



petalogix

LINUX SOLUTIONS FOR A
RECONFIGURABLE WORLD

Dynamic Co-simulation of FPGA- based Linux Systems-on-Chip

<john.williams@petalogix.com>

<peter.crosthwaite@petalogix.com>

PetaLogix Labs

What?!

- Dynamic
 - Data-driven simulation models
- Co-simulation
 - Gluing multiple simulation frameworks together
- FPGA-based Linux SoC
 - Using FPGAs to create custom SoC architectures, and running Linux on them

Why?

- Linux
 - <insert your own answer here>
- FPGA-based SoC
 - designer SoC, with ***workstation tools***
 - No Fab required
- Simulation
 - Increasingly necessary and popular approach
 - Test against platforms that don't yet exist
 - Prototype HW too rare to share among SW development teams
 - Allow behavioural visibility beyond that possible in real HW
 - e.g. Android DK (QEMU-based)

The story so far

- ELC 2009
 - Embedded Linux on FPGAs for fun and profit
- ELC 2010
 - Custom Hardware Modelling for FPGAs and Embedded Linux Platforms with QEMU
- Videos of these talks available thanks to Free Electrons
 - <http://free-electrons.com/blog/elc-2009-videos/>
 - <http://free-electrons.com/blog/elc-2010-videos/>

Dynamic QEMU platforms

- To date, all QEMU machine models are static
 - `qemu/hw/...`
 - `#define ADDRESS_OF_MY_DEVICE 0xdeadbeef`
 - etc
- Plenty of kernel machine models are static too
 - `linux-2.6.x/arch/arm/mach-foo/..`
 - `#define ADDRESS_OF_MY_DEVICE 0xdeadbeef`
 - MicroBlaze, PowerPC and a few others get this right
 - ARM is coming eventually, we hope

Dynamic QEMU platforms

- Device Trees are an increasingly popular representation of the machine model
 - Permits completely generic kernel platform code, all data-driven
- Way back in 2009, we wondered
 - Since our kernel is DT driven, can we just assemble QEMU machines on the fly from the same data structure?
- The answer is a resounding “yes”
 - Implementations for MicroBlaze and PPC440
 - ARM Cortex A9 (Zync) platform coming

Dynamic QEMU platforms

```
$ qemu-system-microblaze \  
-M microblaze-fdt -hw-dtb system.dtb \  
-kernel vmlinux
```

- `system.dtb` is (but needn't be) exactly the same binary blob that will configure the kernel when it boots
 - in the VM, or
 - on the real board

So this is cool, what's next?

- Developers of FPGA-based SOCs typically assemble a mix of standard and custom IP
- Standard stuff
 - CPU, busses, interrupt controllers, timers, etc
- Custom stuff
 - Anything they want!
 - Custom IO, data processors

Test what you fly, fly what you test

- There's not much point only simulating the bits that you already know work
- We want to be able to develop complete driver and SW stacks against simulation models
 - Rapid feedback on IP/driver interactions
 - Simple architecture modelling

Test what you fly, fly what you test

- It is quite possible to create new QEMU device models for arbitrary custom 'stuff', but...
 - It's not easy (fairly complex device model, esp. for asynchronous behaviours)
 - you have to hack on QEMU (Makefiles, source code, ...)
- We may want a **range** of models for the same device
 - Fast, low fidelity – good for quick driver/device prototyping
 - Slow, high fidelity – detailed low-level view of device behaviour

External device models

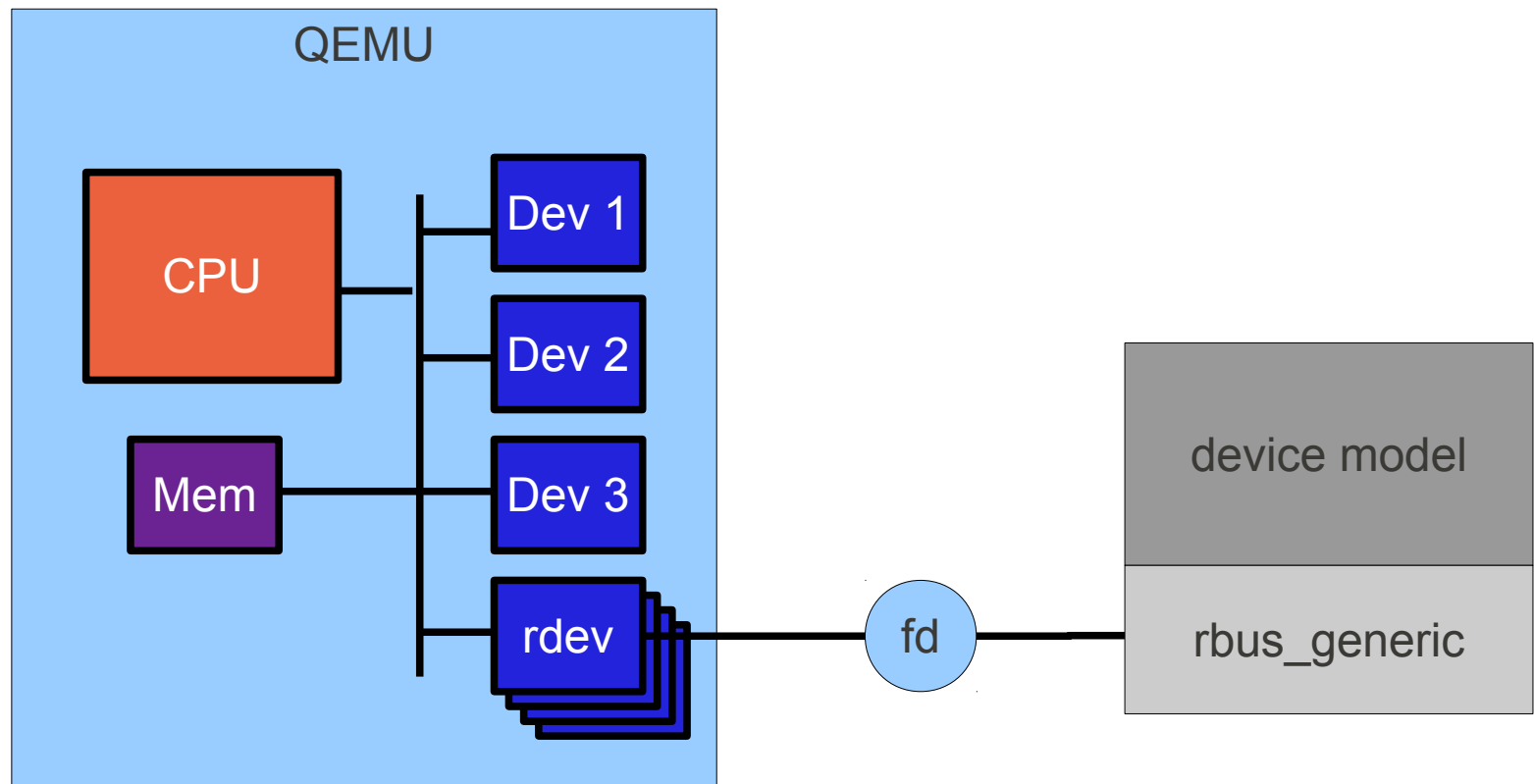
- Is there a way we could wrap the QEMU device model API into some simplified abstraction?
 - Yes!
- Communicate to external device models over
 - sockets
 - pipes
- Need support for
 - slave transactions
 - master transactions
 - IRQs and other asynchronous events

External device models

- Build upon the 'rbus' (remote bus) API from Edgar Iglesias
 - In essence, a way of serialising bus transactions into a generic packet format
- To QEMU, we added
 - Standard handling for transport (pipes/sockets etc)
 - A generic rbus stub device that is the bridge between QEMU and the outside world
 - Dynamic binding mechanism
 - Device Trees still rule the world

External device models

- One 'rdev' device for each remote device



External model spawning

- Added ability for QEMU to spawn child processes for each remote device

```
$ qemu-system-microblaze \  
-M microblaze-fdt -hw-dtb system.dtb \  
-kernel vmlinux \  
-rbus-spawn /path/to/devmodel-exe \  
-rbus-spawn /path/to/devmodel2-exe
```

- QEMU will spawn one instance of the remote device model, for each instance found in the system Device Tree

External model binding

- How to bind an external model to a particular 'personality'?
 - In the startup handshake, device models declare their compatibility
 - QEMU remote device handling binds based on this
- In this model, all devices of a particular type, will bind to the same remote model
 - It is also possible to specify bindings on a per-instance basis

“Hello, device model world”

- Peripheral has a single register @ offset 0x0
- The register is both readable and writeable
 - 32 bit bus architecture
 - Byte/halfword ops forbidden
- Updating the register changes the state of attached LEDs
 - Simple text output to simulation console

Read/Write Transaction Handlers

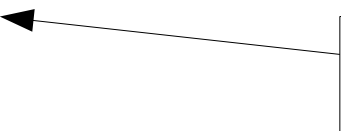
```
uint32_t reg;
```

The single register



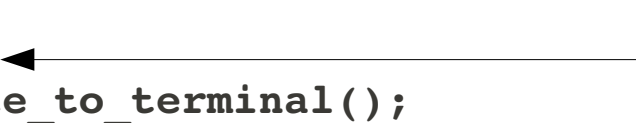
```
int gpio_leds_slave_read(uint32_t addr, uint8_t byte_en, uint32_t * ret)
{
    *ret = reg;
    return 0;
}
```

Return reg on a read operation

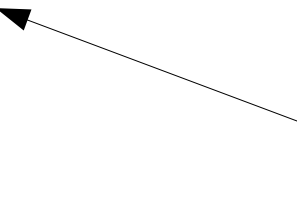


```
int gpio_leds_slave_write(uint32_t addr, uint32_t data, uint8_t byte_en)
{
    reg = data;
    dump_leds_state_to_terminal();
    return 0;
}
```

Update reg on a write operation



Tell the user that the LEDs are now different



Defining the device

```
struct rdev_addr_range addr_ranges [] = {
    {.offset = 0x0, .length = 0x4},
    {.end_of_list = 1}
};

char compats[] = "xlnx,xps-gpio-2.00.a";

const struct rdev_env gpio_leds_env = {

    .slave_read_ranges = addr_ranges,
    .slave_write_ranges = addr_ranges,

    .slave_read = gpio_leds_slave_read,
    .slave_write = gpio_leds_slave_write,

    .slave_read_restricts = RDEV_AR_ALIGN_WORD |
        RDEV_AR_32_BIT | RDEV_AR_WORD_ONLY,
    .slave_write_restricts = RDEV_AR_ALIGN_WORD |
        RDEV_AR_32_BIT | RDEV_AR_WORD_ONLY

};

int main (...) {
    rdev_bind(NULL, compats, NULL);
    rdev_start(&gpio_leds_env);
}
```

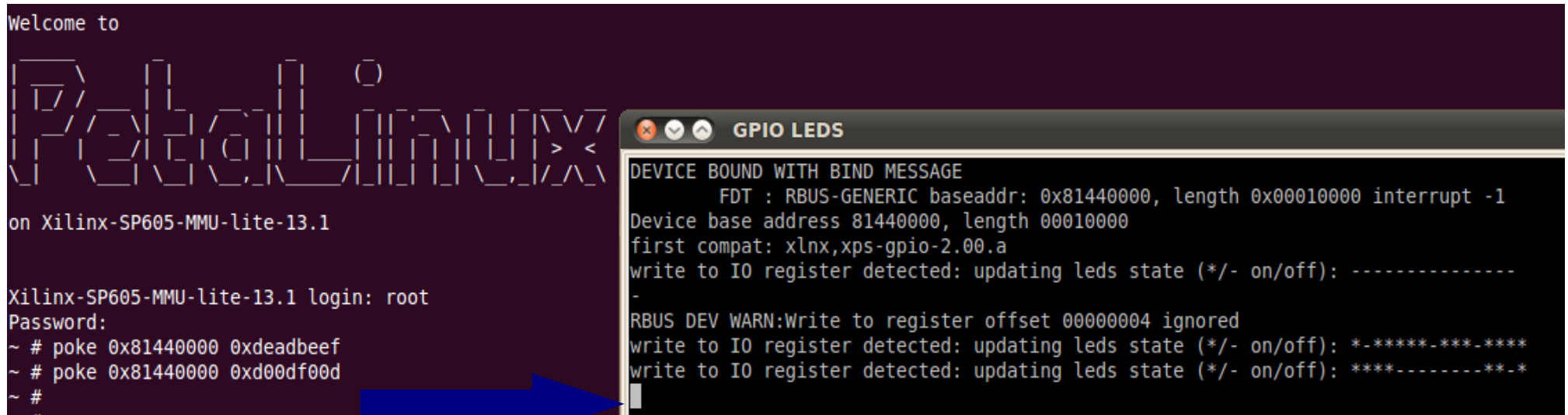
Valid Address Range
(Single Register @ 0)

Register our handlers

Define bus width and
enforce word alignment
with no byte/halfword
access

Start event loop

Runtime



```
Welcome to
PetaLinux
on Xilinx-SP605-MMU-lite-13.1

Xilinx-SP605-MMU-lite-13.1 login: root
Password:
~ # poke 0x81440000 0xdeadbeef
~ # poke 0x81440000 0xd00df00d
~ #
```

```
GPIO LEDS
DEVICE BOUND WITH BIND MESSAGE
  FDT : RBUS-GENERIC baseaddr: 0x81440000, length 0x00010000 interrupt -1
Device base address 81440000, length 00010000
first compat: xlnx,xps-gpio-2.00.a
write to IO register detected: updating leds state (*/- on/off): -----
-
RBUS DEV WARN:Write to register offset 00000004 ignored
write to IO register detected: updating leds state (*/- on/off): *-*****-***-****
write to IO register detected: updating leds state (*/- on/off): ****-*****-***-*
```

PetaLinux Root Console

GPIO_LEDS Device Console

- QEMU binds the device model when compatible FDT node is found
- Kernel initialises the device
- User pokes device directly to change LEDs

Other interesting things

- Added ability for QEMU to load device models from .so libraries
 - Use full QEMU device model, but dynamically linked
 - Requires special device model Makefiles, and ties device models to a particular QEMU API version
- Prototyped ability to simulate device models in ISIM
 - Some interesting questions about time

Q&A