Lecture 4: Linux OS Creation over MPSoC

(Hardware Accelerated Memory Copy System)

Thor Kamp Opstrup,
University of Southern Denmark

Slides by: Frederik Falk Nyboe

Agenda

- Theory (~1 hour)
  - The Xilinx Multi-Processing System-on-Chip (MPSoC)
  - MPSoC development methodology
    - Vivado and Vitis
    - Petalinux
  - Embedded Linux and Yocto Linux
    - Kernel drivers
    - Device tree
  - Ultra96-V2
  - MPSoC4Drones
- Exercise (~2.5 hours + homework)
  - FPGA design in Vivado
  - Bare-metal testing in Vitis
  - OS build with Petalinux and MPSoC4Drones
  - Booting and connecting
  - Building Linux test applications

The Typical Robotics Computational Setup

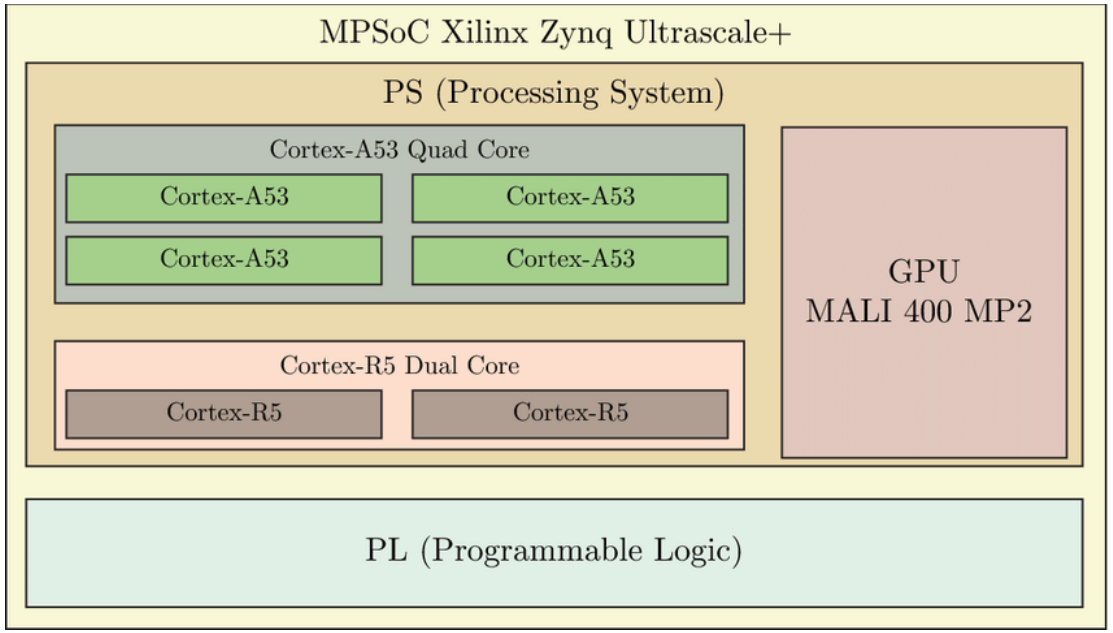
- Raspberry Pi or similar low-end SBC
  - Low computational capacity
  - Low power consumption
  - Drone applications
- Intel NUC or similar higher-end SBC
  - Higher computational capacity
  - Advanced drone applications and robotic applications
- Nvidia Jetson or similar GPU-based SBC
  - Specific computational capacity for AI
  - High power consumption
  - AI-reliant robotics and drone applications

The Typical Robotics Computational Setup

- Linux-based OS
- Robot Operating System (ROS)
- *Low level of flexibility for digital circuitry*
- *Limited computational capacity*

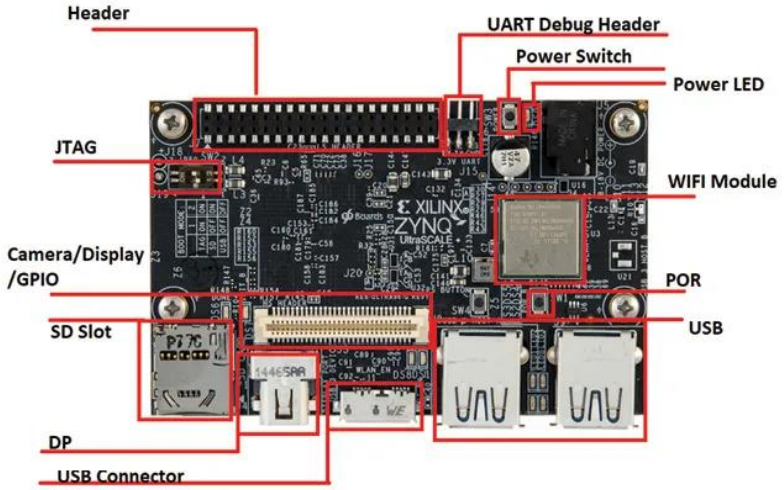
The Xilinx Multi-Processing System-on-Chip (MPSoC)

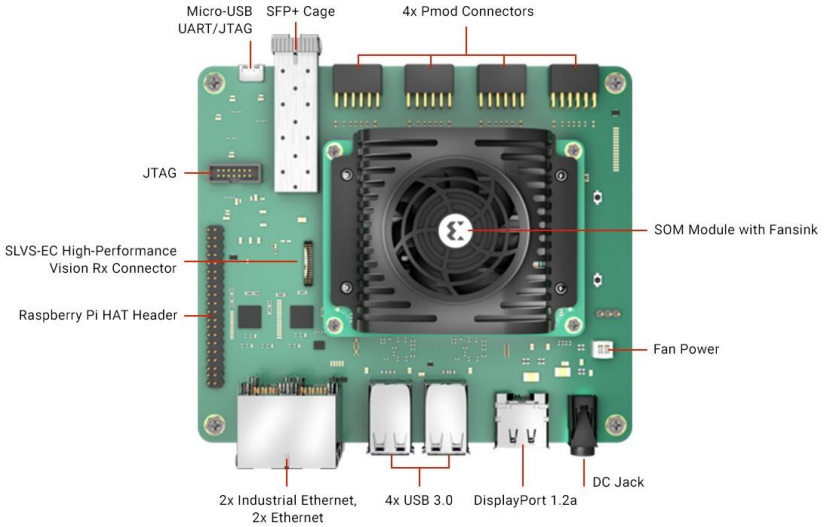
- Novel chip from Xilinx
  - Large commercial push for robotics applications
- Quad-core application processing unit (APU)
  - Good for running Linux with ROS
- Dual-core real-time processing unit (RPU)
  - Good for running a real-time operating system such as FreeRTOS
- FPGA-fabric / programmable logic (PL)
  - Good for custom digital circuitry and hardware acceleration kernels

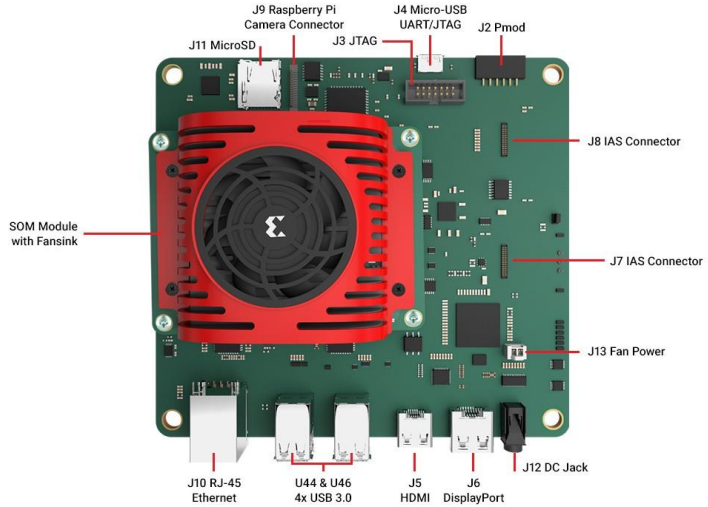


The Xilinx Multi-Processing System-on-Chip (MPSoC)

- Novel chip from Xilinx
  - Large commercial push for robotics applications



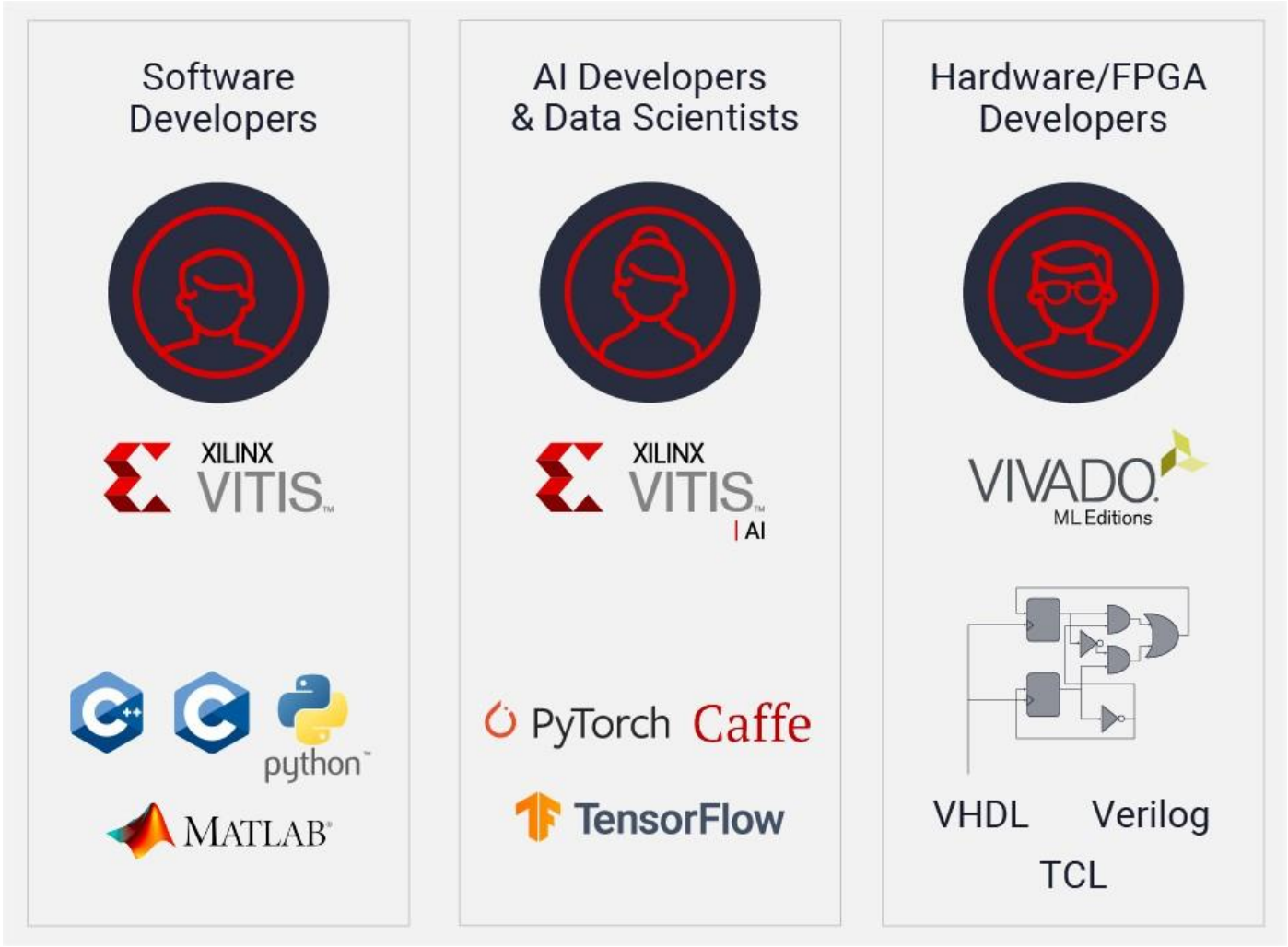




MPSoC Development Methodology

- Bottom-up design - traditional method
  - Hardware designers build FPGA system
  - Embedded engineers integrate and build OS
  - Application developer write higher-level software
  - High level of control, slow design process
- Top-down design – newer method
  - Specify algorithms in a high-level language
  - Use high-level synthesis tools to synthesize FPGA implementations of algorithms to be accelerated
  - Hardware-layer is abstracted away
  - Low level of control, fast design process

MPSoC Development Methodology

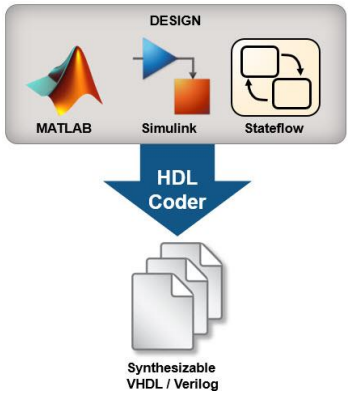


MPSoC Development Methodology

- Hardware-software co-design
  - Build some custom circuitry for the FPGA in a hardware-description language
  - Build some accelerator kernels for the FPGA using high-level synthesis
  - Requires full-stack embedded skills
  - High flexibility, moderate speed design process
- Choice of workflow depends on application requirements

MPSoC Development Toolchain

- Vivado
  - Register-transfer-level FPGA design
  - FPGA system integration
- Vitis HLS, FINN, MATLAB…
  - Algorithm-level FPGA design
- Vitis
  - Firmware development for RPU
  - Bare-metal system testing
- Petalinux
  - Linux OS development

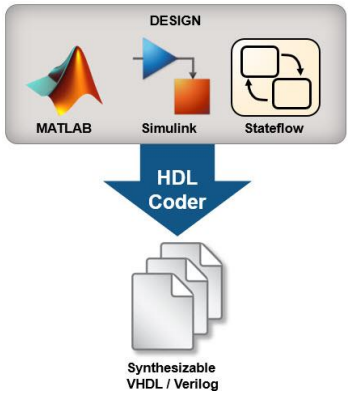


MPSoC Development Toolchain

- Vivado
  - Register-transfer-level FPGA design
  - FPGA system inte
- Vitis HLS, FINN, MAT
  - Algorithm-level F
- Vitis
  - Firmware develo
  - Bare-metal syste
- Petalinux
  - Linux OS development

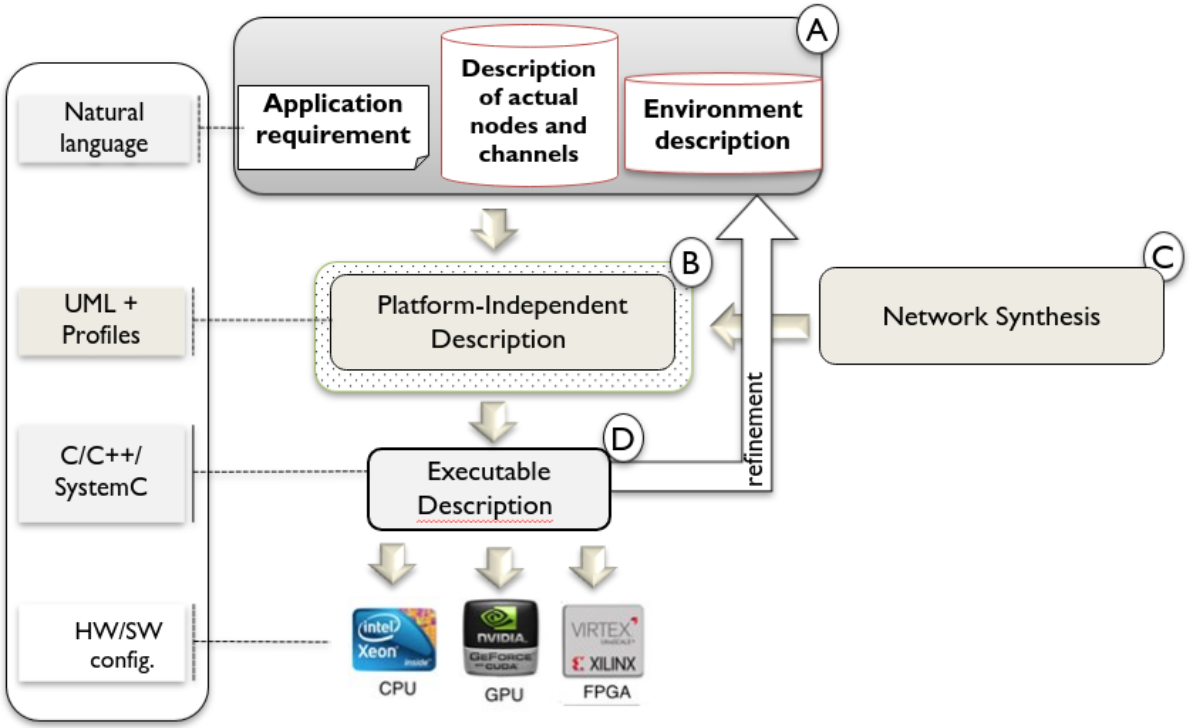
VIVADO™

FINN



MPSoC Development Methodology

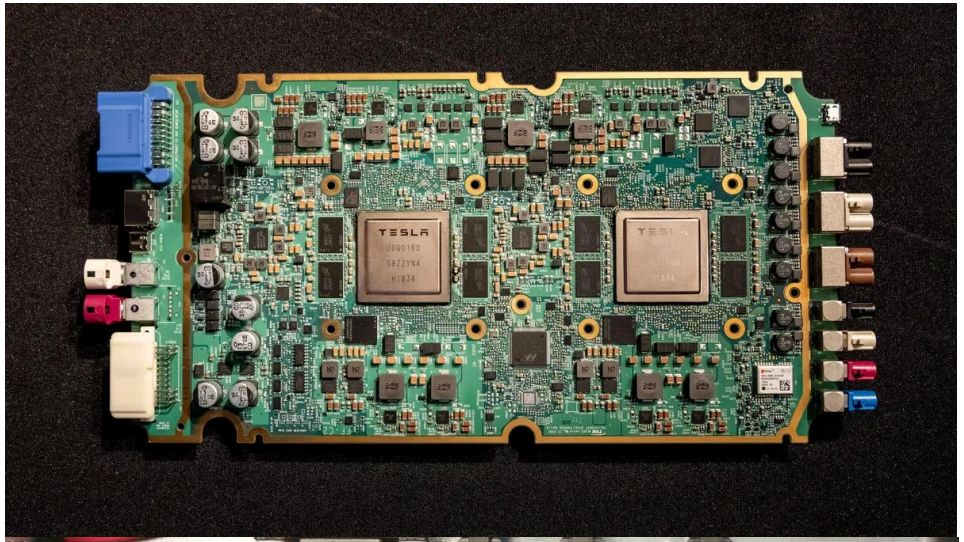
- The high-level methods only abstracts away the hardware layer
- In any methodology, the software needs to be aware of the hardware, not the other way around
- FPGA modules are connected to AXI
  - The registers of the FPGA module will map into the physical memory of the CPU
- The FPGA modules are interfaced from the CPU by reading from and writing to the corresponding addresses in memory
- *Hardware comes first* (in a build)

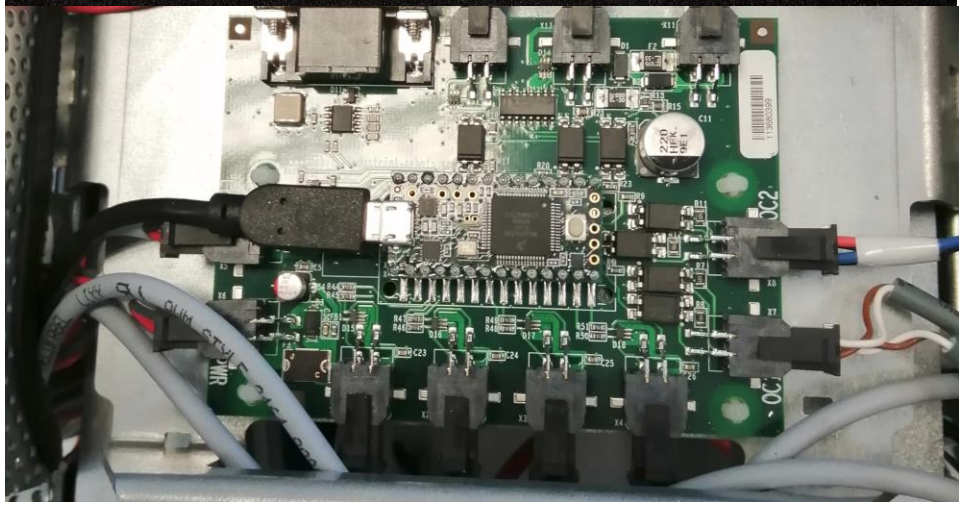


E. Ebeid et. "Model-Driven Design of Network Aspects of Distributed Embedded Systems," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Embedded Linux

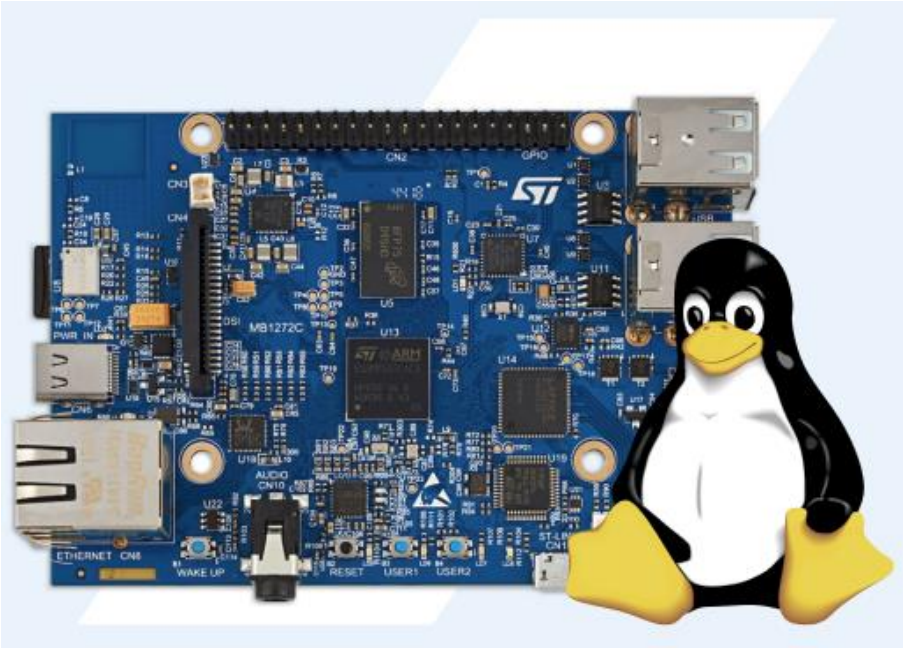
- Linux running on some sort of embedded processor
- Why use embedded Linux?
  - Networking
  - High-level programming
  - Modularity
  - USB
  - Community and open-source
  - …
  - ROS





Embedded Linux

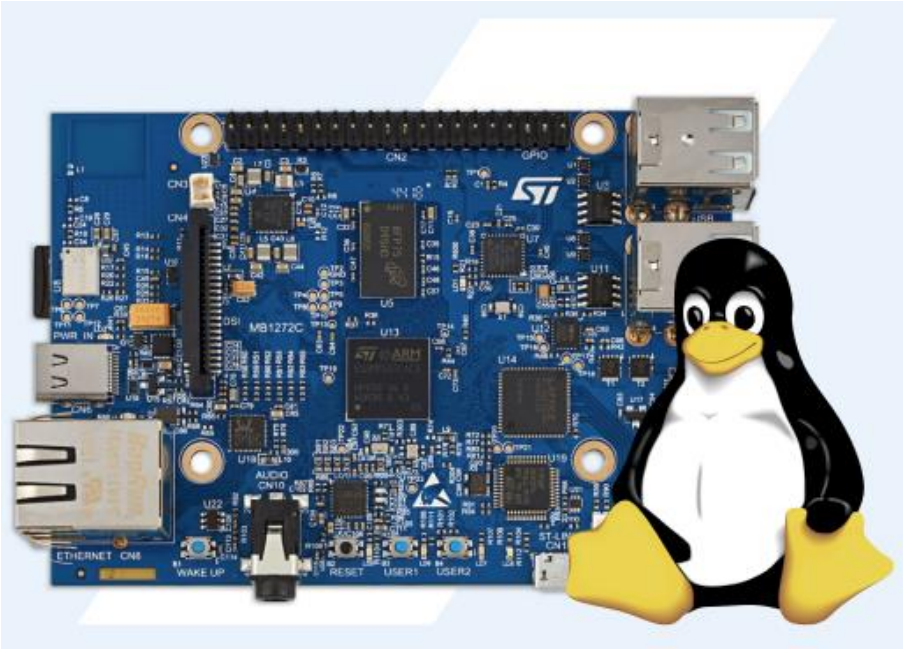
- Very dependent on the hardware
  - Embedded hardware is by definition very different and specific to the application
  - Different processors, different peripherals, different memory
  - Different intra-chip features such as DMA, MMU, instruction cache, etc.
- The requirement for building custom Linux distributions led to tools such as Yocto Project



yocto PROJECT

Yocto Linux

- ”The Yocto Project. It's not an embedded Linux Distribution, It creates a custom one for you.”
- Deploys a software-like modular layered approach to building Linux distributions
- The layers can define anything in the build: Kernel modules, hardware interfaces, applications, packages...
  - The chip manufacturer supplies some chip-specific layers
  - The board manufacturer the adds some board-specific layers
  - The end-developer tweaks and adds final layers



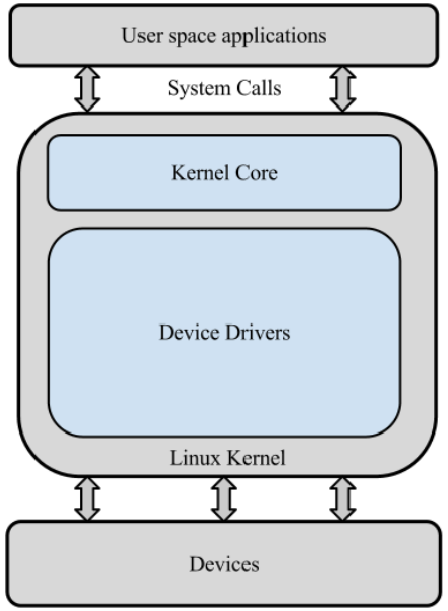
yocto PROJECT

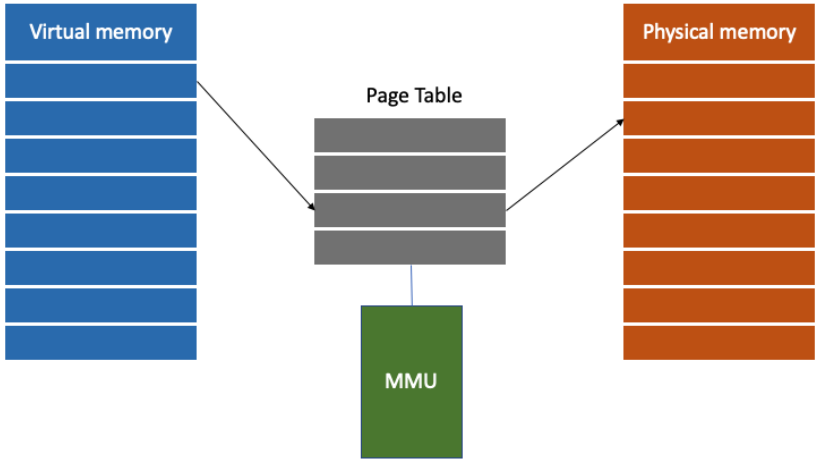
Yocto Linux

- ”The Yocto Project. It's not an embedded Linux Distribution, It creates a custom one for you.”
- Deploys a software-like modular layered approach to building Linux distributions
- The layers can define anything in the build: Kernel modules, hardware interfaces, applications, packages...
  - The chip manufacturer supplies some chip-specific layers
  - The board manufacturer the adds some board-specific layers
  - The end-developer tweaks and adds final layers
- Petalinux is just a wrapper for Yocto Linux
  - Exposes a few handy commands
  - Adds functionality to read a Vivado hardware specification file (.xsa)
  - Adds some drivers and packages for binding custom FPGA hardware to the Linux kernel

The Linux OS

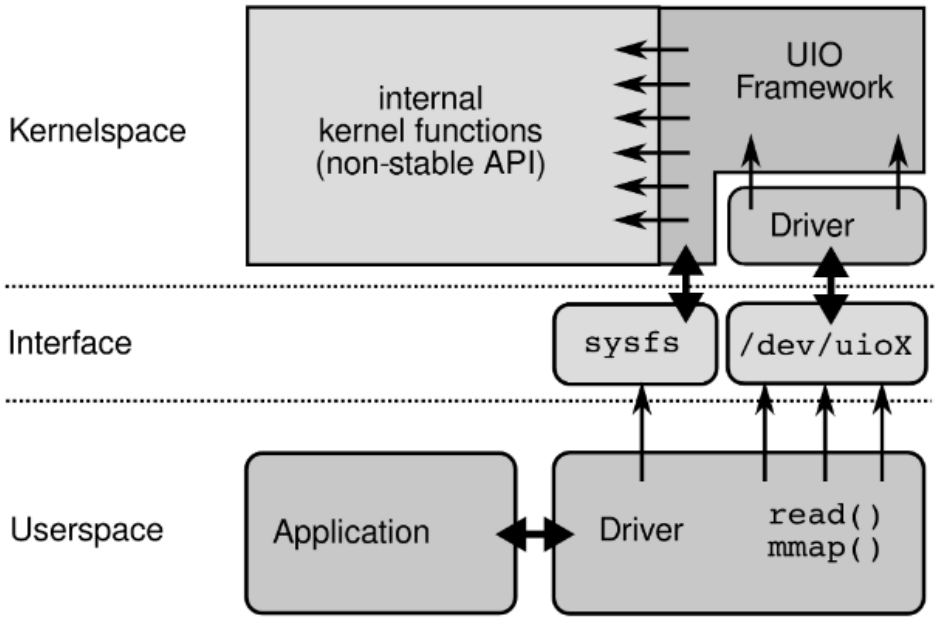
- Two spaces:
  - User space is where our application processes are running – restricted access to memory
  - Kernel space is where the kernel is running along side kernel drivers – full access to memory
- A memory management unit (MMU) maps between user space virtual addresses and physical addresses
  - A kernel driver is required to access a specific physical memory address from a user space application





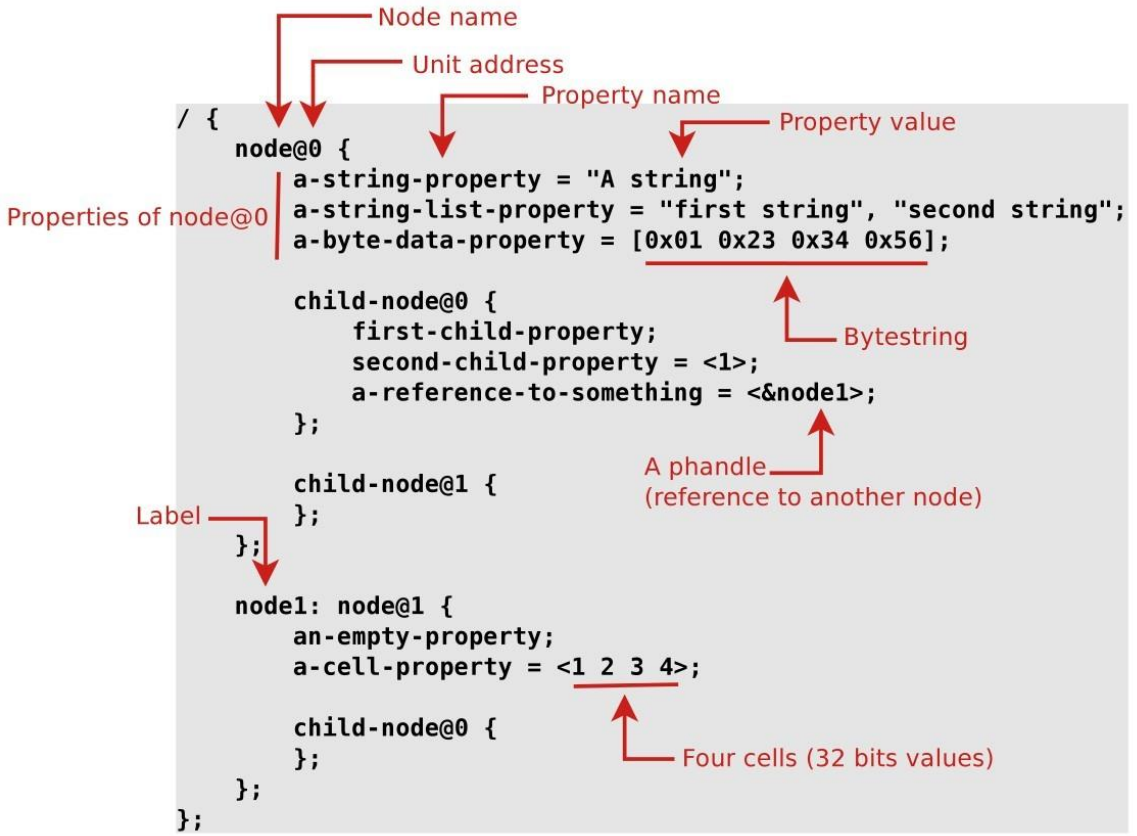
The UIO Kernel Driver

- The Userspace I/O (UIO) kernel driver is a generic driver which can be applied to memory mapped hardware
- A device file (/dev/uioX) is exposed per each memory mapped hardware device
- The device is controlled from user space by reading and writing to this device file



Device Tree

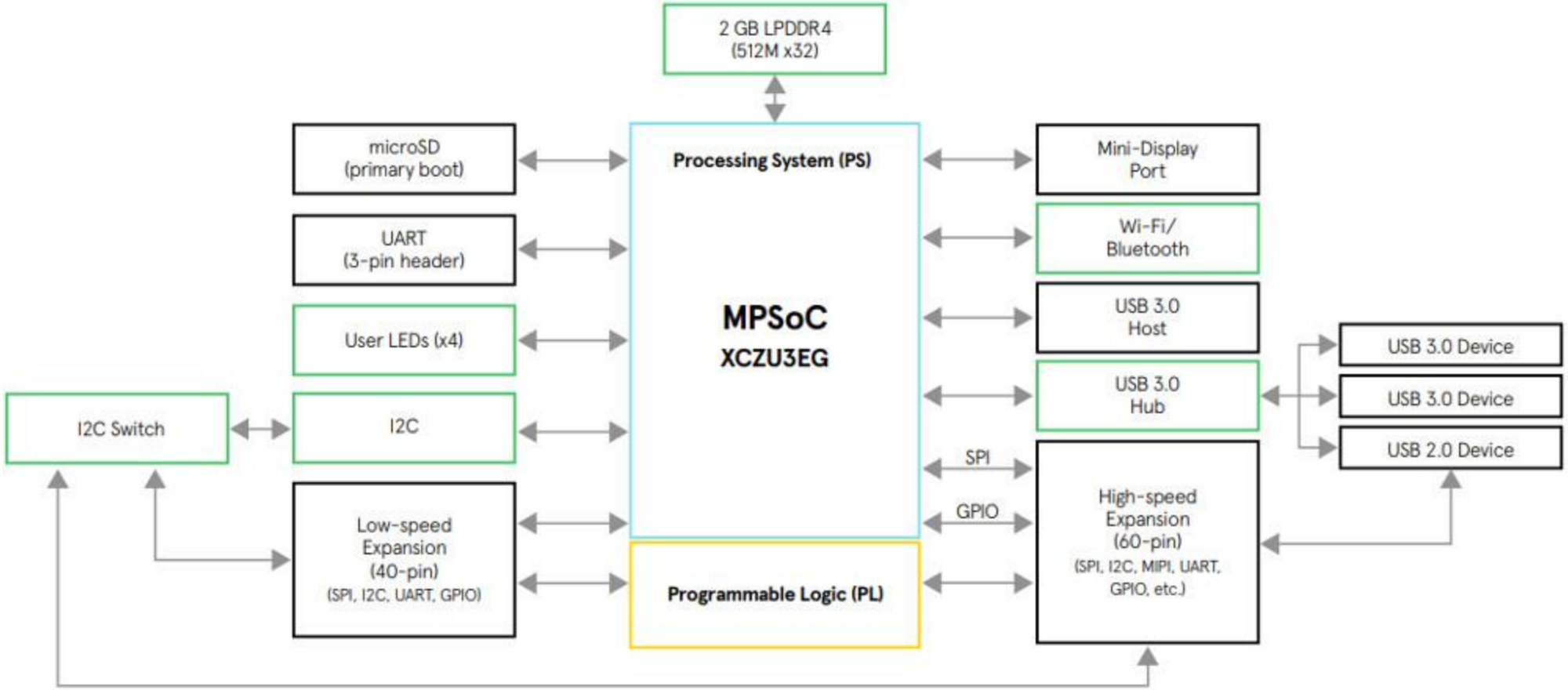
- One or multiple device tree files are used to specify information about the hardware to the Yocto Linux build
- Specifies e.g. how storage is connected, how peripheral devices are connected, etc.
- Uses node-structure syntax
- In Petalinux, it can be used to instruct the kernel to associate the UIO driver with AXI-mapped FPGA modules
- More about this in the exercise



The Ultra96-V2

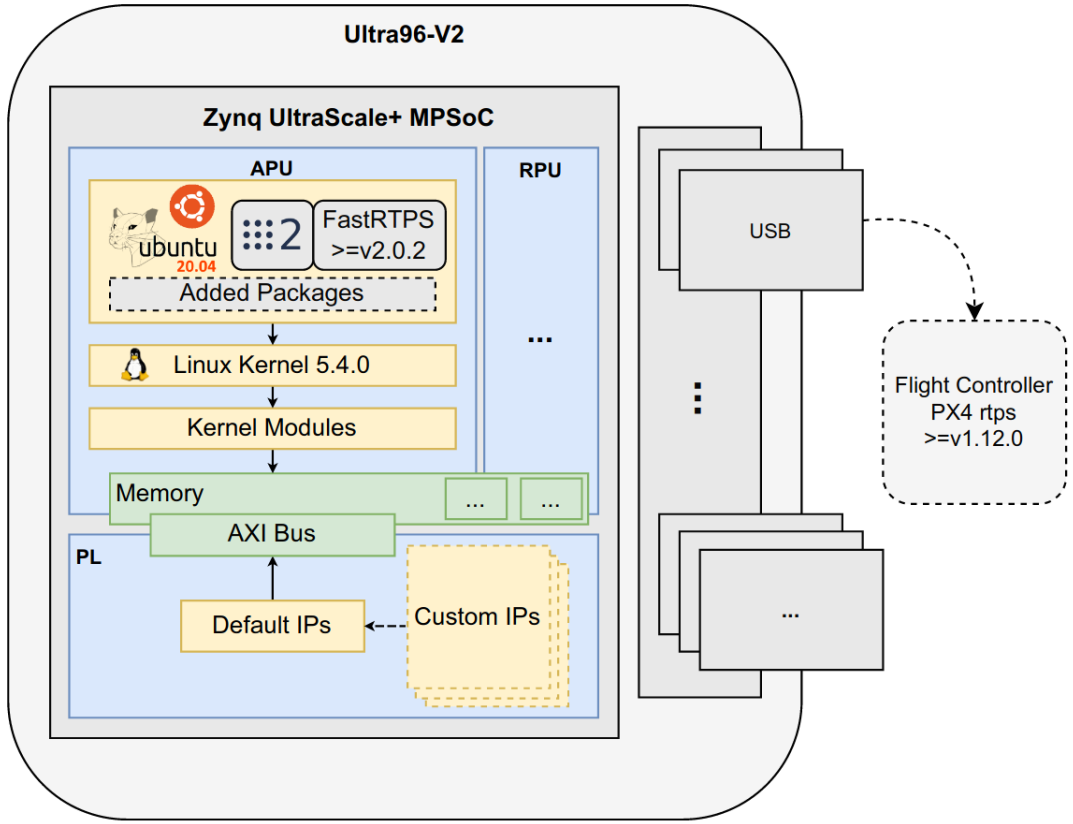
- Nice little Single Board Computer (SBC)
- Lots of pinout
- Two USB3 ports
- WiFi
- Featuring one of the smallest MPSoCs (lowest amount of FPGA fabric)

The Ultra96-V2

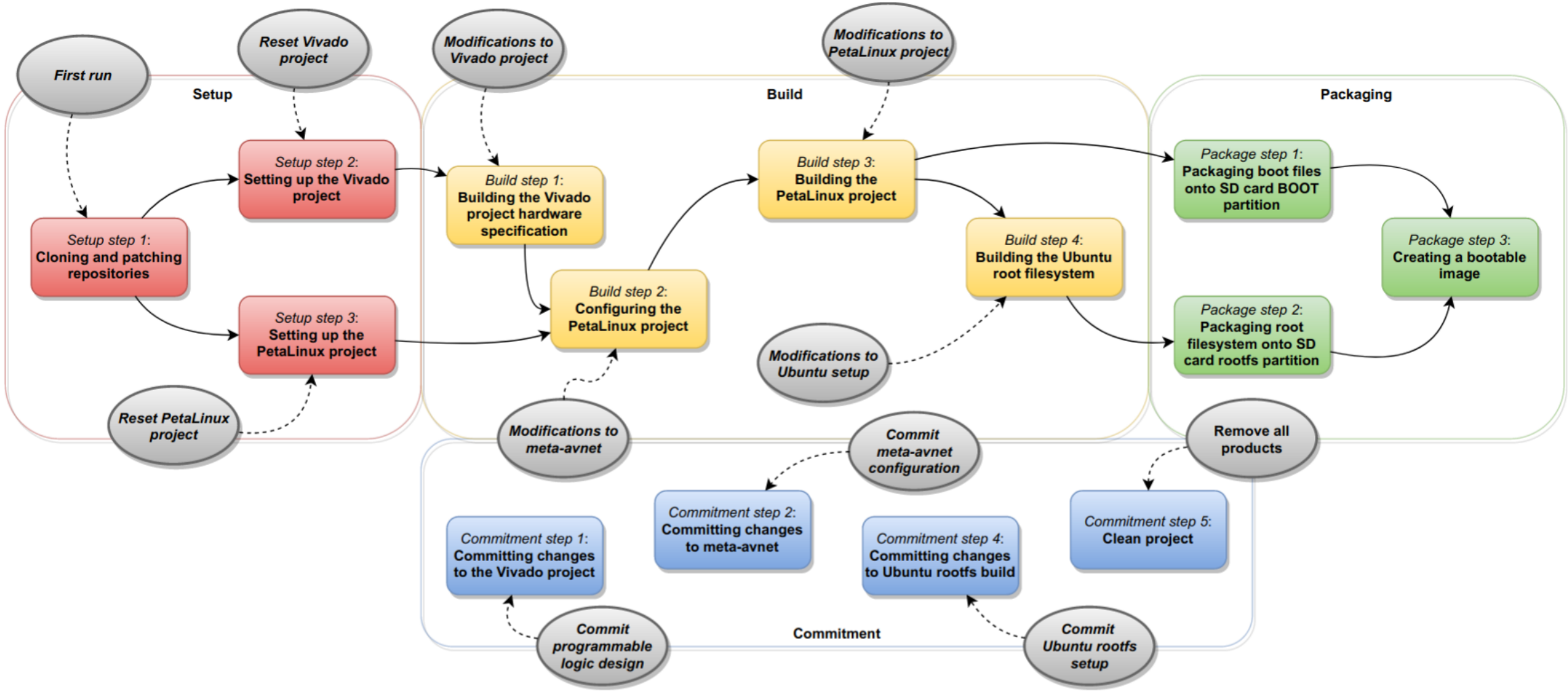


MPSoC4Drones

- Framework and toolset for building OS for MPSoC
- … Works also for non-drone applications
- Integrates custom FPGA design
- Builds an Ubuntu 20.04 rootfs on top of the Yocto kernel
- Integrates ROS2
- Supports iterative workflow
- Automatically integrates Yocto meta-layers from Avnet (Ultra96-V2 manufacturer)



MPSoC4Drones Workflow



Exercise
Hardware Accelerated Memory Copy System

- Demonstrating the full OS build workflow
- Full exercise description in Itslearning