RISC-V Linux

Getting started with Embedded Linux on RISC-V

CREATING A SMARTER FUTURE TODAY

Markus Jämbäck & Pauli Oikkonen

Wapice introduction

Wapice in Short

Full-service Software Company

Solutions used by domain leading industrial companies around the world.

Digitalization Forerunner

We offer close technology partnership and digital services to our customers.

Succeeding Together

The success of our customers is vital to us as it is also a precondition for our own success.

FINLAND

- **Vaasa**
- Helsinki
- Hyvinkää
- Jyväskylä
- Oulu
- Seinäjoki
- Tampere
- Turku

Contact us via
www.wapice.com/contact/company

Technology & Digitalization Services

Analytics, AI and Big Data

Cloud Solutions

Electronics Design & Embedded Systems

Internet of Things Solutions

Web and Mobile Solutions

+

Consulting

Cyber Security

DevOps

Design Services

For more information visit www.wapice.com/services

Preliminary tasks for workshop participants

Prerequisites

- Basic knowledge about Linux in general
  - Running commands in terminal
- Desktop Linux environment
- 40 GB of free disk space
- Material downloaded and extracted

Instructions

- Required material available in Funet Filesender
  - Participants attending to workshop portion should have the link
- Setup one of the alternatives for Linux desktop environment
  - Download and import provided VirtualBox OVA (password: ubuntu)
    - Make sure to use latest version of VirtualBox
    - Importing OVA: File -> Import Appliance…
  - Any OS supported by Yocto with dependencies installed (no support if this causes problems)
    - See https://docs.yoctoproject.org/singleindex.html#system-requirements
- Download and unzip workshop material
- Download and unzip build cache
- Execute initialization script

```
$ wget workshop.zip
$ unzip workshop.zip
$ cd workshop
$ wget sstate-cache.zip
$ unzip sstate-cache.zip
$ ./init.sh
```

Motivation

Advantages of RISC-V in embedded systems

- **Licensing:** free and open to use for everyone
- **Open ecosystem:** possibility for IP reuse
- **Simplicity:** small and fixed base Instruction Set Architecture (ISA)
- **Extensibility:** multiple standard extensions to ISA, also custom ones are possible
- **Versatility:** both soft CPUs and SoCs available
- **Modern:** designed to handle modern compute workloads without legacy burden

Advantages of Linux in embedded systems

- **Licensing:** royalty-free and open-source
- **Hardware support:** all major CPU architectures (RISC-V) with Memory Management Unit (MMU)
- **Networking:** robust TCP/IP network stack and wide range of other protocols
- **Modularity and scalability:** from supercomputers to small embedded systems
- **Commercial support:** huge development effort around the ecosystem
- **Software and libraries:** most applications already run on Linux
- **Tooling:** reasonable to use Linux in actual products

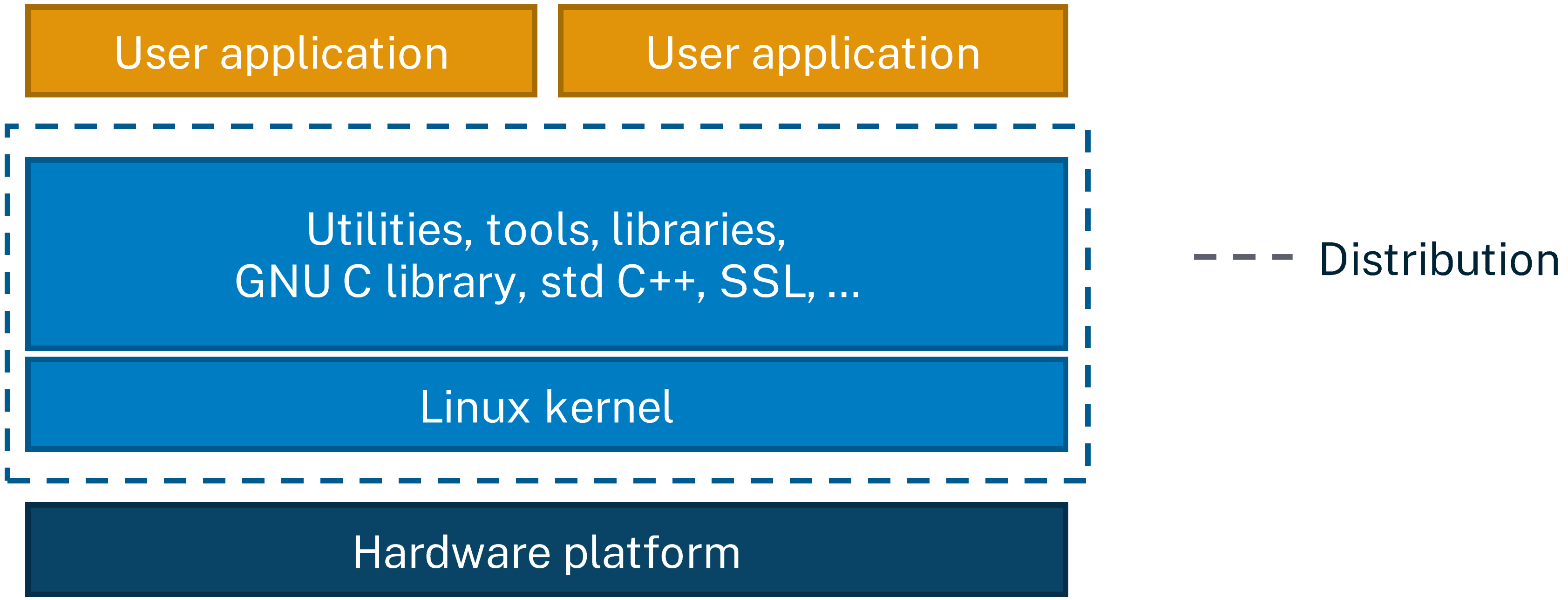
Goals of this workshop

- Introduction to used tools
- Running Linux on emulated RISC-V environment
- Demonstrating same thing on actual hardware
- Developing application software to run on RISC-V platform

Embedded Linux in general

Linux distribution

- In addition to the Linux kernel a functional operating system also requires libraries, utilites and tools
- The whole collection is called Linux distribution



Making custom Linux distributions

- General purpose distribution like Ubuntu?
- Custom distributions especially important for embedded systems
  - Limited resources
  - Efficiency
  - Extensibility
- Building a distribution from scratch is not an easy job
  - C-library (glibc, musl, ...)
  - Init system (systemd, sysvinit, ...)
  - Libraries and utilities (BusyBox, GNU utils, ...)
  - Dependencies between them

Yocto Project

- "The Yocto project. It's not an embedded Linux distribution, it creates a custom one for you."
- Features
  - Widely used in the industry
  - Very flexible and extendable
  - Plenty of ready-made recipes
- Challenges
  - Steep learning curve
  - Slow build times
- Yocto documentation
  - Official: https://docs.yoctoproject.org/
  - Poky source code: https://git.yoctoproject.org/cgit/cgit.cgi/poky/

Core components of Yocto

- OpenEmbedded Core
  - Collection of recipes for building common packages
- BitBake
  - Build engine for executing statements defined in the recipes
- Poky
  - Reference distribution for getting started
  - Contains OpenEmbedded Core and BitBake
- And some others
  - Not so important for today

Getting started

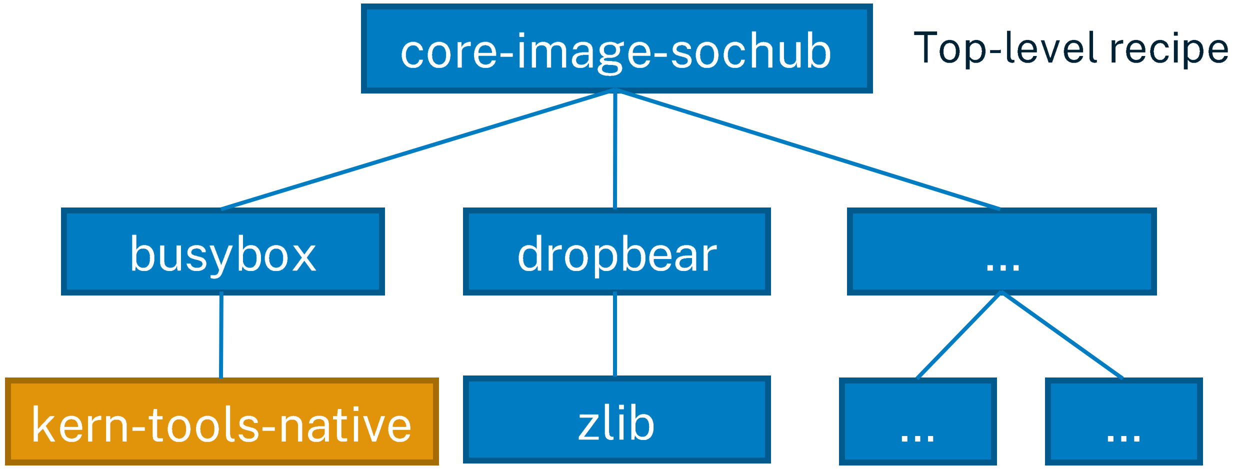
- Yocto repository and sstate cache should be already downloaded
- Directory structure needs to be exactly the same as here (`sstate-cache.zip` not needed)
- Initialization done with simple init.sh script
  - Could be Git submodules etc.

```
workshop/
├── application
├── envsetup.sh
├── init.sh
├── meta-openhw
├── meta-sochub
├── poky
├── sstate-cache.zip
└── sstate-cache
    ├── 00
    ├── 01
    ├── 02
    ├── (…)
    └── ff
```

Internals of Yocto

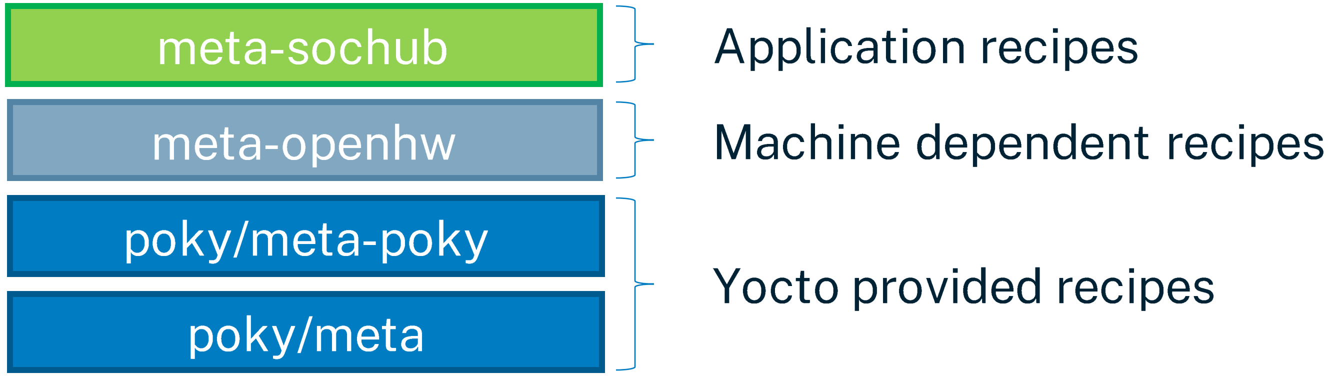
Recipes inside the repository

- Recipe describes how a piece of software is built and packaged
  - How to fetch sources
  - The configuration and build commands
  - How the software is installed to the filesystem
- Recipes for dependency tree that describes the order of the build process
  - Dependencies can be both runtime dependencies as well as build dependencies



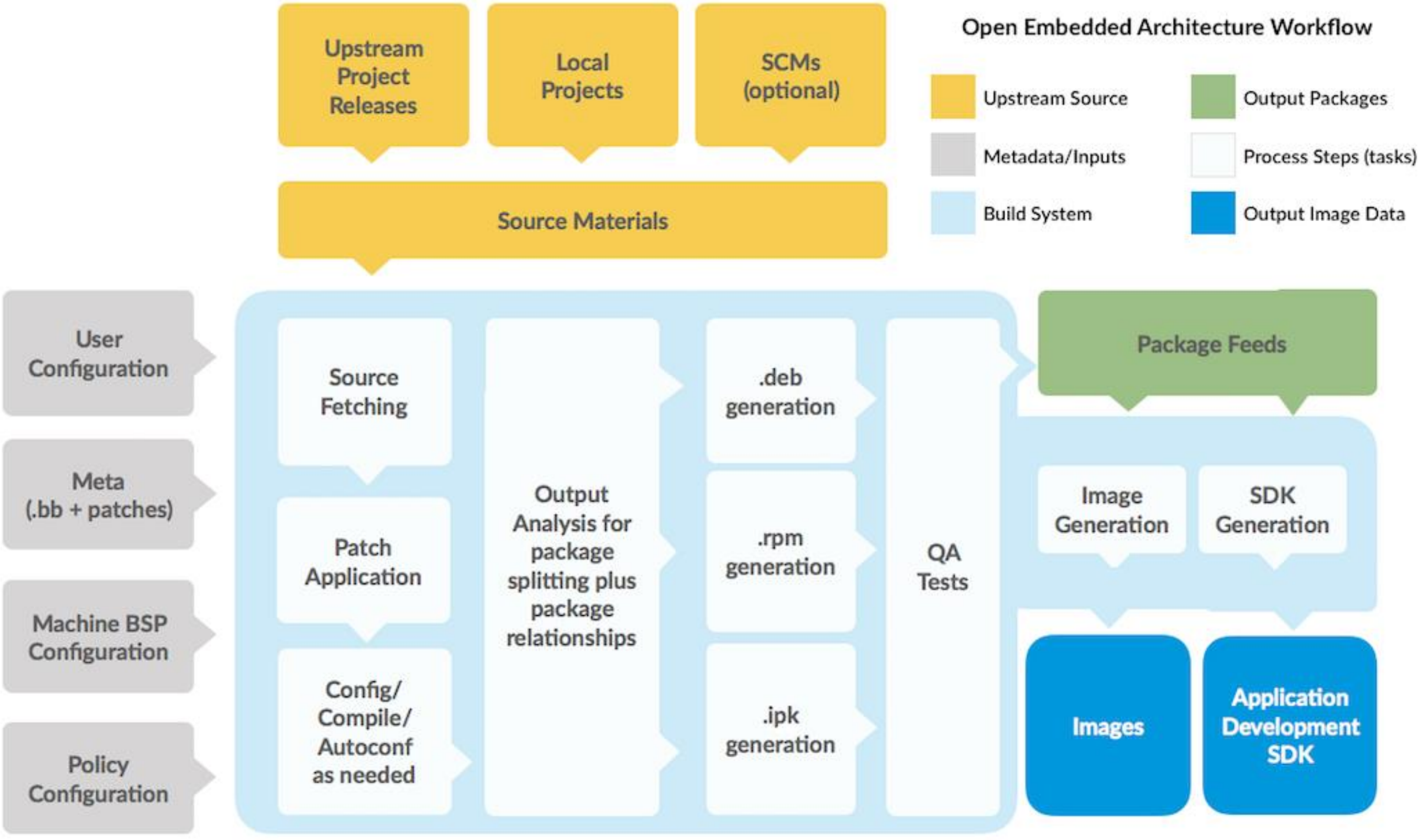
Layers inside the repository

- Recipes are grouped in layers for easier management and re-use
- Layer is defined by directory that contains `layer.conf` file
- Yocto layer stack is defined in `bblayers.conf`



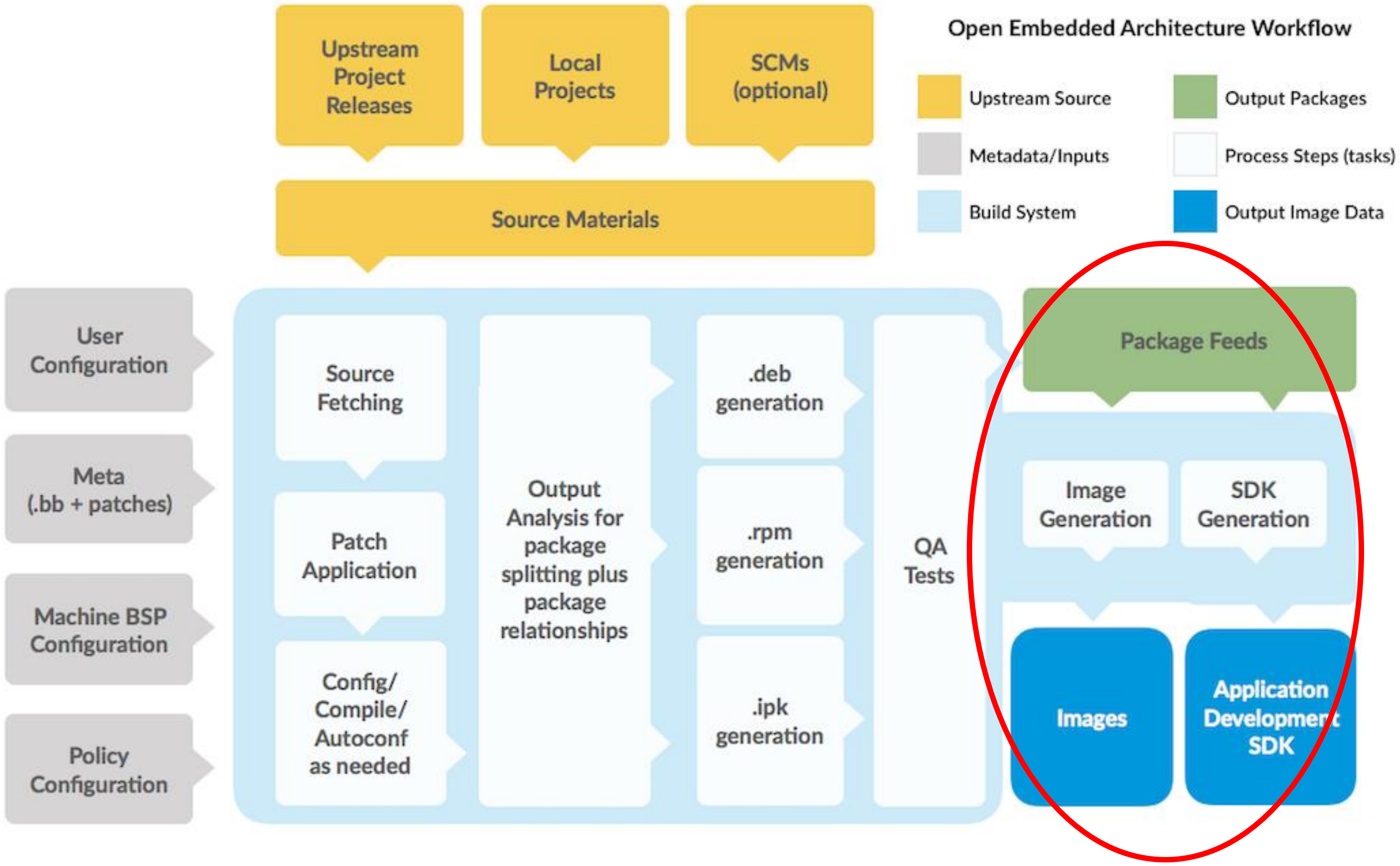
What happens during the build process

- Fetch – get the source code
- Extract – unpack the sources
- Patch – apply patches for bug fixes and new capability
- Configure – set up your environment specifications
- Build – compile and link
- Install – copy files to target directories
- Package – bundle files for installation



Using sstate cache

- The sstate cache contains the results of each build step
- With sstate cache the build can skip directly to image creation step
  - Reduces build times significantly
  - Build to generate sstate cache took ~4h



Picture: https://docs.yoctoproject.org/singleindex.html

Executing the builds

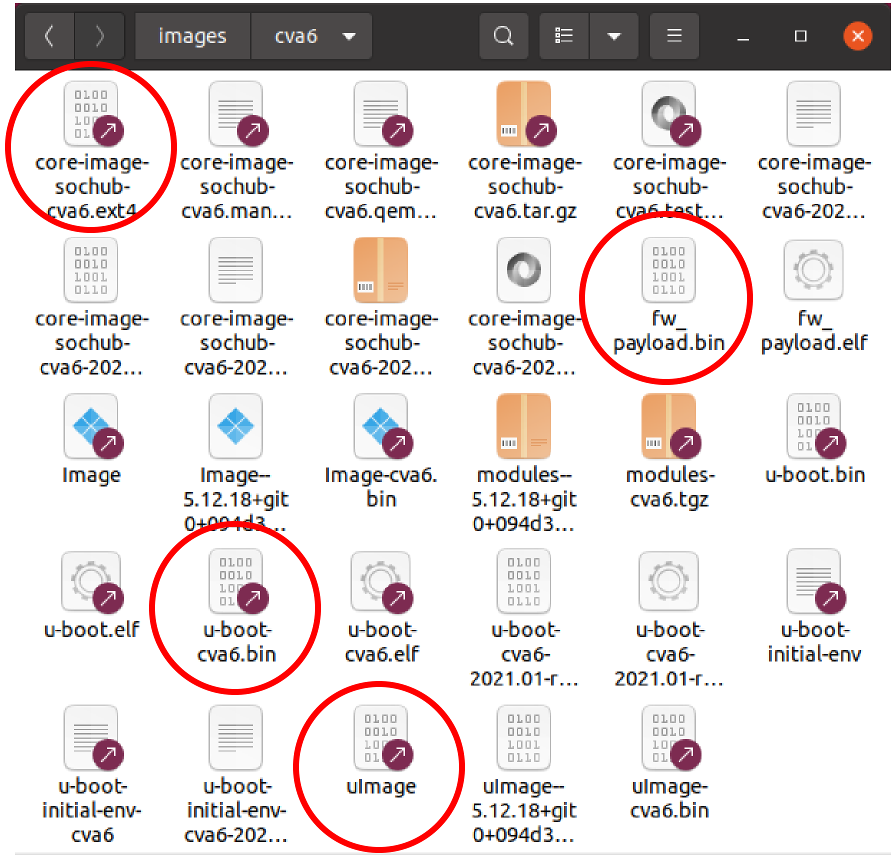
- Setting up the environment
  - New directories added to PATH etc.
- Building image and SDK with BitBake
- Builds will take ~20min

```
$ source envsetup.sh

$ printenv

$ bitbake core-image-sochub

$ bitbake -c populate_sdk core-
image-sochub
```

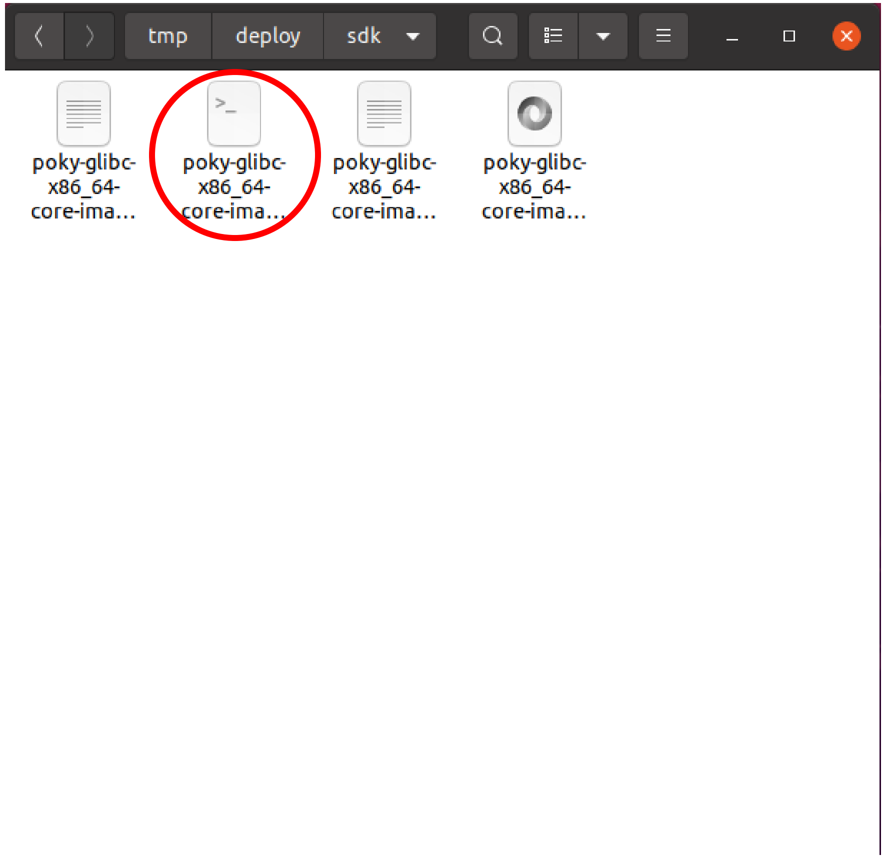
Running custom Linux in QEMU

Build results

- Image files available in `build/tmp/deploy/images/cva6`



- SDK files available in `build/tmp/deploy/sdk`



Emulator environment QEMU

- QEMU is a generic and open source machine emulator (and virtualizer)
- Allows to execute RISC-V operating system and programs on x86 machine
  - Dynamic translation is used to get good performance
- Emulates every aspect of a working hardware environment
- Emulates a full-featured RISC-V computer
  - Usually virtio hardware interface is used
- Yocto has automatically built this for us

Running the finished image

- Yocto provides wrapper script for convenient QEMU usage
  - Used options defined in a configuration file
- Usable RISC-V Linux environment for general application development
- Image configured with debug settings
  - username: root
  - password empty

```
$ runqemu slirp nographic

 (Linux boot output)


cva6 login: root

root@cva6:~# uname -a

root@cva6:~# poweroff
```

Discussion with audience: Hardware vs. QEMU

Differences and similarities?

Discussion with audience: Hardware vs. QEMU

- Different boot setup
- Processor speed
- Hardware drivers
- Both support Machine, Supervisor and User privilege levels
- Both implement memory protection with MMU
- Different ISA extensions available
  - CVA6: RV64I (64-bit base instruction set), M (Integer multiply and divide), A (Atomic instructions), C (Compressed instructions)
  - QEMURISCV64: RV64I, M, A, C, F (Single precision floating point), D (Double precision FP)

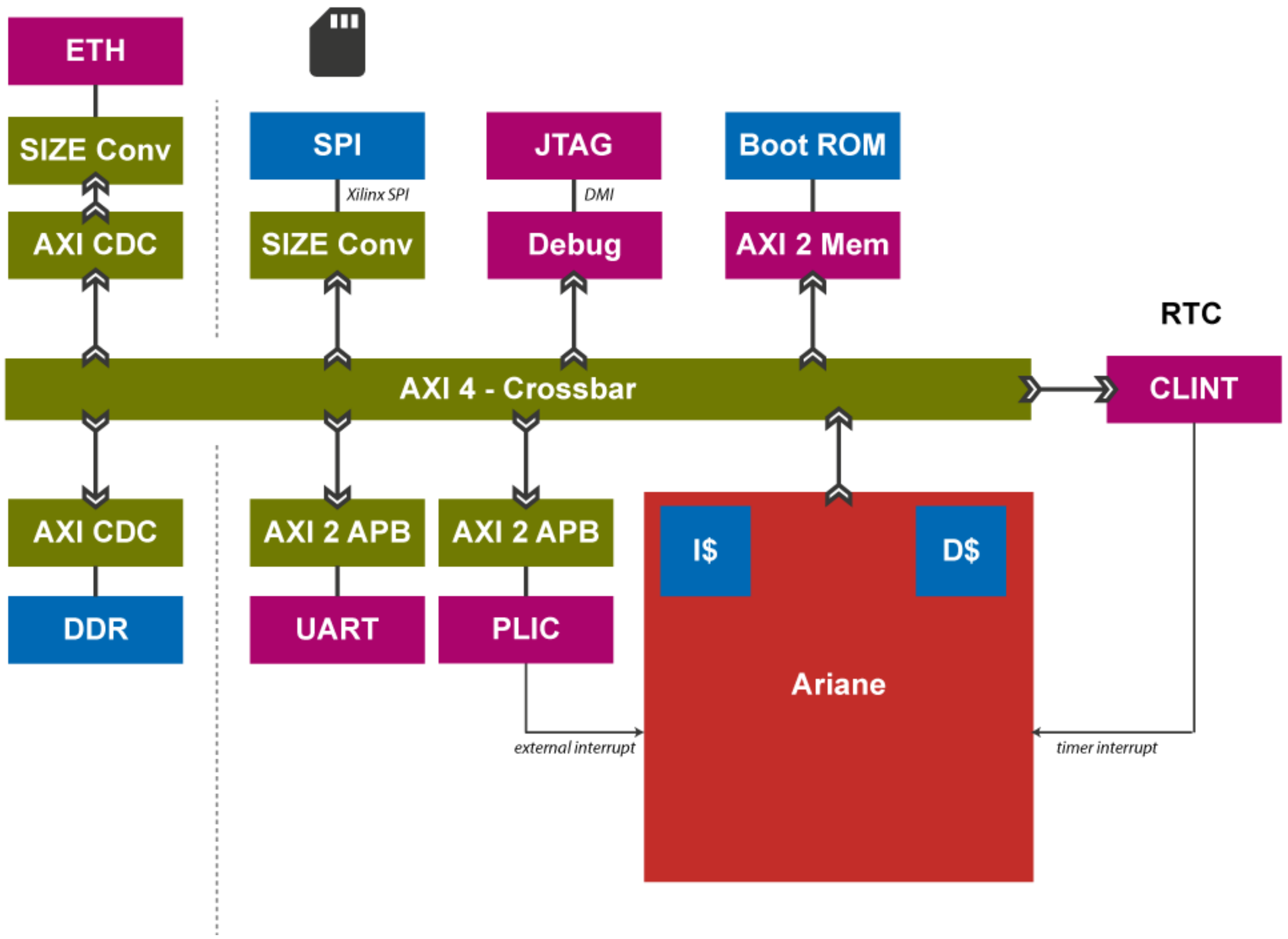
Demo

Genesys 2 with CVA6 core

- Genesys 2 FPGA platform
- SoC Hub chips will have multiple CVA6 cores identical to the one running on FPGA
- The kernel and rest of the OS running in QEMU also run unmodified on CVA6

CVA6

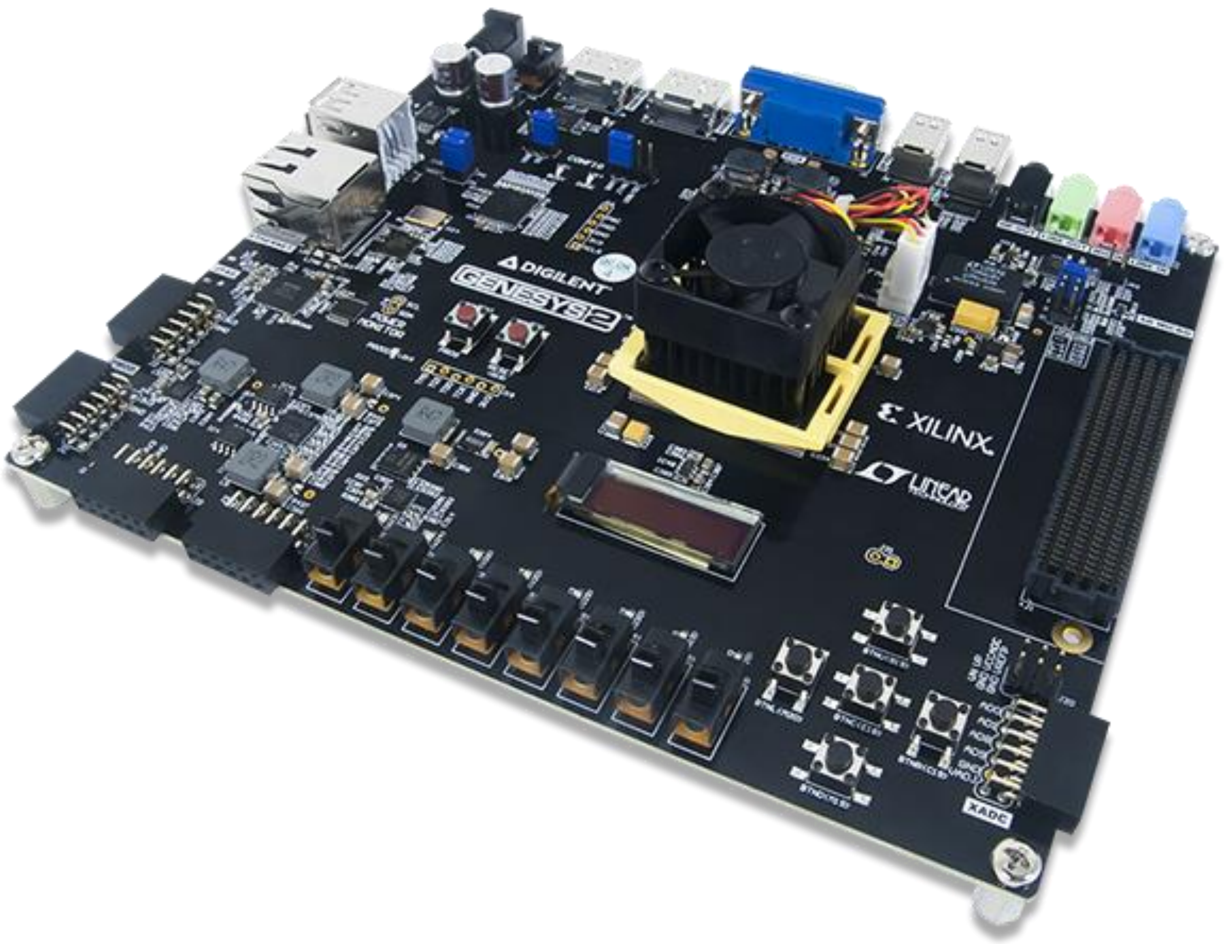
- CPU implementation that uses the RISC-V ISA
- Sufficient to run Linux
- CVA6 peripherals are supported by Linux
  - 16750 UART for serial
  - Xilinx SPI for SD card
- U-Boot also supports both after fixing some driver oversights



Picture: https://github.com/openhwgroup/cva6

Genesys 2

- Kintex-7 FPGA Development Board
- Chosen because CVA6 offers out of the box support for Genesys 2
  - Prebuilt bitstream



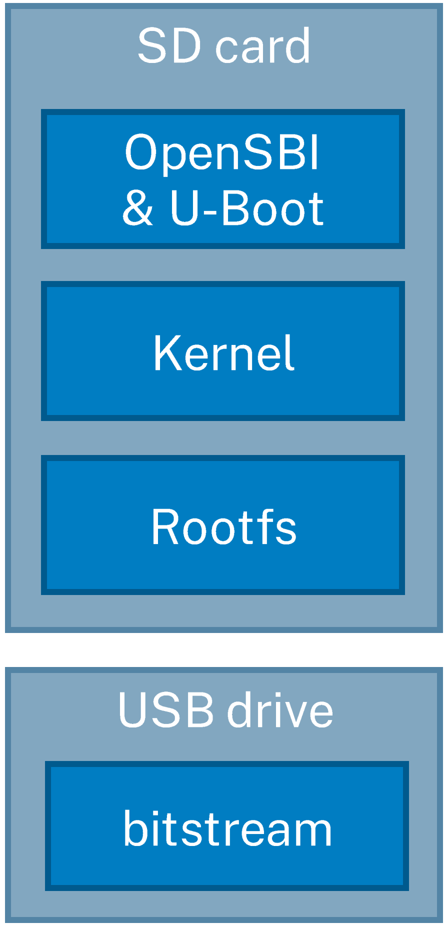
Picture: https://digilent.com/reference/programmable-logic/genesys-2/start

CVA6 in Yocto

- Supported by meta-openhw BSP layer in Yocto
  - CVA6 support added by `cva6.conf` machine configuration file
- Specific patches and configs for bootloader
- Standard mainline Linux
  - Kernel configured to include UART and SPI/SD drivers for CVA6
- Other miscellaneous configurations

Setting up the board (and Genesys 2 kit)

- Bitstream from CVA6 release onto a USB drive
  - Partition 1: FAT, **`ariane_xilinx.bit`** bitstream
- An SD card is used for mass storage
  - Partition 1: raw, **`fw_payload.bin`** bootloader image
  - Partition 2: ext4, **`uImage`** compressed kernel
  - Partition 3: ext4, **`core-image-sochub-cva6.ext4`** root filesystem



RISC-V boot process

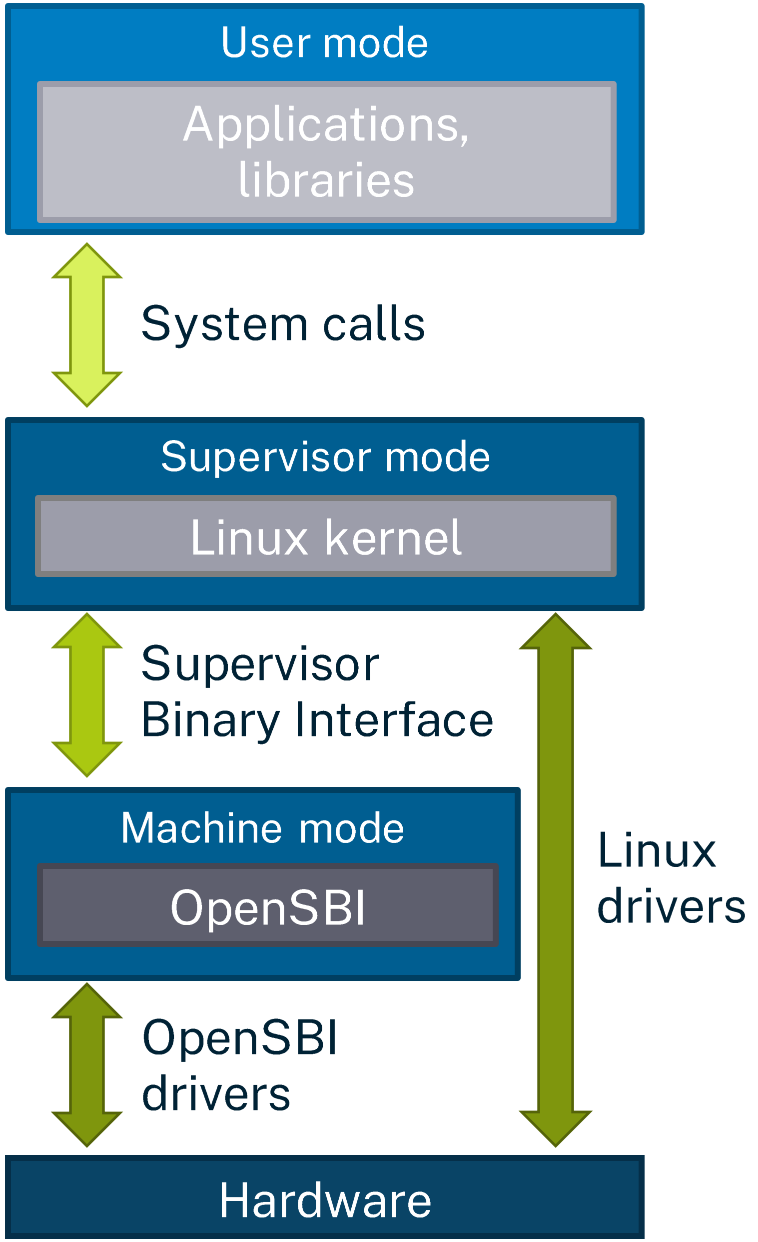
Booting on the board

- Boot ROM → OpenSBI → U-Boot → Linux
  - No FSBL in this case
- Extremely slow because SD access is slow on FPGA

```
U-Boot 2021.01 (Jan 11 2021 - 18:11:43 +0000)

CPU:   rv64imafdc
Model: eth,ariane-bare
DRAM:  1 GiB
MMC:   xps-spi@20000000:mmc@0: 0
In:    uart@10000000
Out:   uart@10000000
Err:   uart@10000000
Net:   No ethernet found.
Hit any key to stop autoboot:  0
4729011 bytes read in 292020 ms (15.6 KiB/s)
## Booting kernel from Legacy Image at 83200000 ...
   Image Name:   Poky (Yocto Project Reference Di
   Image Type:   RISC-V Linux Kernel Image (gzip compressed)
   Data Size:    4728947 Bytes = 4.5 MiB
   Load Address: 80200000
   Entry Point:  80200000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 82200000
   Booting using the fdt blob at 0x82200000
   Uncompressing Kernel Image
   Using Device Tree in place at 0000000082200000, end 000000008220400f

Starting kernel ...

[    0.000000] Linux version 5.12.18-sochub (oe-user@oe-host) (riscv64-poky-linux-gcc (GCC) 10.2.0,
GNU ld (GNU Binutils) 2.36.1.20210209) #1 SMP Mon Jul 19 08:01:28 UTC 2021
```

RISC-V boot process terminology

- A RISC-V CPU boots in M-mode (machine mode), while kernel runs in S-mode (supervisor mode) and userspace in U-mode (user mode)
- M-mode software remains in memory after bootloader drops its privilege level to S-mode, providing basic services similar to BIOS interrupts on x86
- Kernel can call M-mode services via Supervisor Binary Interface (SBI) which is usually implemented by OpenSBI



RISC-V boot sequence for Linux

- Zero-Stage Bootloader (ZSBL) on CPU ROM loads FSBL (First Stage Bootloader), from SD card on CVA6
- FSBL loads OpenSBI and U-Boot, OpenSBI stays in the background to provide M-mode services
  - OpenSBI then runs U-Boot in S-mode
- U-Boot loads Linux kernel from SD and boots it, staying in S-mode
- Linux kernel executes init from root filesystem and runs it as the first userspace process, in U-mode

Developing C applications for RISC-V

Yocto SDK

- Running C applications in RISC-V
- Yocto SDK provides tools required for cross-compiling to target system
  - Compiler, linker etc.
  - Libraries
  - Debuggers
  - QEMU
  - Environment variables
  - Other commonly used tools
- Contents of the SDK is completely configurable

Installing and using the SDK

- After populate_sdk stage, a toolchain script has been generated
  - Executing it installs the SDK, by default under /opt
- To use SDK, source the script from /opt
  - This will initialize the SDK in your current terminal

```
(In a new terminal)

$ cd build/tmp/deploy/sdk

$ ./poky-glibc-x86_64-core-image-sochub-riscv64-cva6-toolchain-3.3.2.sh

$ source /opt/poky/3.3.2/environment-setup-riscv64-poky-linux

$ printenv
```

Compiling the application

- Cross-compiler produces binaries for RISC-V ISA
- SDK sysroot is used during the compilation

main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

(In the SDK terminal)

```
$ cd application

$ echo $CC

$ $CC $CFLAGS $LDFLAGS main.c

$ ./a.out

bash: ./a.out: cannot execute binary
file: Exec format error

$ $OBJDUMP -f a.out
```

Running the application

- Executable needs to be copied into rootfs
- Executing the binary is now possible inside QEMU

```
$ cd build/tmp/deploy/images/cva6

$ sudo mount core-image-sochub-cva6.ext4 /mnt

$ sudo cp a.out /mnt/home/root

$ umount /mnt

$ runqemu slirp nographic

 (…)

root@cva6:~# ./a.out
```

Creating custom recipes

Same using recipes

- Instructions for BitBake to execute previous manual steps
  - Like any other recipe
- The recipe is located in `meta-sochub/recipes-app/hello`
- `main.c` exactly the same as before
- Needs to be added to image using `IMAGE_INSTALL`
  - Finished image will contain the hello binary

```
meta-sochub/
└── recipes-app
    └── hello
        ├── files
        │   └── main.c
        └── hello_1.0.bb
```

hello_1.0.bb

meta-sochub > recipes-app > hello > hello_1.0.bb

```
SUMMARY = "Hello World application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "\
    file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://main.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} main.c -o hello
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 hello ${D}${bindir}
}

```

Running the executable

- The executable was is actually already installed to the rootfs
- Executed just as any other command
  - Located in `/usr/bin`

```
$ runqemu slirp nographic
 (…)
root@cva6:~# which hello
root@cva6:~# hello
```

Using recipes vs. manual approach with SDK

- Both accomplish the same end result
  - Matter of preference
- However, there are some typical procedures
  - Standard tools, libraries and drivers installed using recipes
  - Custom applications developed and installed using the SDK flow

SoC Hub chip with Linux

Running Linux on the actual SoC Hub chip

- More complicated boot process
- Changes required to low level software
  - OpenSBI
  - U-Boot
  - Linux kernel
- Different peripherals that may not have mainline support
  - Device drivers
  - Devicetree
- Yocto support

Conclusions

Conclusions

- Yocto can be used to build custom Linux distributions
- RISC-V and Linux is already a viable setup
  - Some extra effort may still be required
- No differences to high-level application development
  - Tools handle different ISAs for us
  - Good thing!
- SoC Hub's chip will require some more development work to port Linux

Questions?

CREATING A SMARTER FUTURE TODAY

Visit **wapice.com**