ARM64 + FPGA and more: Linux on the Xilinx ZynqMP

Opportunities and challenges from a powerful and complex chip

Luca Ceresoli, AIM Sportline
luca@lucaceresoli.net
http://lucaceresoli.net

FOSDEM 2018

About me

- Embedded Linux engineer at AIM Sportline
  `http://www.aim-sportline.com/`
  - Develop real products on custom hardware
  - Kernel, bootloader, drivers
  - Integration, build system
- Open source enthusiast
  - Contributor to Buildroot and a few other projects

Agenda

- Introduction
- Development tools
- Linux on your FPGA design
- Booting
- GPU
- Video Codec Unit
- Conclusion

Introduction

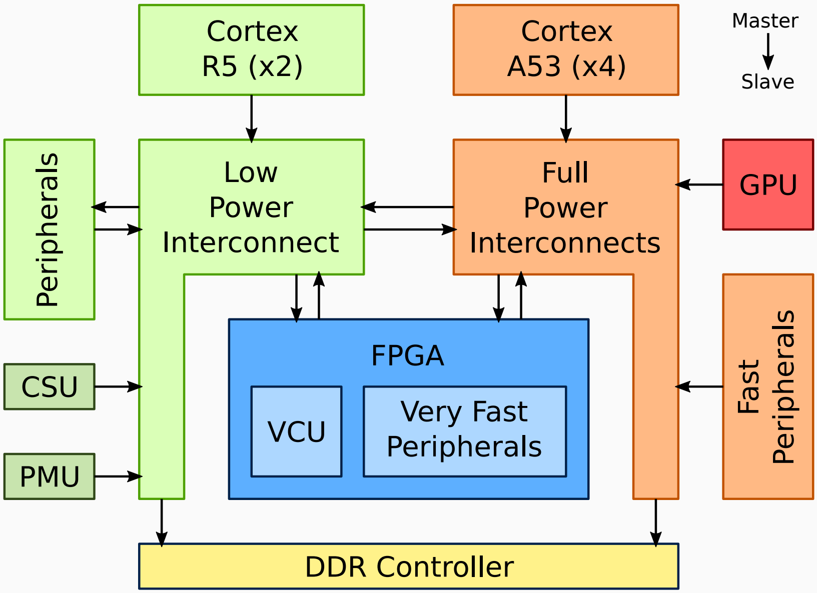
SoC+FPGA: what is it?

- A SoC and an FPGA on a single chip
- Connected on-chip
- Other technologies:
  - 2 chips: a SoC and an FPGA, connected via pins
  - An FPGA with a synthesized “soft core” CPU
- A good introduction to SoC+FPGA:
  Introduction to SoC+FPGA, Marek Vašut, ELC-E 2017
  `https://elinux.org/images/e/ed/Elce-2017-socfpga.pdf`
  `https://youtu.be/R3gJhnGjjWY`

Current Linux-capable SoC+FPGAs

- Xilinx
  - 1st generation: Zynq 7000
  - 2nd generation: Zynq UltraScale+ MPSoC (aka ZynqMP)
- Intel (Altera)
  - 1st generation: Cyclone V and Arria V
  - 2nd generation: Stratix 10

ZynqMP block diagram (simplified)



Development tools

Documentation

- A lot of documentation by Xilinx
- Start from the Xilinx ZynqMP Documentation hub

  `https://www.xilinx.com/support/documentation-navigation/design-hubs/`

  `dh0070-zynq-mpsoc-design-overview-hub.html`

FPGA development

FPGA Development Tools — Xilinx

Xilinx Vivado Design Suite

- Vivado: design FPGA to bitstream
- XSDK: eclipse IDE for firmware development
- Runs on Linux
- Has a zero-cost version (has most features, not the advanced ones)
- Closed source, huge, has some bugs and issues

FPGA Development Tools — Open source

- No open source FPGA toolchain available
- Bitstream reverse engineering in progress (`https://symbiflow.github.io/`)
- On why FPGA open source toolchains matter: `https://blog.elphel.com/2013/10/fpga-is-for-freedom/`

BSP components

U-Boot

- Xilinx is active in mainlining
- Discussion: `U-Boot@lists.denx.de`
- Mainline U-Boot enough to boot
- Newer boards are available at `https://github.com/xilinx/u-boot-xlnx`
- Note: ZynqMP cannot boot U-Boot the “normal” way, see *Booting* section later

Linux kernel

- Xilinx is active in mainlining
- Discussion: `linux-arm-kernel@lists.infradead.org`
- Mainline Linux: partially implemented
- Development in progress at `https://github.com/xilinx/linux-xlnx` (especially the `master` branch)
    - “Hard” silicon features
    - Recent UltraScale+ IPs from Xilinx, mostly video-related

Build system

Available workflows

Typical ways to build the software stack for Xilinx products:

- The **“Xilinx workflow”**
  - Officially supported by Xilinx
- The **“Community workflow”**
  - Similar to other open source projects
  - Supported by the community (with Xilinx contributions)
- Other/mixed workflows

The Xilinx workflow

- FPGA: Vivado
- Baremetal and bootloaders: XSDK
- Petalinux
  - A Xilinx-specific embedded build system
  - Nowadays internally uses Yocto
- Yocto layers on `https://github.com/xilinx`
  - A `meta-xilinx-bsp` fork
  - `meta-xilinx-tools` to use Xilinx tools during the build
  - `meta-petalinux`: a distro layer
  - Note: these layers will soon be moved to subdirs of `meta-xilinx`

The Community workflow

- FPGA: Vivado
- A little bit of XSDK
- Yocto `meta-xilinx-bsp` layer
  `https://git.yoctoproject.org/cgit/cgit.cgi/meta-xilinx/`
  subdir `meta-xilinx-bsp`
  - Until a few weeks ago: in the top dir, and called `meta-xilinx`
- Goal: follow the common practices in FOSS/Yocto
- Not all features supported

Other resources

- meta-xilinx mailing list
  - `https://lists.yoctoproject.org/listinfo/meta-xilinx`
  - Discussion on the Yocto layers
  - Also for general discussion about Linux on Xilinx hardware
- `https://github.com/topic-embedded-products/meta-topic`
  - Support for boards by Topic Embedded
  - But has very useful code (see *Booting* section later)

Buildroot

- Work in progress
- Patches to add basic support under discussion on the Buildroot mailing-list

Linux on your FPGA design

Build your own SoC!

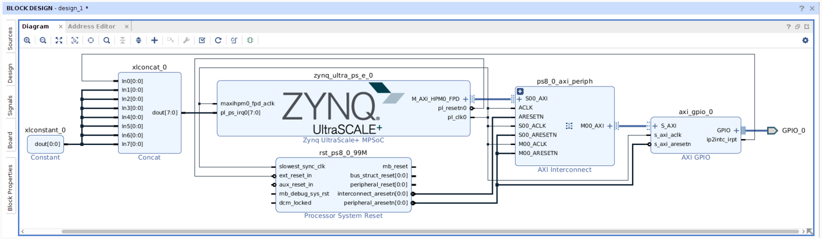
- FPGA $\leftrightarrow$ CPU interface is AXI4
- AXI = Advanced eXtensible Interface (part of AMBA)
- Peripherals in FPGA are accessible on the physical address space
  - Like hard peripherals and traditional SoCs

Typical workflow

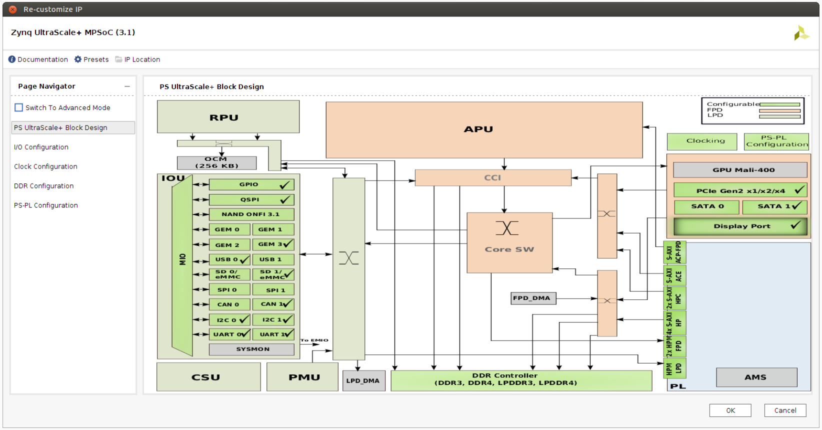
- Workflow with Vivado
    1. IP integration
    2. Address editor
    3. Constraints

Step 1: IP integration

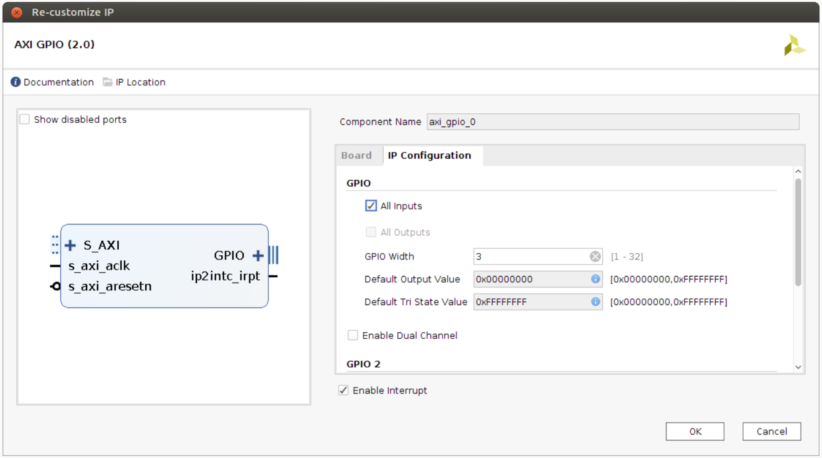
Vivado: block design



Vivado: customize the PS block



Vivado: customize an IP block



Re-customize IP

AXI GPIO (2.0)

Documentation   IP Location

Component Name: axi_gpio_0

Board | IP Configuration

GPIO

[✓] All Inputs

[ ] All Outputs

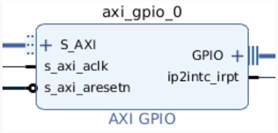
GPIO Width	3	[1 - 32]
Default Output Value	0x00000000	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

[ ] Enable Dual Channel

GPIO 2

[✓] Enable Interrupt

Linux: instantiate a device

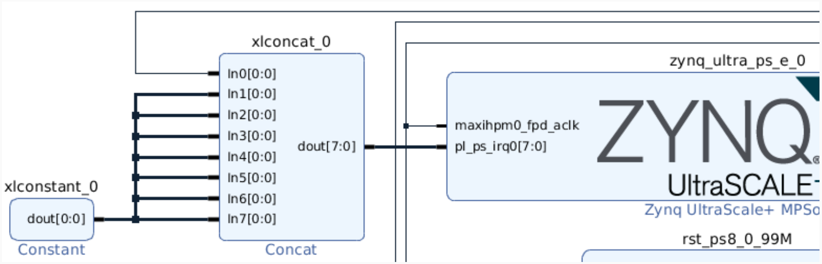


Linux drivers for Xilinx IPs:

`wiki.xilinx.com/Linux+Drivers`

```
gpio: gpio@a0000000 {
  #gpio-cells = <2>;
  compatible = "xlnx,xps-gpio-1.00.a";
  gpio-controller;
  xlnx,gpio-width = <3>;
  xlnx,all-inputs = <1>;
  /*...*/
};
```

Vivado: interrupts



Linux: interrupts

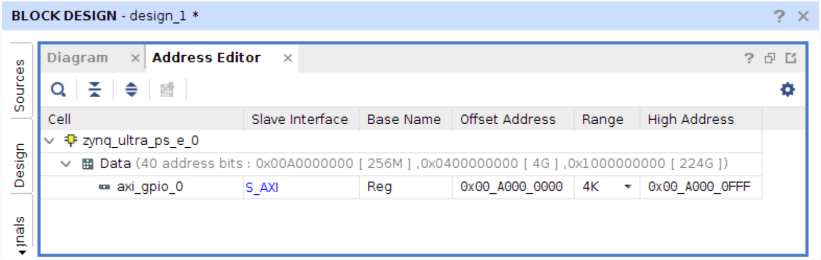
```
#include <dt-bindings/interrupt-controller/arm-gic.h>
#include "zynqmp-irqs.dtsh" // (*)

gpio: gpio@a0000000 {
  #gpio-cells = <2>;
  compatible = "xlnx,xps-gpio-1.00.a";
  gpio-controller;
  interrupt-parent = <&gic>;
  interrupts = <GIC_SPI PL_PS_GRP0_IRQ_0
                IRQ_TYPE_LEVEL_HIGH>;
  /*...*/
};
```

(*) From `github.com/xilinx/qemu-devicetrees/blob/master/`

Step 2: Address editor

Vivado: Address editor



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G])					
axi_gpio_0	S_AXI	Reg	0x00_A000_0000	4K	0x00_A000_0FFF

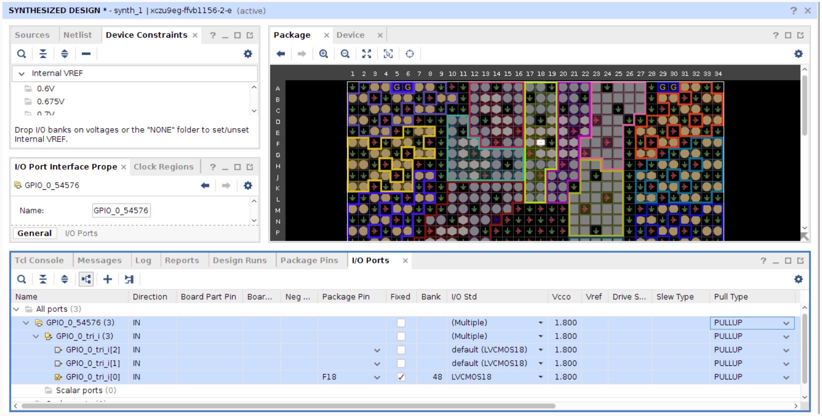
Linux: address map

```
gpio: gpio@a0000000 {
  #gpio-cells = <2>;
  compatible = "xlnx,xps-gpio-1.00.a";
  gpio-controller;
  reg = <0x0 0xa0000000 0x0 0x10000>;
  /*...*/
};
```

Step 3: Constraints

Vivado: constraints

- Connect nets to pins: placement, IO standard, pull-up/down...
- No visible effect in Linux



Internal VREF
0.6V
0.675V
0.7V

Drop I/O banks on voltages or the "NONE" folder to set/unset Internal VREF.

GPIO_0_54576

Name: GPIO_0_54576

Name	Direction	Board Part Pin	Boar...	Neg ...	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive S...	Slew Type	Pull Type
All ports (3)													
GPIO_0_54576 (3)	IN					☐		(Multiple)	1.800				PULLUP
GPIO_0_tri_i (3)	IN					☐		(Multiple)	1.800				PULLUP
GPIO_0_tri_i[2]	IN					☐		default (LVCMOS18)	1.800				PULLUP
GPIO_0_tri_i[1]	IN					☐		default (LVCMOS18)	1.800				PULLUP
GPIO_0_tri_i[0]	IN				F18	✓	48	LVCMOS18	1.800				PULLUP
Scalar ports (0)													

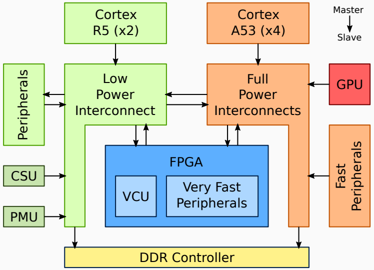
Booting

Simple ARM32 booting

- Good old simple booting on ARM32:
  1. Boot ROM loads SPL into internal RAM
  2. SPL initializes SDRAM, loads U-Boot
  3. U-Boot loads kernel
- But waking up in an ARM64 world is much more complex...

The Platform Management Unit (PMU)

- Duties:
  - Power gates peripherals, power islands, power domains
  - Clock gates peripherals
- Can't boot without it
- Similar to other ARM64 boards
  - E.g. the ARM Juno board

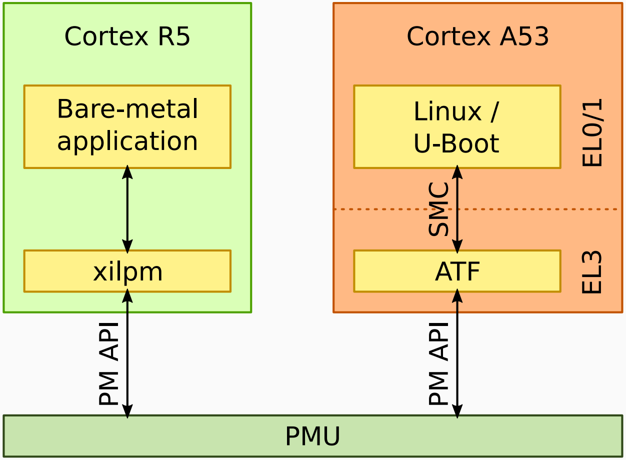


PMU firmware

- PMU is a Microblaze core with 128 kB RAM
- Executes a firmware that *can* be reprogrammed
  - But the one in ROM is not enough, so it *must* be reprogrammed
  - Source at `https://github.com/xilinx/embeddedsw`
  - `meta-xilinx-bsp` `master` can build it (not the Xilinx branches)
- The PMUFW needs a *configuration object*
  - Tells which master (CPU) owns which slave (peripheral)
  - Depends on how the hardware is configured in Vivado

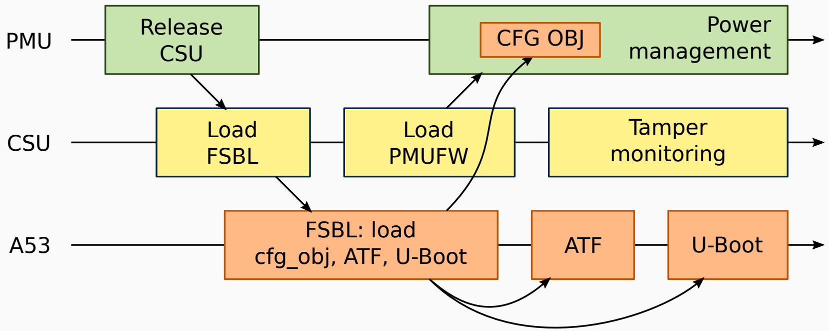
ARM Trusted Firmware

A secure monitor reference implementation

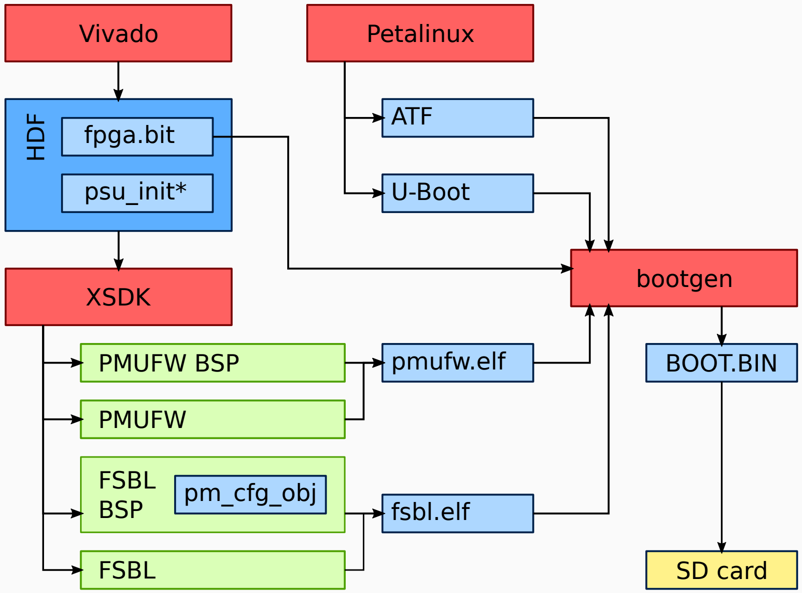


Booting — The Xilinx workflow

Boot sequence



Building the pieces



Pros and cons

Pros

- The Xilinx tools make it easy
- FSBL is easy to understand and debug

Cons

- FSBL is slow (∼3 seconds to load a 4 MB FPGA bitstream)
- The Xilinx tools: big and heavy, hard to automate
- Proprietary `bootgen` tools needed to generate BOOT.BIN
- Non-standard (w.r.t. other SoCs)

Booting — The Community workflow

FSBL → U-Boot SPL

Can U-Boot SPL replace Xilinx FSBL?

FSBL → U-Boot SPL

Can U-Boot SPL replace Xilinx FSBL?

- SPL cannot load FPGA
    - but U-Boot can (∼10x faster)

FSBL → U-Boot SPL

Can U-Boot SPL replace Xilinx FSBL?

- SPL cannot load FPGA
    - but U-Boot can (∼10x faster)
- SPL loads U-Boot proper, not ATF!
    - but there's a trick for that

FSBL $\rightarrow$ U-Boot SPL

Can U-Boot SPL replace Xilinx FSBL?

- SPL cannot load FPGA
  - but U-Boot can (~10x faster)
- SPL loads U-Boot proper, not ATF!
  - but there's a trick for that
- SPL cannot load PMUFW configuration object
  - but there's a workaround for that

SPL: loading ATF

- U-Boot SPL must load U-Boot proper
- But ATF must be there *before* U-Boot proper
- The trick for SPL to load *both* is:

SPL: loading ATF

- U-Boot SPL must load U-Boot proper
- But ATF must be there *before* U-Boot proper
- The trick for SPL to load *both* is:

`configs/xilinx_zynqmp_*_defconfig`

```
CONFIG_SPL_OS_BOOT=y // aka Falcon Mode
```

SPL: loading ATF

- U-Boot SPL must load U-Boot proper
- But ATF must be there *before* U-Boot proper
- The trick for SPL to load *both* is:

configs/xilinx_zynqmp_*_defconfig

```
CONFIG_SPL_OS_BOOT=y // aka Falcon Mode
```

include/configs/xilinx_zynqmp.h

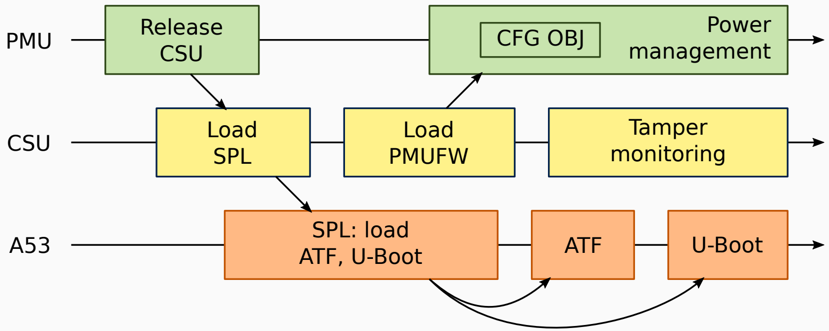
```
/* u-boot is like dtb */
#define CONFIG_SPL_FS_LOAD_ARGS_NAME    "u-boot.bin"
#define CONFIG_SYS_SPL_ARGS_ADDR        0x8000000

/* ATF is my kernel image */
#define CONFIG_SPL_FS_LOAD_KERNEL_NAME  "atf-uboot.ub"
```

Loading PMUFW configuration object

- SPL cannot load PMUFW configuration object
- The current best workaround is:
  - Link `pm_cfg_obj.c` in the PMUFW
  - Load it during PMUFW startup
  - Original patches on the `meta-topic` layer
  - A trivial script to apply the patches and build PMUFW:
    `https://github.com/lucaceresoli/zynqmp-pmufw-builder`
- Pros: it works!
- Cons: must rebuild PMUFW when configuration changes
  - But in the Xilinx workflow you'd have to rebuild FSBL anyway

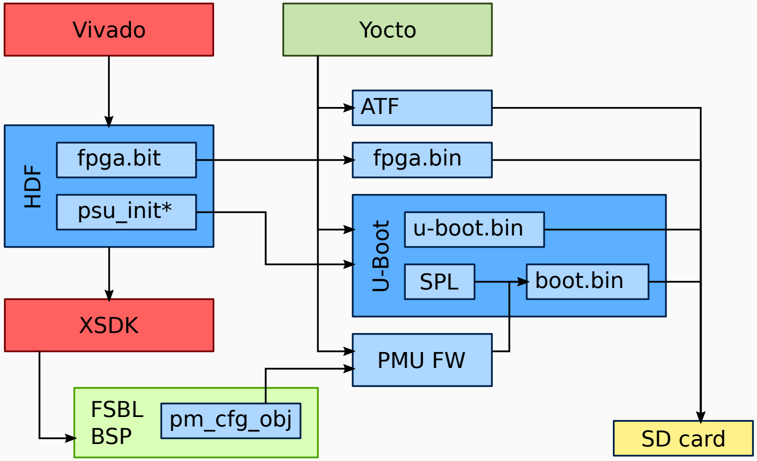
Boot sequence



Peripheral initialization

- Pinctrl and most peripherals must be configured by the bootloader
- To do this in U-Boot:
  1. In Vivado: File → Export → Export hardware...
     - Generates a .HDF file, actually a ZIP file
  2. Extract the HDF file, get `psu_init_gpl.c` and `psu_init_gpl.h`
  3. Put them in the U-Boot sources
     - in `board/xilinx/zynqmp/`
     - and make sure it's not using the bundled ones (see the source code)

Building the pieces



GPU

The GPU

- ARM Mali-400 MP2
- Official software support from ARM/Xilinx:
  - Requires a binary blob
  - Lacks some features
- Open source alternatives:
  - `limadriver.org`: reverse engineering, abandoned
  - `github.com/yuq/mesa-lima`: new Mesa driver in progress

Official software support

- A kernel module
  - GPLv2
  - Not in mainline (perhaps never)
- A userspace library
  - Where the interesting stuff is done
  - Implements OpenGL ES APIs
  - Proprietary, binary only
  - SoC-specific

MALI kernel module

- Yocto recipe on `meta-xilinx-bsp`
- Sources from ARM
- + 10 patches
    - ZynqMP customizations
    - Updates for recent kernels
- Works fine

libmali-xlnx: find it

- Yocto recipe in the Xilinx fork (GitHub)
- *Not* in the “official” `meta-xilinx-bsp` (`master` on `yoctoproject.org`)
- To stay on the `master` path:
    - Copy the whole recipe from Xilinx’s `rel-v2017.4` in your layer

libmali-xlnx: fetch the sources

- The recipe fetches from `gitenterprise.xilinx.com`
  - → `No address associated with hostname`
  - An internal Xilinx repo?
- `docs/MALI-binaries` has the procedure:
  1. Register on `xilinx.com`
  2. Download a tarball, extract it
  3. Read the docs there, extract a 2nd tarball
  4. Extract a 3rd tarball (oh, looks like a bare git repo)
  5. Set `SOURCE_MIRROR_URL` to that path
- Small improvement: extract the 3rd tarball in `DL_DIR`, skip step 5

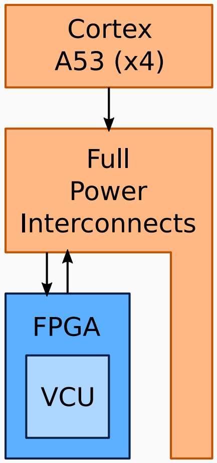
libmali-xlnx: use it

- Two versions to choose from:
- fbdev fullscreen
  - Enough for many embedded products
  - No multi-screen support
- X11
  - Multi-screen does not seem to work

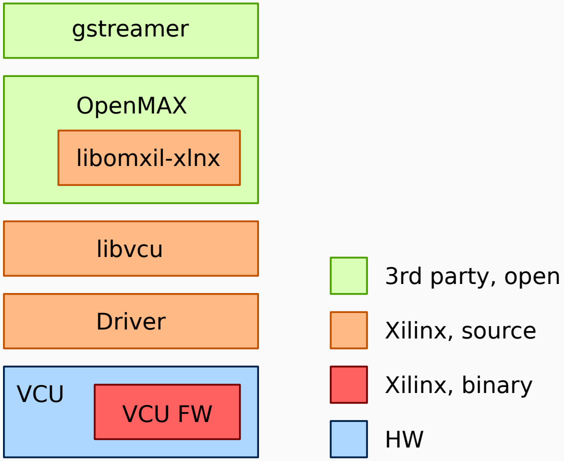
Video Codec Unit

The VCU

- VCU = Video Codec Unit
- H.264 and H.265 encoder and decoder in hardware
- Floating in FPGA, not connected to the hard interonnects



Hardware/Software stack



Instantiate the VCU

1. Instantiate in Vivado (pretend it's an IP block)
2. Devicetree in Linux (in the `linux-xlnx` kernel)

Software

- All recipes on `meta-petalinux`
- Not usable in the Community workflow or with recent Yocto:
  - forces gstreamer 1.8 (Yocto 2.4 has 1.12)
  - points to patches it does not ship
    - some not in poky anymore
    - some don't apply on gstreamer 1.8
- But easy to use manually, see next slide

Shopping list

1. Take Xilinx specific recipes as-is
   - `kernel-module-vcu_git.bb`
   - `vcu-firmware_git.bb`
   - `libvcu-xlnx_git.bb`
   - `libomxil-xlnx_git.bb`
2. Take gstreamer-omx tweaks from Xilinx as-is
   - `gstreamer1.0-omx_%.bbappend`
   - Switches to Xilinx fork
   - Adds build flags
3. install `gstreamer1.0-omx`

Use it

```
gst-launch-1.0 -ve v4l2src device=/dev/video0 \
  ! 'video/x-raw,format=NV12,
     width=1920,height=1080,framerate=(fraction)60/1' \
  ! omxh265enc \
  ! filesink location=video.h265
```

Conclusion

Conclusion

- Very powerful, flexible hardware
- But very complex
- Software support is complex (partly avoidable)
- Xilinx works to support users
  - Slowly moving towards standard tools
- Mainlining effort

Thank you for your attention

Questions?

Luca Ceresoli

luca@lucaceresoli.net

http://lucaceresoli.net