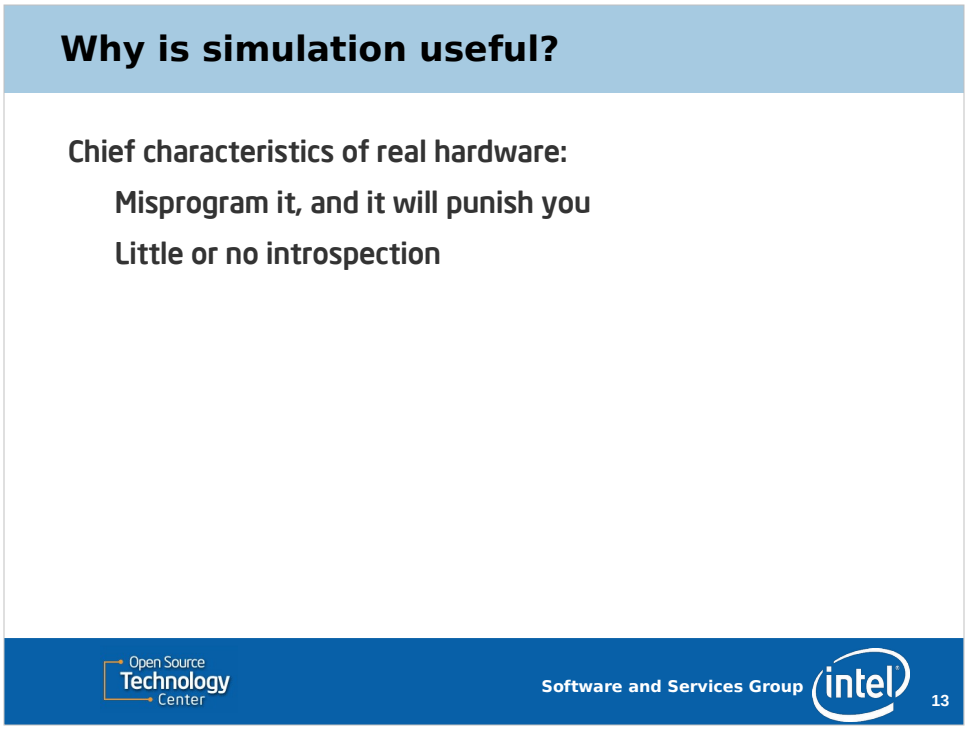
Especially in the presence of hardware bugs, the difference between an accepted offering and running and screaming is subtle and illogical.

Why is simulation useful?

Chief characteristics of real hardware:

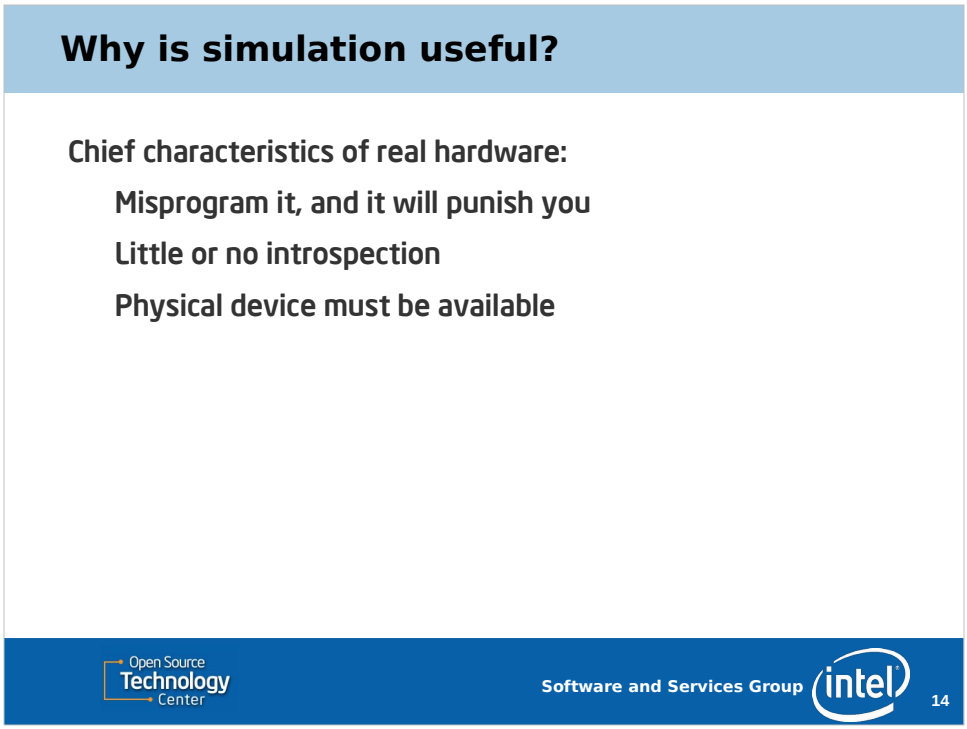
Misprogram it, and it will punish you

- System crash
- Blank screen
- Damage to physical device
- ...
- Little of no feedback of what went wrong



Even in cases where the is introspection available, it is generally focused on performance characteristics of the system. This is useful for application developers, but it's not as interesting for driver developers.

When the hardware has been misprogrammed, the introspection capabilities that do exist will often fail.



Oh yeah, to work on a physical device, you have to have the physical device!

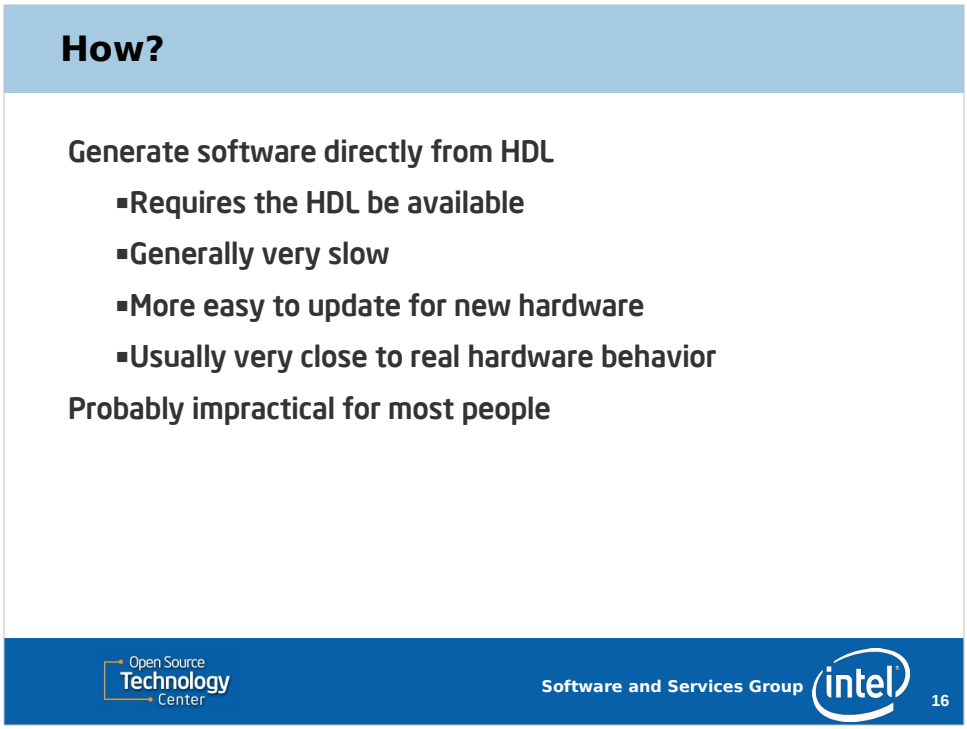
I have a Broadwell GPU in this laptop, but if I need to fix a driver bug on an older or newer chip, I have to go dig a different laptop out of a drawer. Right now, that drawer is about 8,000 km away. Wait a moment, I'll be right back... lol.

How?

Generate software directly from HDL

The HDL is already used to synthesize the physical device, with some work you can synthesize another piece of software the reproduces the behavior of the physical device.

Simple, we're done. Right?



Assuming you can get access to the HDL... show of hands, who has access to the HDL for any GPU?

The HDL is with logic gates in mind, so a simulator generated from the HDL will generally replicate the behavior at or very near the gate level. This results in a lot of CPU instructions to do even a very simple operation. **S-L-O-W.** A single frame that takes 10ms on real hardware may take minutes or hours to simulate.

But... there are many advantages to this method.

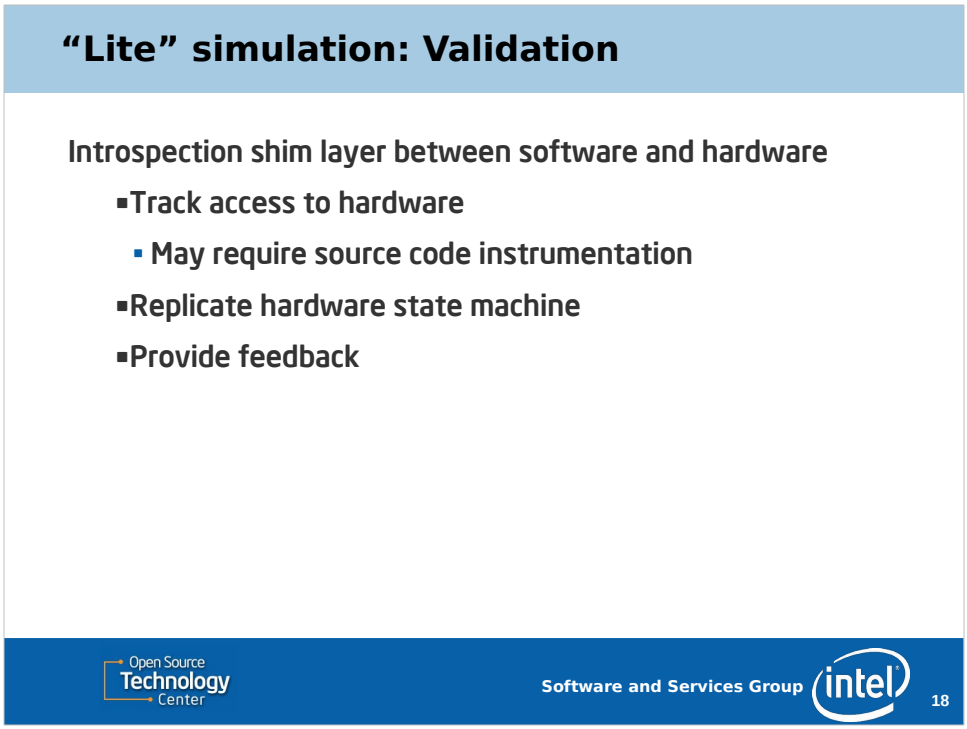
All of the hardware companies have simulators generated from HDL. We have one for all our GPUs, and the simulator is available long before the physical device is, so this is a critical part of bringing up support for new hardware in time for launch.

For most things beyond bring up, this simulator is just too painful to use.

How?

We can start by dipping our toes in the simulated water...

NOTE: This is supposed to be an animated GIF of simulated water being poured into a container. This probably won't make it into the exported PDF.



We can start by dipping our toes in the simulation waters by running on real hardware but also partially simulating the internal state of the hardware.

I'll call this a *validator*. We just want to ensure that all of the stuff sent to the hardware is valid.

“Lite” simulation: Validation

Prevent direct software access to hardware

Otherwise the simulator can’t see the hardware accesses

Before:

```
// Enable copper and display DMA
custom->dmaconr = 0xD018;
```

After:

```
// Enable copper and display DMA
WRITE_WORD(custom, dmaconr, 0xD018);
```

Open Source Technology Center — Software and Services Group — intel — 19

I actually started down this road last summer. July was the 30th anniversary of the Amiga computer, so I started playing around with some coding projects. I actually got my start writing demos on the Amiga... long ago, so it was fun to go back to it for a bit.

The Amiga has a set of memory mapped registers. All of the registers are read-only or write-only. Poking values in the registers make things happen. In addition, it has a coprocessor called the copper that can perform register writes synchronized to CRT beam positions.

A common thing to do is enable DMA for the copper and the display. To validate this, we have to replace the direct write with an indirect access.

(NOTE: This code has at least two bugs... more on that later.)

“Lite” simulation: Validation

```
#ifdef SIMULATION
#define WRITE_WORD(base, field, value)       \
    write_u16(base,                          \
              offsetof(struct Custom, field),\
              value);
#else
    do { (base)->field = value; } while(0)
#endif
```


In a “release” build, this becomes a direct write.

In a “simulation” build this goes through an intermediate step that updates the internal state machine of the simulator.

“Lite” simulation: Validation

```
// Enable copper and display DMA
WRITE_WORD(custom, dmaconr, 0xD018);
```


So, we take this bit of innocuous looking code and run it through the simulator...

The high-order DMACON bit selects whether the other bits turn on or turn of particular DMA channels. The remaining bits select the channels to be modified.

“Lite” simulation: Validation

```
// Enable copper and display DMA
WRITE_WORD(custom, dmaconr, 0xD018);
```

Simulator result:

```
Write to read-only register.
```


Riiiiight.. there are two DMA control registers. One is read-only, and the other is write-only. DMACONR is the read-only register, so we'll wonder why the display didn't come on.

So, we'll fix that...

“Lite” simulation: Validation

```
// Enable copper and display DMA
WRITE_WORD(custom, dmacon, 0xD018);
```

“Lite” simulation: Validation

```
// Enable copper and display DMA
WRITE_WORD(custom, dmacon, 0xD018);
```

Simulator result:

```
Write to read-only DMACON bit.
```


In addition to bits for various DMA channels, there are some read-only, informational bits. This value mask is trying to turn on the blitter busy flag. This won't have any affect, but it's probably not what you intended.

“Lite” simulation: Validation

```
// Enable copper and display DMA
WRITE_WORD(custom, dmacon, 0x8018);
```

Simulator result:

Fix that, and the simulator is happy.

Now, this is a very simple example that doesn't require the simulator track any state. It just had to look at the registers and the values.

That was enough to help find some software errors. How many times have you spent hours tracking down a bug that was just a one character typo?

“Lite” simulation: Validation

```
WRITE_SHORT(&custom, bltcon0, BC0F_SRCA |
                              BC0F_DEST |
                              A_TO_D);
WRITE_SHORT(&custom, bltcon1, 0);
WRITE_SHORT(&custom, bltsize, (16 << 6) | 8);
WRITE_LONG(&custom, bltapt, src1_addr);
WRITE_LONG(&custom, bltdpt, dst1_addr);
WRITE_SHORT(&custom, bltamod, 0x0000);
WRITE_SHORT(&custom, bltdmod, 0x0000);
```


Here's a more complex example.

The Amiga has a hardware blitter, and this code sets up a simple source-to-dest copy of a 16 line by 8 word block.

The source and destination pitch is determined by the combination of the blit width and the “modulus” registers.

But there's a big problem here. Writing the size register cause the blit to start, and this code is trying to program the pointer and mod registers after programming the size. This may not necessarily be a bug... you might want to use values left from the previous blit.

However, programming any blitter registers while the blitter is busy will probably not do what you wanted because it will cause the hardware to start using the new values in the middle of the blit.

“Lite” simulation: Validation

```
WRITE_SHORT(&custom, bltcon0, BC0F_SRCA |
                              BC0F_DEST |
                              A_TO_D);
WRITE_SHORT(&custom, bltcon1, 0);
WRITE_SHORT(&custom, bltsize, (16 << 6) | 8);
WRITE_LONG(&custom, bltapt, src1_addr);
WRITE_LONG(&custom, bltdpt, dst1_addr);
WRITE_SHORT(&custom, bltamod, 0x0000);
WRITE_SHORT(&custom, bltdmod, 0x0000);
```

Simulator result:

```
Write to blitter register while blitter
active.
```


By tracking whether or not the blitter busy flag has been polled, the simulator can detect that there is probably a problem.

“Heavy” simulation

ksim by Kristian Høgsberg

- Shim between libdrm and the kernel
- About 7kloc
- Replicates rasterizer, execution units (mostly), pixel blending, visibility, etc.

“Heavy” simulation

Kernel / user split already restricts direct hardware access

- Hardware commands are marshaled into buffers
- Buffers are sent to kernel using ioctls
- User-mode software never directly touches hardware

“Heavy” simulation

Kernel / user split already restricts direct hardware access

Simulator catches ioctls in user mode

For most ioctls, marshaled data is sent to a second process

- Uses DRI3 buffer sharing

Simulator process performs request operations, sends data back

- Exactly as the hardware & kernel would have done!

Show demo of glxgears running in the simulator.

Tips

Start small
- Only simulate or validate the elements you need

Unit test
- Lots of small tests that poke simulator corner cases

Probe corner cases on real hardware
- Even if you have hardware docs, they may be wrong or incomplete

Support multiple hardware variations
- Give additional feedback for non-portable constructs

Unit testing is critically important. Almost all of the error cases that a validator detects are corner cases. Unit tests should poke at each of them.

Fringe benefit

By writing and refining your own simulator, you learn a **lot** about the real hardware.

Even if you never run any real software through the simulator, you will learn about the real hardware. Spending time closely examining documentation looking for undefined or invalid behavior, writing tests to probe corner cases, etc. is instructive in ways that nothing else can be.

Questions?

Resources

https://github.com/krh/ksim

http://cgit.freedesktop.org/~idr/amiga-sim

Soon, but not quite yet.

Legal Information


intel
Leap ahead™