M2PGi – M2M

Pr. Didier Donsez (didier.donsez@imag.fr)

Pr. Olivier Gruber (olivier.gruber@imag.fr)

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

This Course – Two Parts

- Part One – Pr. Olivier Gruber – 50%
  - 5 weeks, widening your horizon as developers
  - Wandering outside of the Linux world
  - Also a primer before part two
- Part Two – Pr. Didier Donsez – 50%
  - 5 weeks, experimenting with an IoT infrastructure
  - Sensors → Gateway → Data-gathering Server

Machine to Machine

Freely-adapted excerpts from Wikipedia:

Machine to machine (M2M) is direct communication between devices using any communications channel, including wired and wireless.

M2M communication can include industrial instrumentation, enabling sensors to communicate the information it records to software systems that can use it.

The Internet of things (IoT) describes the network of physical objects—"things"—that embed sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and software systems over the Internet.

Traditional fields of embedded systems, wired or wireless sensor networks, control systems, automation (including home and building automation), and many others all contribute to enabling the Internet of things.

IoT reference architecture

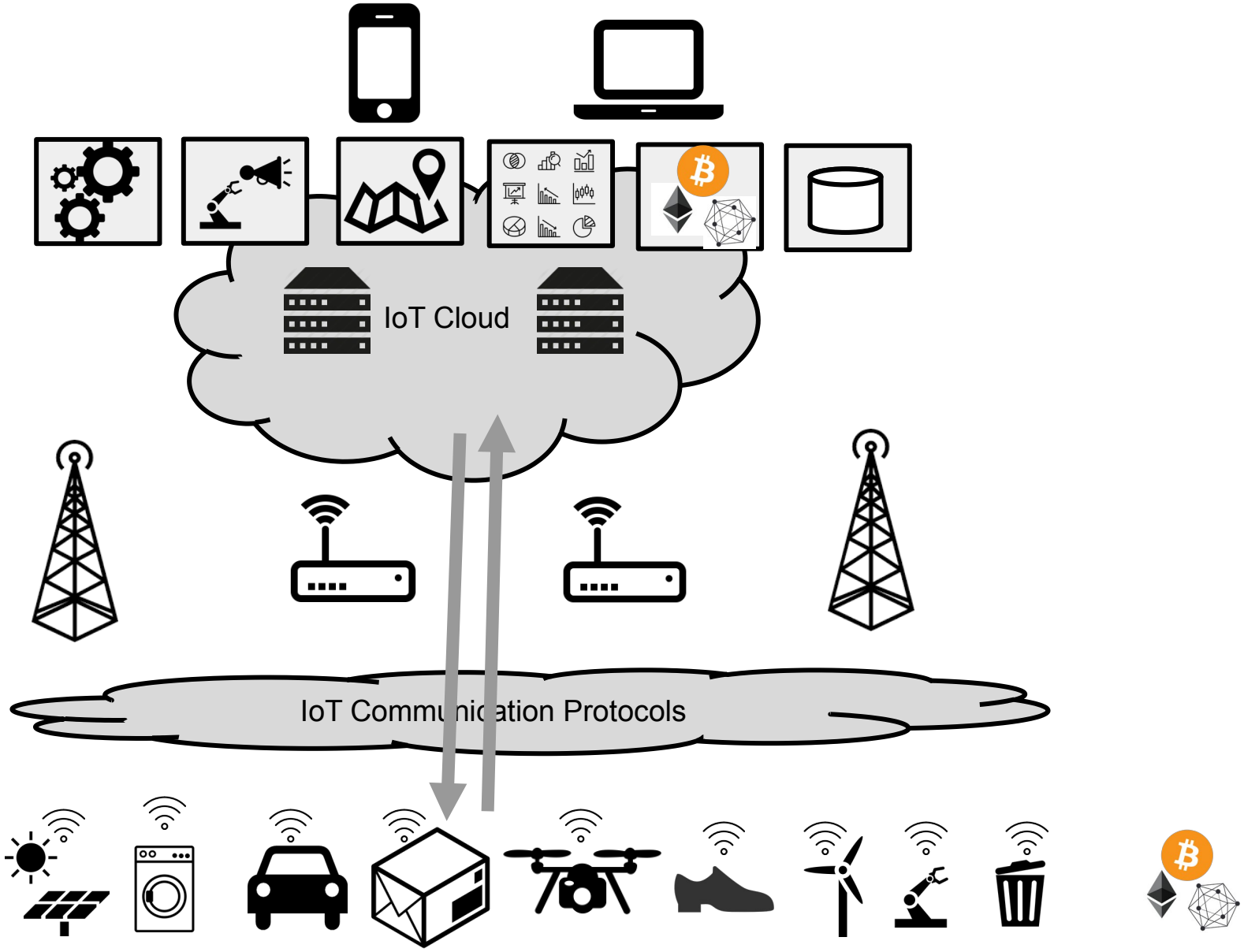
IoT Applications

Cloud infrastructure
(public, private)

Fog/Edge Computing

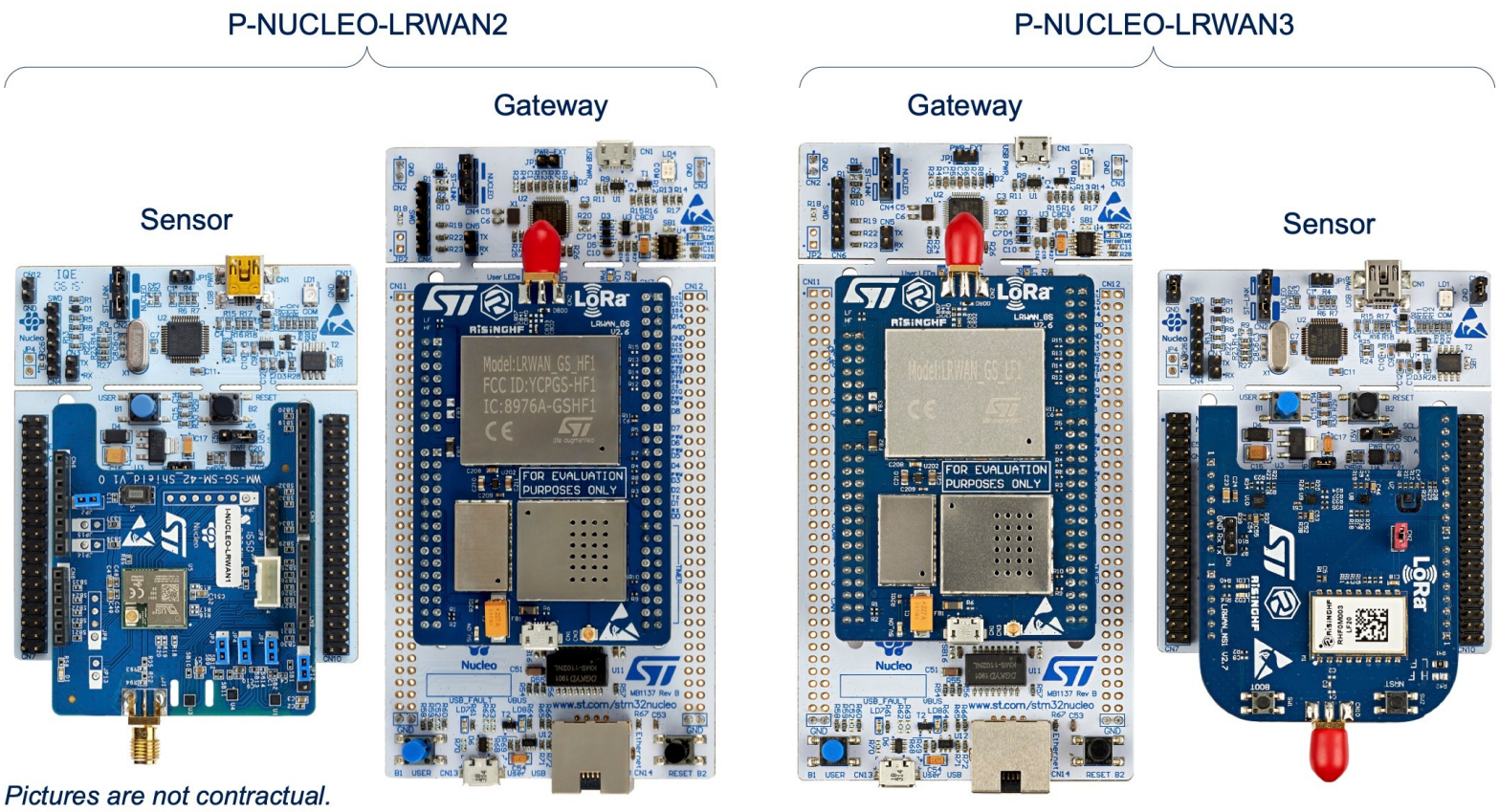
Communications
- wired/wireless
- IP / No IP
- licensed/free bands

Connected Things
(sensors & actuators)



P-Nucleo-LRWAN Starter Packs

ToDo: Pick up your P-Nucleo-LRWAN2 at the fablab, after January 30th, but before the lectures start with Pr. Didier Donsez.



Pictures are not contractual.

Software – RIOT

The friendly operating system for the internet of things.

RIOT is free open-source, developed by a grassroots community

Comparison of Current Operating Systems

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	< 2kB	< 30kB	●	✗	●	✓	●	●
Tiny OS	< 1kB	< 4KB	✗	✗	●	✓	✗	✗
Linux	~ 1MB	~ 1MB	✓	✓	✓	✗	●	●
RIOT	~ 1.5kB	~ 5kB	✓	✓	✓	✓	✓	✓

Full support ✓
Partial support ●
No support ✗

This Course – Two Parts

- Part One – Pr. Olivier Gruber – 50%
  - 5 weeks, widening your horizon as developers
  - Wandering outside of the Linux world
  - Also a primer before part two
- Part Two – Pr. Didier Donsez – 50%
  - 5 weeks, experimenting with an IoT infrastructure
  - Sensors → Gateway → Data-gathering Server

First Part – Overview

- Widening your horizon as a developer
  - What you probably know about...

Application programming, right?

C, Java, JavaScript, Python...

Basic usage of your operating system, right?

File system and Graphical User Interface
Maybe a hint of shell usage/scripting...

How do we program these?

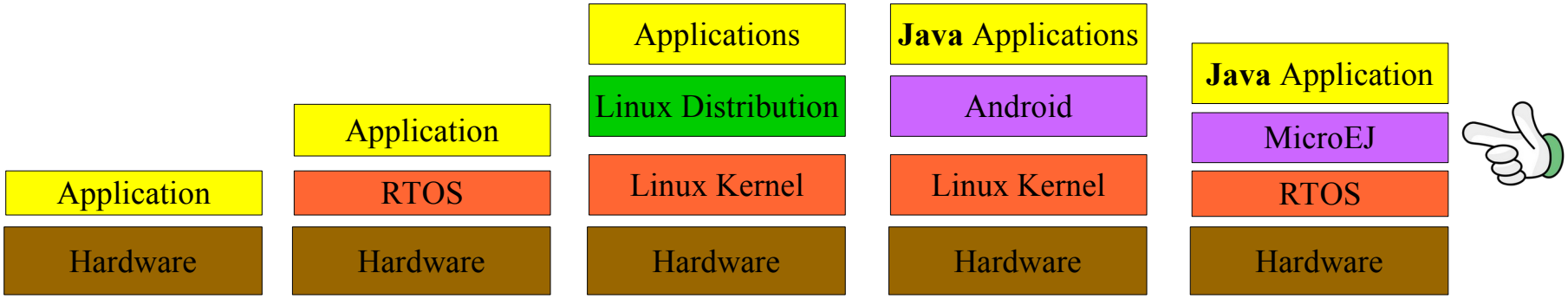
Which tools do we use?

First Part – Overview

Bare-metal programming
- Low-level C programming
- Lucky if using an Hardware Abstraction Layer (HAL)
- Otherwise needs some assembly language programming

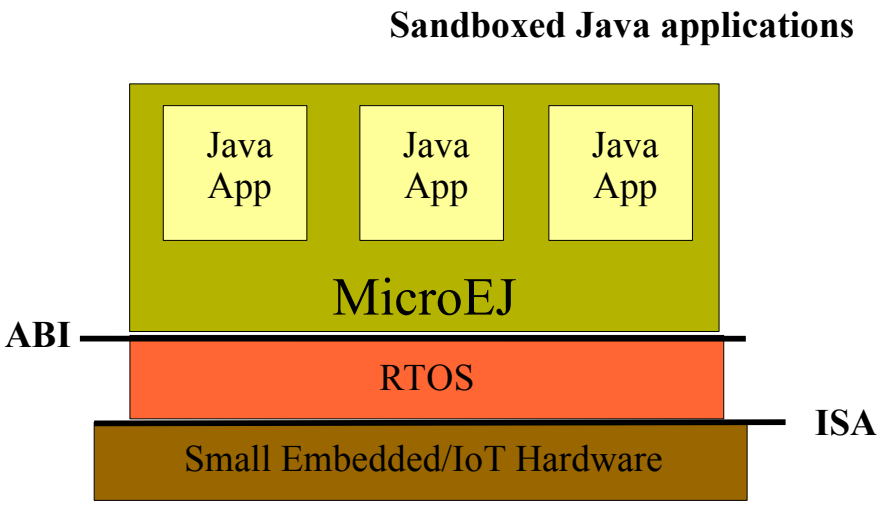
Embedded programming (real-time or specialized)
- Higher-level programming… but not the usual Linux
- Timing-aware programming...

Java programming
- Increasingly more Java in the embedded world…
- Not convinced? What about Android?



French Tech – MicroEJ

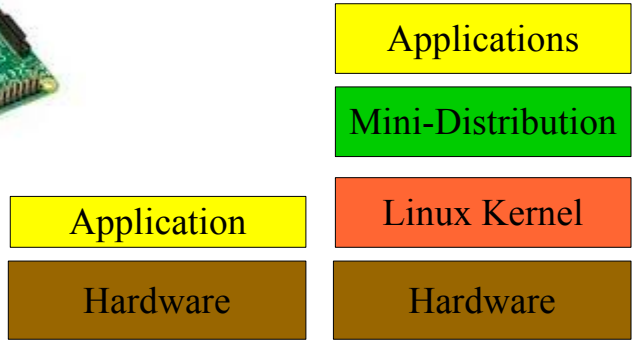
- Java for small and smart IoT devices
  - Similar virtualization technology as Android but *tighter* implementation
    - Android: $15 processor, 32MB of memory
    - MicroEJ: $1 processor, 128KB of memory
  - Google Cloud IoT Partners
    - IoT solutions that leverage Google's secure, global, and scalable infrastructure



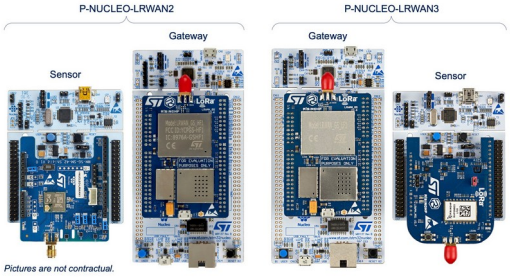
This Course Summary

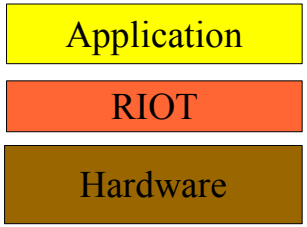
Part-I Bare-metal programming
- Bare-metal programming
- Making your own minimal Linux distribution

→ ***Putting it all in perspective***



Part-II RTOS/IoT programming
- RIOT programming
- IoT Infrastructure
- Lora networking





First Part – Pedagogy

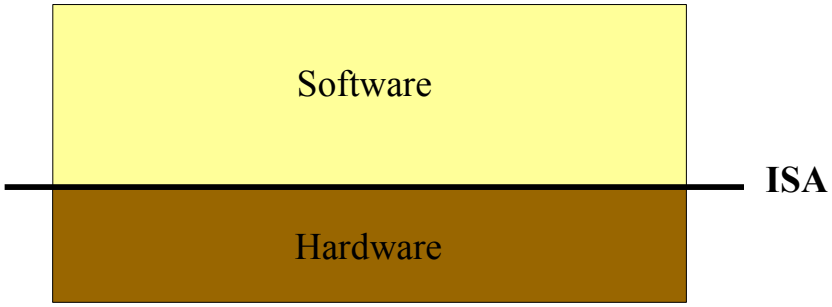
- Inverted pedagogy --- Team learning, individual work
  - Guided hands-on learning and coding
  - A work log to read, to understand, to complete, and to make your own
- The work log is for yourself first
  - A document about what you did and what you have learned
  - This document is first and foremost for your own records
- The destination is not the goal, the goal is the learning along that path
  - Don‘t just answer the questions…
  - Absorb the demonstrated know-how, ask questions about it, put it in practice...
  - Discuss with others what you have learned/understood
  - Help each others, the goal is the learning

First Part – Evaluation

- Part-I Evaluation
  - No exam – but weekly progress checkpoints
  - Every week: you will surrender your work (work log and your code)
  - Your weekly involvment is the largest part of your final grade for this part of the M2M course

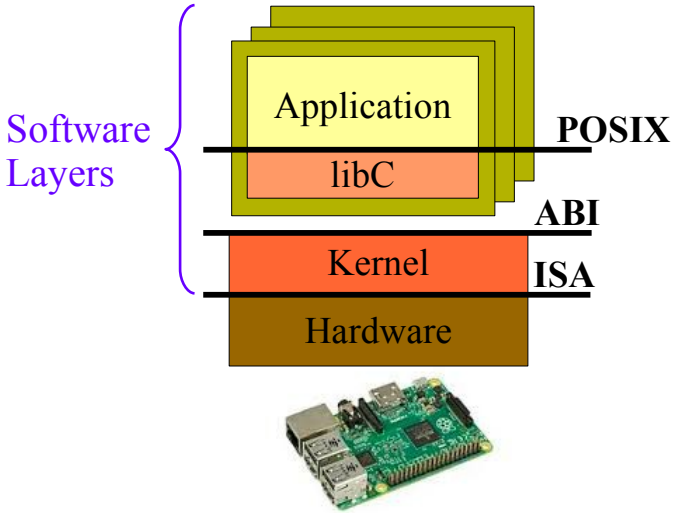
Bare-metal Development

- Bare-metal Software
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set
  - Defines other concepts such as page tables, traps, interrupts, etc.
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools



Toolchain Background

- GNU toolchain
  - GNU gcc, gdb, linker, and utility programs (binutils)
  - Designed to work with Linux kernel
- Major hardware/software interfaces
  - ISA: Instruction Set Architecture
  - *(E)*ABI: *(Embedded)* Application Binary Interface
  - POSIX: Portable Operating System Interface standards



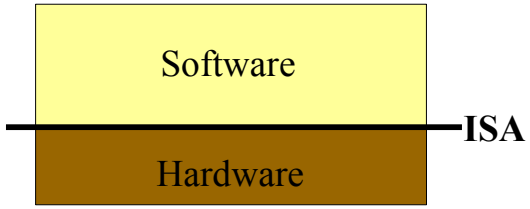
- Different GNU toolchains
  - For different processors, different ABI, and different "kernels/distributions"
  - Example: **i386-linux-abi** or **amd64-linux-abi**
  - Each Linux distribution starts with a toolchain
    - Used to build the kernel
    - Used to build the libC
    - Used to build the applications

Different pieces of code, compiled independently, but interacting at runtime. Thus, the same toolchain must be used by all developers

Cross-Compilation

- Cross compilation
  - The development machine and the compilation target are different
  - May have different ISA as well as different software stack



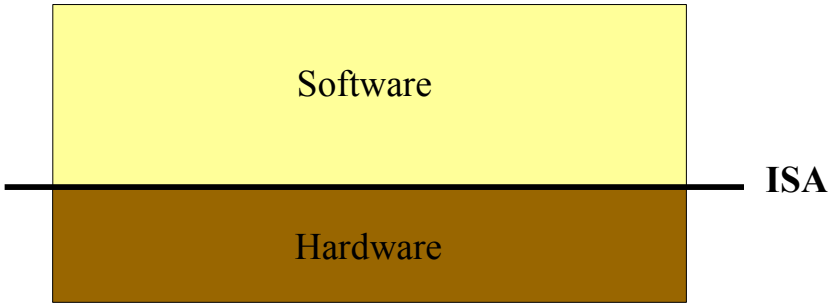
- Cross-compiling bare-metal software
  - Developping on your AMD64 laptop, running Linux kernel, and an Ubuntu distribution
  - Targetting a ARM board, like the VersatilePB or the Raspberry Pi
  - Using different toolchains
    - Raspi: gcc-8-arm-linux-gnueabi
    - **VersatilePB: <u>arm</u>-<u>none</u>-<u>eabi</u>**

Can be installed via ***apt-get***[1]…
If you need something else,
you need to build your own toolchain...

(1) Using the program "synaptic" for a nice GUI over apt-get that allows you to search for available packages

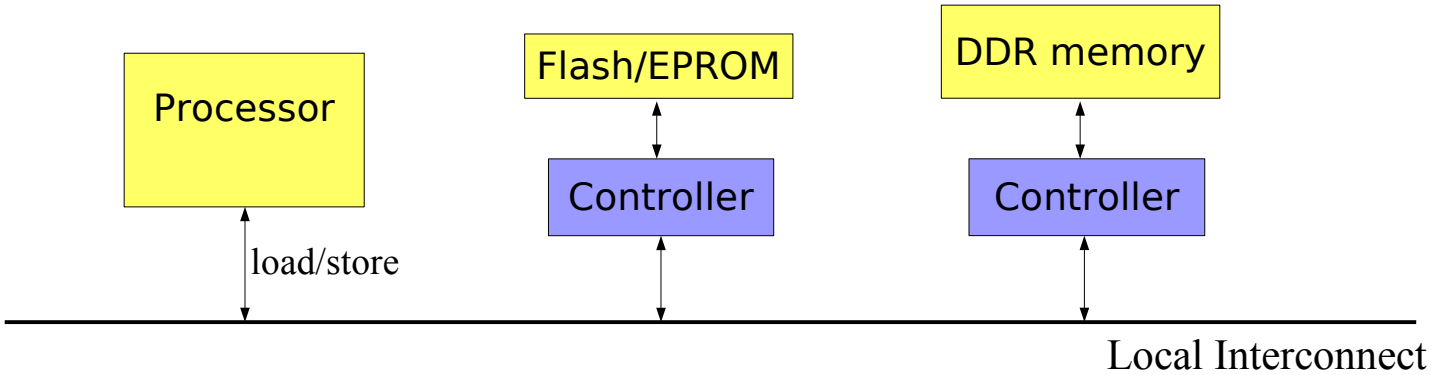
Bare-metal Development

- Bare-metal Software
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set
  - Defines other concepts such as page tables, traps, interrupts, etc.
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools

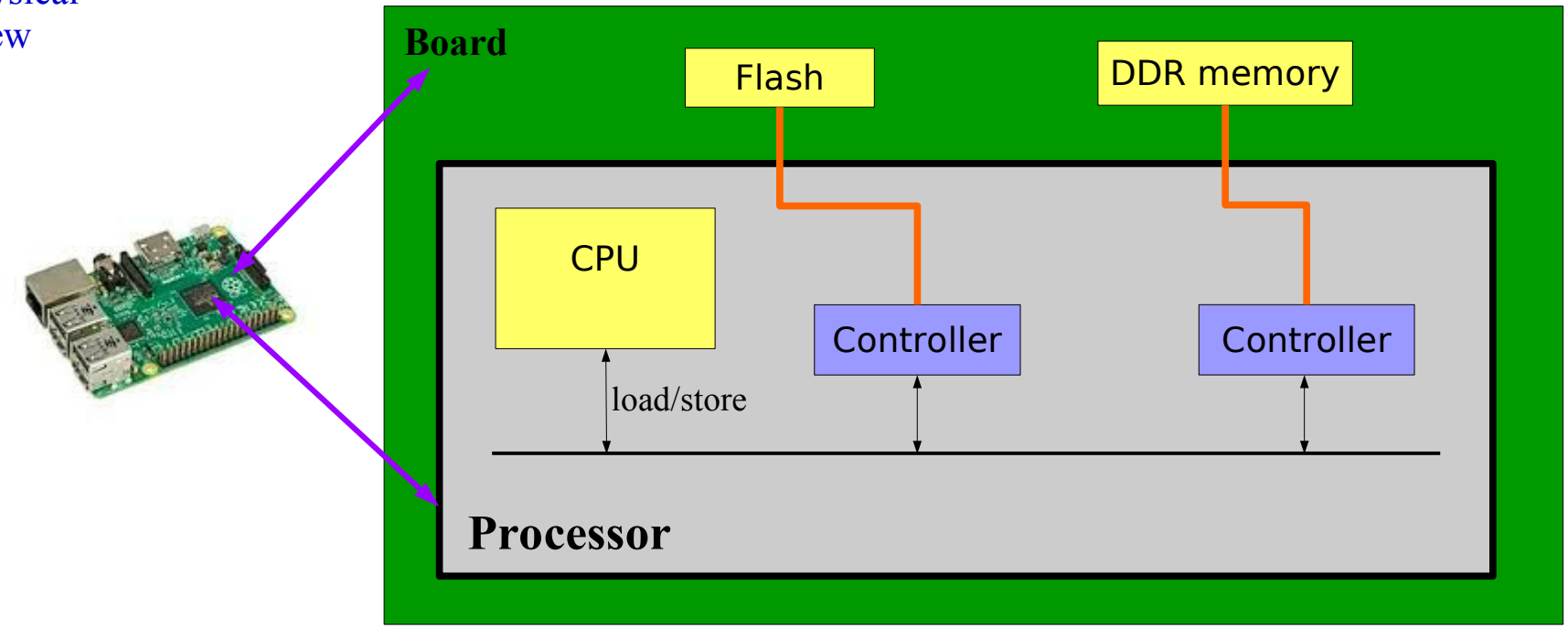


Hardware – Basics

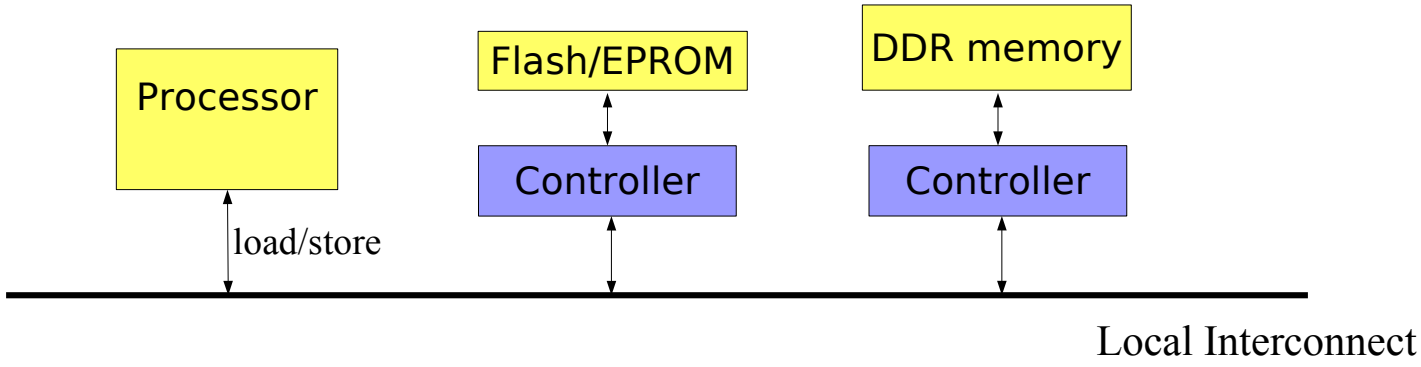
Conceptual View



Physical View

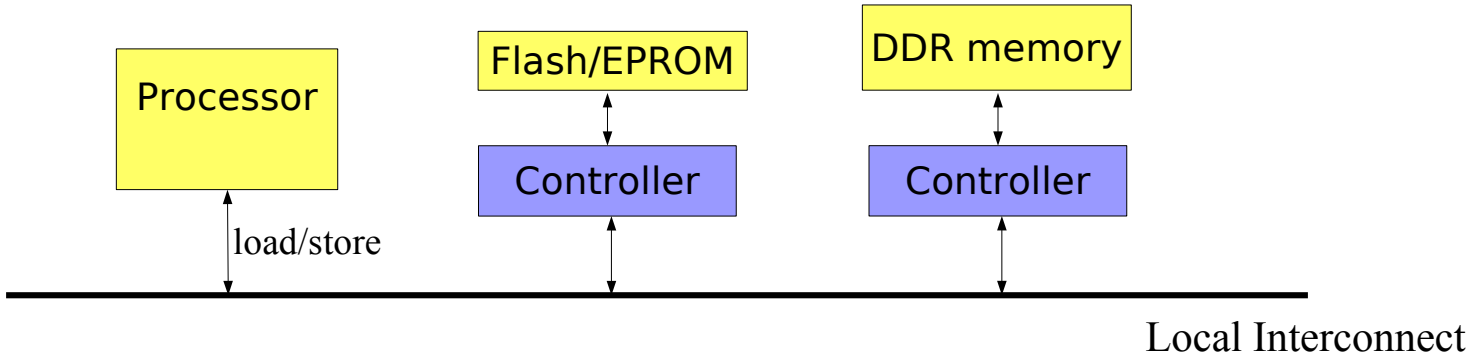


Hardware – Reset / Power-up



- Processor
  - Only knows how to issue ***load/store operations*** on the bus
  - Forever loop: ***fetch – decode – issue***
- Local Interconnect (also called bus)
  - Data wires + control wires
  - Routes load/store operations to controllers
  - Routing is based on the ***memory map***

Hardware – Reset / Power-up



- Boot sequence – ARM example
  - Processor wakes up at a given address, at 0x0000-0000 (reset vector)
  - Starts executing there, but what is there?

ARM Example – Hardware/Software Boundary

- Page 0x0000-0000
  - Interrupts/Traps are ***exceptions*** for the processor

The 7 known exceptions, room for one 4-byte instructions per exception

```
.globl _vector = 0x00000000
_vector:
    ldr     pc, _reset
    ldr     pc, _undefined_instruction
    ldr     pc, _software_interrupt
    ldr     pc, _prefetch_abort
    ldr     pc, _data_abort
    ldr     pc, _not_used
    ldr     pc, _irq
    ldr     pc, _fiq

_reset:                     .word reset
_undefined_instruction:     .word undefined_instruction_handler
_software_interrupt:        .word software_interrupt_handler
_prefetch_abort:            .word prefetch_abort_handler
_data_abort:                .word data_abort_handler
_not_used:                  .word not_used_handler
_irq:                       .word irq_handler
_fiq:                       .word fiq

reset:
    ...
```

↔ ldr pc, [pc+18]

But what is the content of memory after powering up?

Zynq-7000 Memory Map

Table 4-1: **System-Level Address Map**

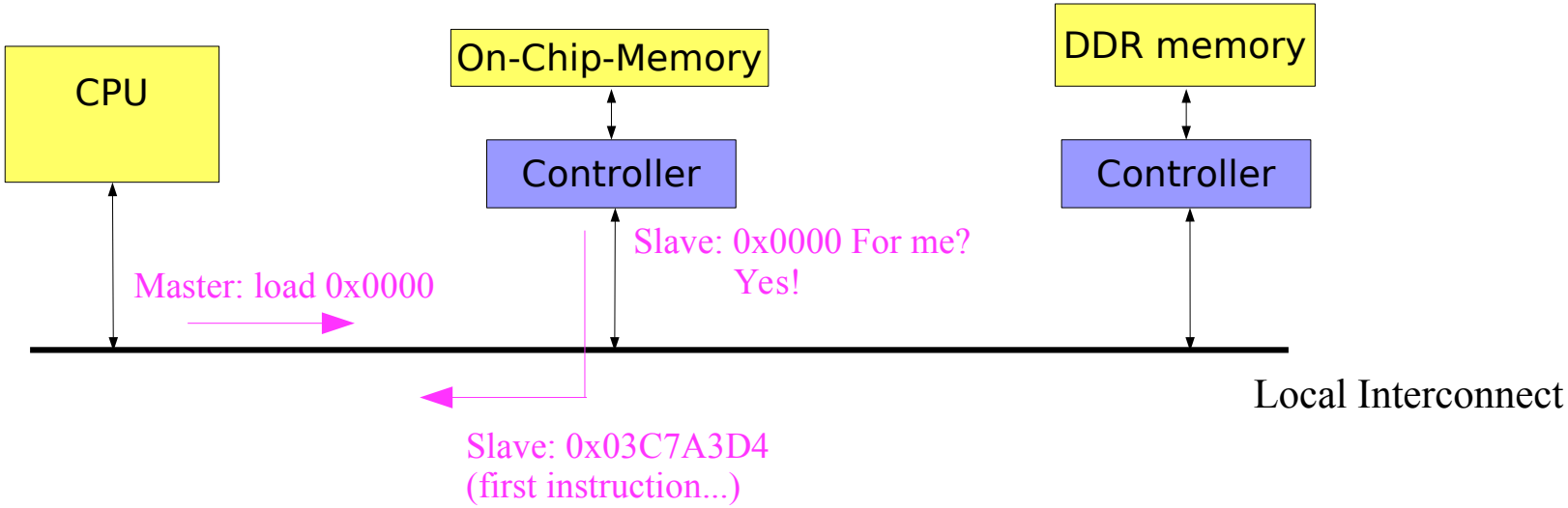
Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Addresses are not filtered by the Snoop Control Unit (not configured yet)

On-Chip-Memory is mapped low, so the low addresses have meaningful contents

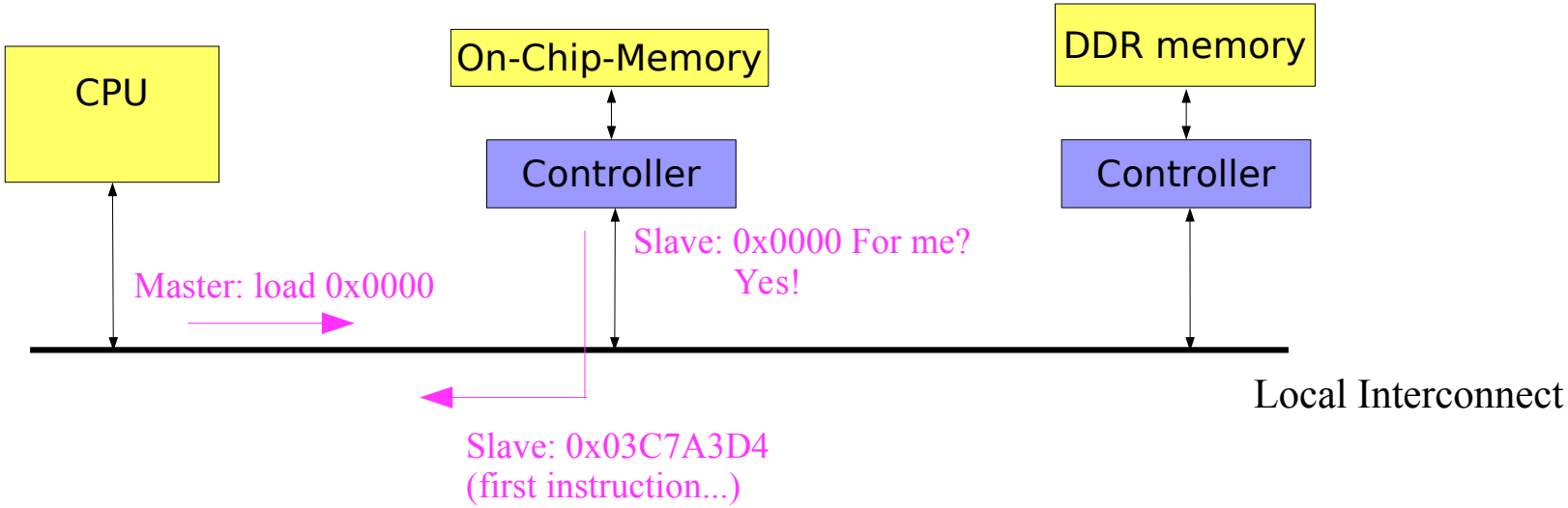
Technical Reference Manual: Zynq-7000 All Programmable SoC

Zybo – Reset / Power-up



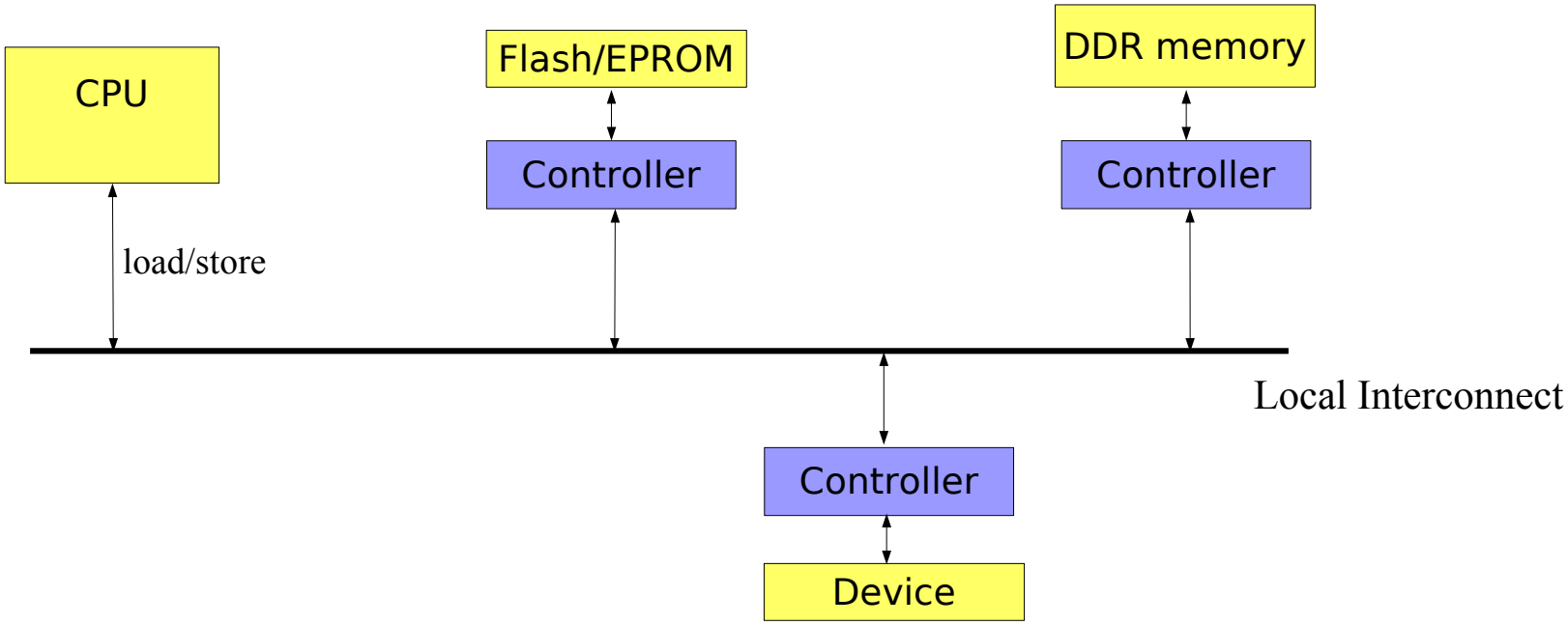
- Primary boot sequence steps
  - Reset the processor sets the program counter register to 0x0000-0000
  - OCM is mapped low (0x0000-0000), so execute program stored in the OCM
  - The code will initializing the various controllers/devices
  - That code may be the only code that will ever run (like in a small IoT sensor)
  - Or it may launch a secondary boot sequence, after having remapped the OCM high

Zybo – Reset / Power-up



- Secondary optional boot sequence steps
  - Loading some larger software from some mass-storage-like device, like an SD card
    - On Intel machine, the primary boot sequence is the BIOS
    - Then goes on loading a boot loader or a Linux kernel for example
  - Copies a trampoline in DDR and jumps to it
  - The trampoline allows to remap the OCM high and then jump to the loaded code

Hardware – Questions

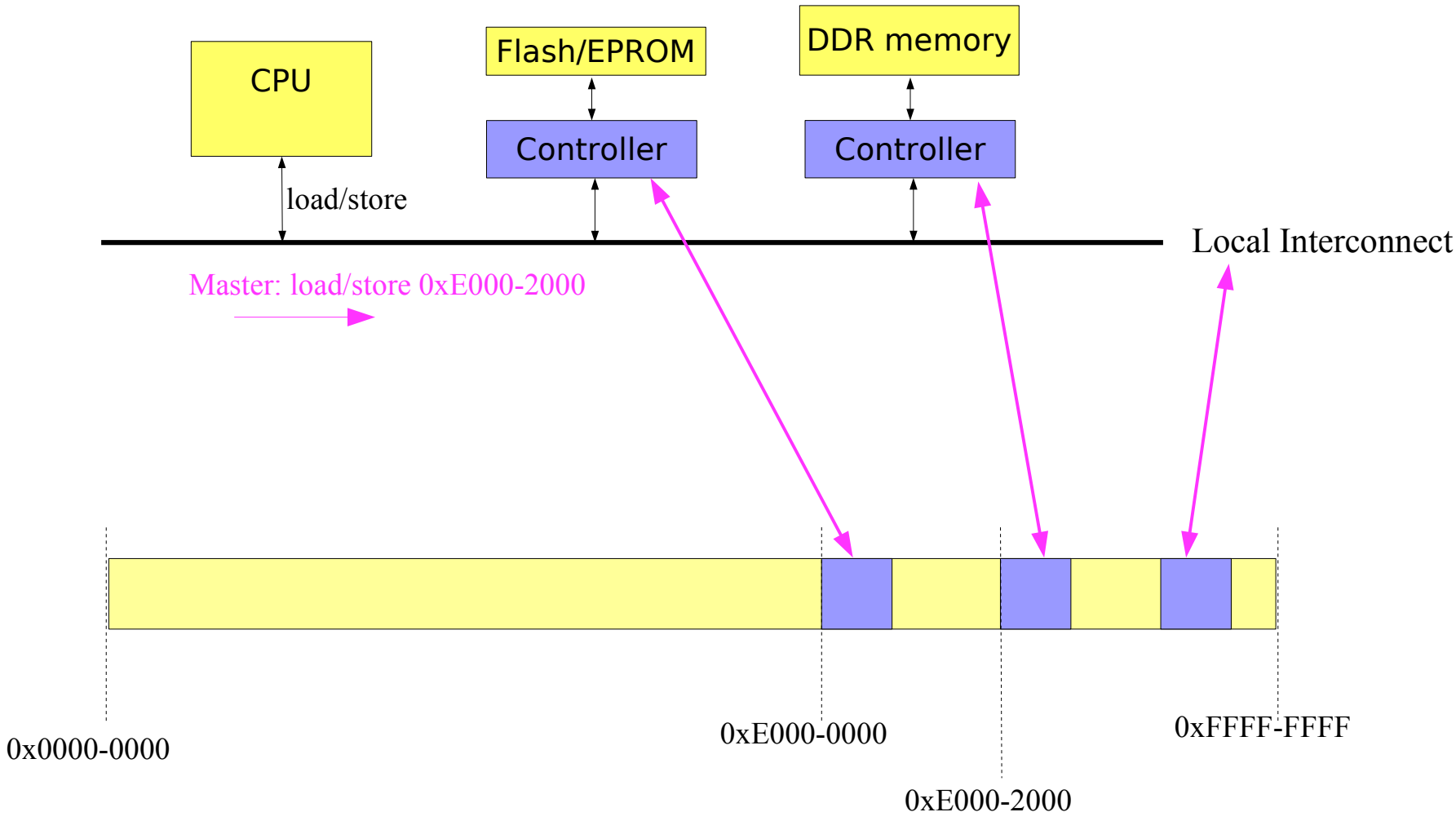


How can the Flash/EPROM can be remaped?

How can software configure hardware controllers?

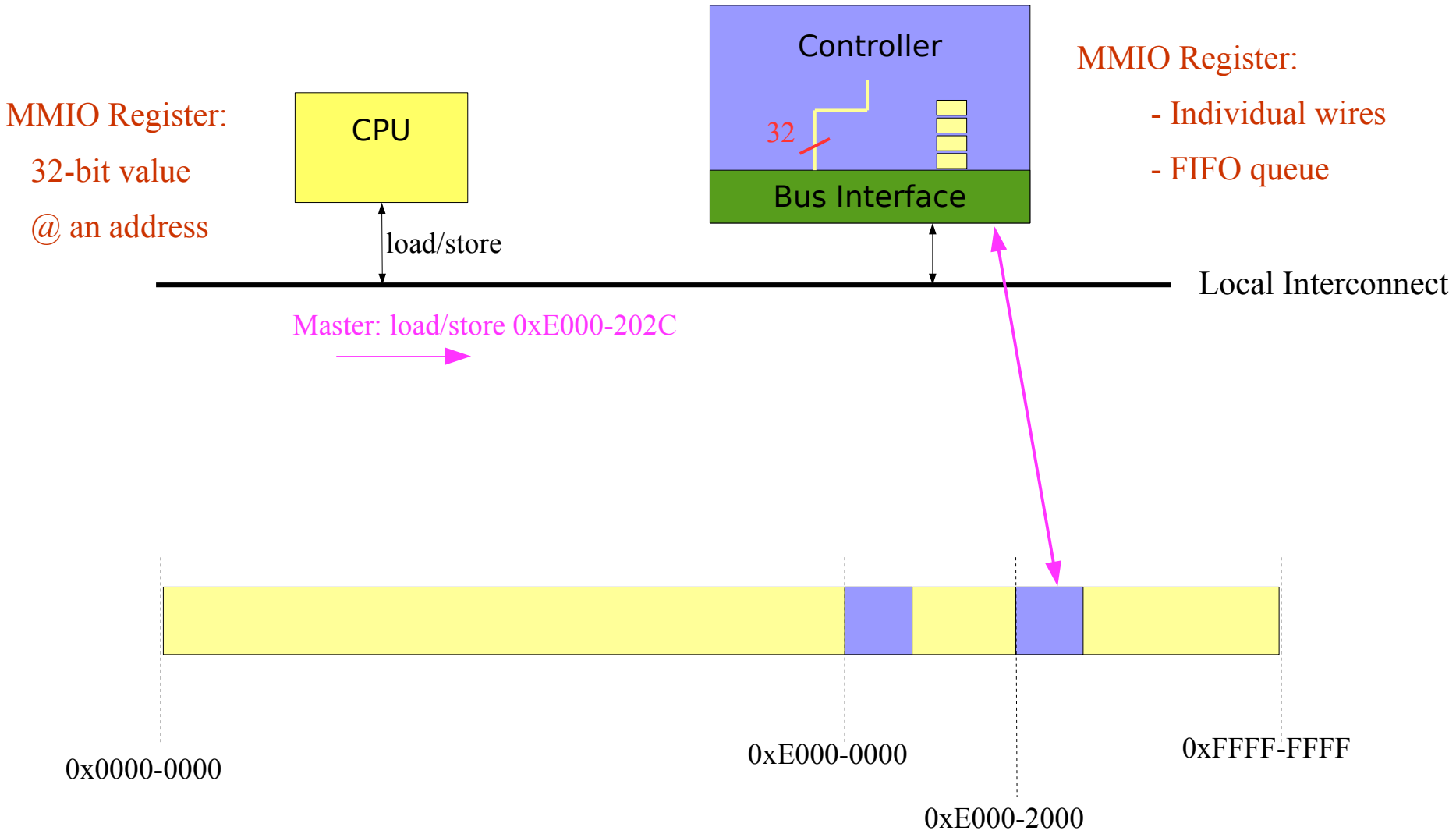
More generally, how does software interacts with devices attached to hardware controllers?

Hardware – MMIO Registers



Memory-Mapped Input/Output Ranges

Hardware – MMIO Registers



MMIO: Memory-Mapped Input/Output

Hardware – Zynq-7000 Overview

AP

Processing System

I/O Peripherals: USB OTG, USB OTG, GigaE, GigaE, SD SDIO, SD SDIO, GPIO, UART, UART, CAN, CAN, I2C, I2C, SPI, SPI

Clock Generation

Reset

USB 2, GigE 2, SD 2 → IOP Master Interconnect

IRQ

IOP Slave Interconnect

MIO-EMIO

Memory Interfaces: SRAM/NOR, ONFi 1.0 NAND, Q-SPI CTRL

Central Interconnect

Processing System Master Interconnect

Processing System Slave Interconnect

Application Processor Unit

WDT

TTC

System Level Control Regs

DMA 8 Channel

FPU and NEON Engine: MMU, ARM Cortex A-9 CPU, 32 KB I-Cache, 32 KB D-Cache

FPU and NEON Engine: MMU, ARM Cortex A-9 CPU, 32 KB I-Cache, 32 KB D-Cache

GIC

Snoop Controller Unit

512 KB L2 Cache & Controller

OCM Interconnect

256K SRAM

Memory Interfaces: DDR2/3, LPDDR2 Controller

CoreSight Components

DAP

DVC

Programmable Logic to Memory Interconnect

EMIO

XADC 12 bit ADC

DMA Sync

IRQ

Config AES/SHA

ACP

GTM

Legend
Arrow direction shows control (master to slave)
data flows in both directions
AXI3 32bit/64bit, AXI 64bit, AXI 32bit, AHB 32bit, APB 32bit, Custom

Programmable Logic

PCIe

SelectIO

X12500

Figure 2-1: **Zynq-7000 AP SoC Processor System High-Level Diagram**

Zynq-7000 Memory Map

Table 4-1: **System-Level Address Map**

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Technical Reference Manual: Zynq-7000 All Programmable SoC

Zynq-7000 Memory Map

Table 4-7: **PS System Register Map**

Register Base Address	Description (Acronym)	Register Set
F800_1000, F800_2000	Triple timer counter 0, 1 (TTC 0, TTC 1)	ttc.
F800_3000	DMAC when secure (DMAC S)	dmac.
F800_4000	DMAC when non-secure (DMAC NS)	dmac.
F800_5000	System watchdog timer (SWDT)	swdt.
F800_6000	DDR memory controller	ddrc.
F800_7000	Device configuration interface (DevC)	devcfg.
F800_8000	AXI_HP 0 high performance AXI interface w/ FIFO	afi.
F800_9000	AXI_HP 1 high performance AXI interface w/ FIFO	afi.
F800_A000	AXI_HP 2 high performance AXI interface w/ FIFO	afi.
F800_B000	AXI_HP 3 high performance AXI interface w/ FIFO	afi.
F800_C000	On-chip memory (OCM)	ocm.
F800_D000	eFuse(1)	-
F800_F000	Reserved	Reserved

Zynq-7000 Memory Map

Table 4-6: **I/O Peripheral Register Map**

Register Base Address	Description
E000_0000, E000_1000	UART Controllers 0, 1
E000_2000, E000_3000	USB Controllers 0, 1
E000_4000, E000_5000	I2C Controllers 0, 1
E000_6000, E000_7000	SPI Controllers 0, 1
E000_8000, E000_9000	CAN Controllers 0, 1
E000_A000	GPIO Controller
E000_B000, E000_C000	Ethernet Controllers 0, 1
E000_D000	Quad-SPI Controller
E000_E000	Static Memory Controller (SMC)
E010_0000, E010_1000	SDIO Controllers 0, 1

UART = Universal Asynchronous Receiver and Transmitter

Also called RS-232 (a standard for a serial line over 2 wires

UART Device Example

UART... serial line controller, following the RS-232 protocol...

→ Essentially a FIFO and a status register...

Corresponding the **mmio registers** defined in the Zynq-7000/R1P8 Technical Reference Manual

```
#define UART_R1P8_CR        0x0000 /* UART Control Register */
#define UART_R1P8_MR        0x0004 /* UART Mode Register */
#define UART_R1P8_IER       0x0008 /* -- Interrupt Enable Register */
#define UART_R1P8_IDR       0x000C /* -- Interrupt Disable Register */
#define UART_R1P8_IMR       0x0010 /* -- Interrupt Mask Register */
#define UART_R1P8_ISR       0x0014 /* -- Channel Interrupt Status Register */
#define UART_R1P8_BAUDGEN   0x0018 /* Baude Rate Generator Register */
#define UART_R1P8_RXTOUT    0x001C /* -- Receiver Timeout Register */
#define UART_R1P8_RXWM      0x0020 /* -- Receiver FIFO Trigger Level Register */
#define UART_R1P8_MODEMCR   0x0024 /* -- Modem Control Register */
#define UART_R1P8_MODEMSR   0x0028 /* -- Modem Status Register */
#define UART_R1P8_SR        0x002C /* Channel Status Register */
#define UART_R1P8_FIFO      0x0030 /* Transmit & Receive FIFO */
#define UART_R1P8_BAUDDIV   0x0034 /* Baud Rate Divider Register */
#define UART_R1P8_FLOWD     0x0038 /* -- Flow Control Delay Register */
#define UART_R1P8_TXWM      0x0044 /* -- Transmitter FIFO Trigger Level Register */
```

UART Device Example

Interacting with a device:

1) choose one mmio range corresponding to your device

2) choose one or more register (at different offsets in that range)

3) work with one or more bits in that register

```
#define UART0  0xE0000000
#define UART1  0xE0001000

#define UART_R1P8_SR            0x002C /* Channel Status Register */
#define UART_R1P8_FIFO          0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 <<  4)
#define UART_R1P8_SR_TEMPTY   (1 <<  3)
#define UART_R1P8_SR_RFUL     (1 <<  2)
#define UART_R1P8_SR_REMPTY   (1 <<  1)
#define UART_R1P8_SR_RTRIG    (1 <<  0)
```

UART – Initialization

```
void
uart_r1p8_init_regs(void* uart) {  /* See Zynq TRM sequence (UG585 p598) */

  /* UART Character frame */
  mmio_reg_write32(uart,UART_R1P8_MR,UART_R1P8_MR_8n1);

  /* Baud Rate configuration */
  mmio_reg_setbits32(uart,UART_R1P8_CR,     UART_R1P8_CR_RXDIS | UART_R1P8_CR_TXDIS);
  mmio_reg_write32(uart,UART_R1P8_BAUDGEN,  UART_R1P8_115200_GEN);
  mmio_reg_write32(uart,UART_R1P8_BAUDDIV,  UART_R1P8_115200_DIV);
  mmio_reg_setbits32(uart,UART_R1P8_CR,     UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES);
  mmio_reg_setbits32(uart,UART_R1P8_CR,     UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN);

  /* Disable Rx Trigger level */
  mmio_reg_write32(uart,UART_R1P8_RXWM,     0x00);

  /* Enable Controller */
  mmio_reg_write32(uart,UART_R1P8_CR,     UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES |
          UART_R1P8_CR_RSTTO | UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN |
          UART_R1P8_CR_STPBRK);

  /* Configure Rx Timeout */
  mmio_reg_write32(uart,UART_R1P8_RXTOUT,   0x00);
  mmio_reg_write32(uart,UART_R1P8_IER,      0x00);
  mmio_reg_write32(uart,UART_R1P8_IDR,      UART_R1P8_IxR_ALL);

  /* No Flow delay */
  mmio_reg_write32(uart,UART_R1P8_FLOWD,    0x00);

  /* Desactivate flowcontrol */
  mmio_reg_clearbits32(uart,UART_R1P8_MODEMCR, UART_R1P8_MODEMCR_FCM);

  /* Mask all interrupts */
  mmio_reg_clearbits32(uart,UART_R1P8_IMR, 0x01FFF);
}
```

UART – Output

```
#define UART0  0xE0000000
#define UART1  0xE0001000

#define UART_R1P8_SR            0x002C /* Channel Status Register */
#define UART_R1P8_FIFO          0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 <<  4)
#define UART_R1P8_SR_TEMPTY   (1 <<  3)
#define UART_R1P8_SR_RFUL     (1 <<  2)
#define UART_R1P8_SR_REMPTY   (1 <<  1)
#define UART_R1P8_SR_RTRIG    (1 <<  0)


void
uart_r1p8_putc(void* uart, uint8_t c) {
  while((mmio_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_TFUL) != 0)
        ;
  mmio_write32(uart,UART_R1P8_FIFO, c);
}
```

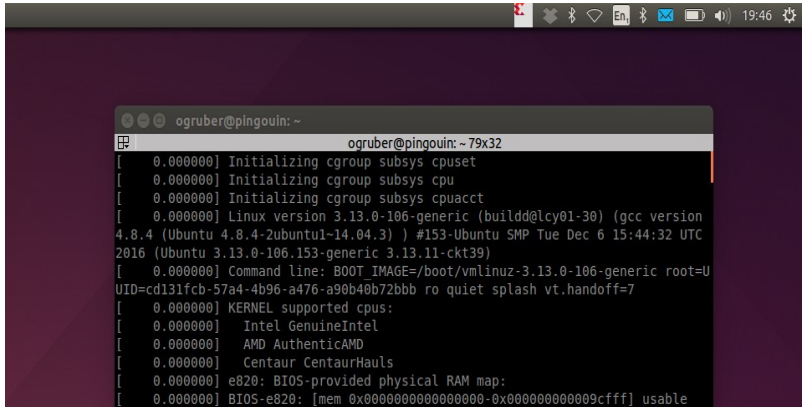
UART – Input

```
#define UART_R1P8_SR            0x002C /* Channel Status Register */
#define UART_R1P8_FIFO          0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 << 4)
#define UART_R1P8_SR_TEMPTY   (1 << 3)
#define UART_R1P8_SR_RFUL     (1 << 2)
#define UART_R1P8_SR_REMPTY   (1 << 1)
#define UART_R1P8_SR_RTRIG    (1 << 0)
```

```
uint8_t
uart_r1p8_getc(void* uart){
  while((mmio_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_REMPTY))
    ;
  return mmio_read32(uart,UART_R1P8_FIFO);
}
```

Development Environment

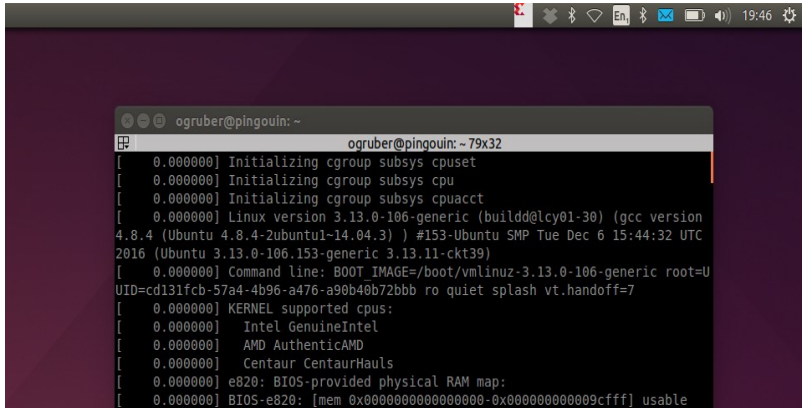
- Using a real board
  - Relies on using a USB-Serial cable
    - Shows as */dev/ttyUSB0* or */dev/ttyUSB1*
    - Look at ***dmesg*** when plugging the cable in
  - Serial line used for the ***console***
    - The *standard input/output streams*
    - Requires using a terminal emulator on your laptop
    - Like ***minicom*** for instance
    - Typical configuration[(1)]: "115200 8N1"

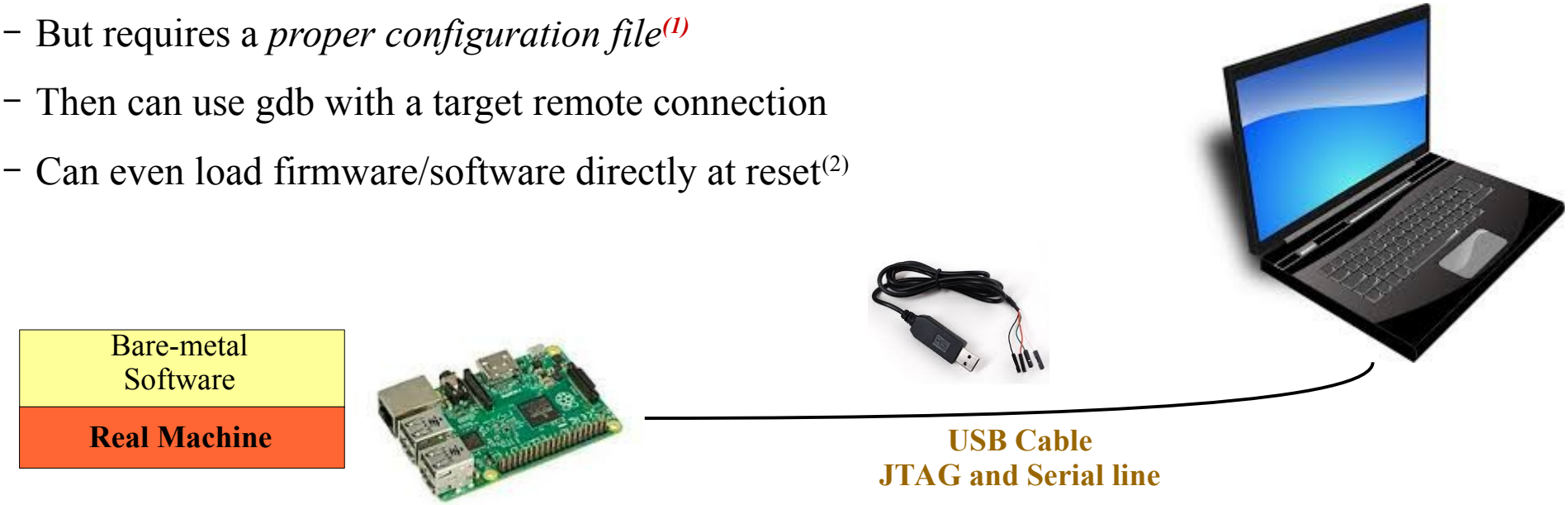


(1) "115200 8N1": the baud rate is set to 115200bps, data-bits set to 8 (the '8' in 8N1), parity is set to none (the 'N' in 8N1), and stop-bits is set to 1 (the '1' in 8N1)

Development Environment

- Hardware debugging
  - Through JTAG, often behing a USB connector
- Software debugging
  - Open On-Chip Debugger (www.openocd.org)
  - Acts as a GDB debug server
  - But requires a *proper configuration file*[1]
  - Then can use gdb with a target remote connection
  - Can even load firmware/software directly at reset[2]

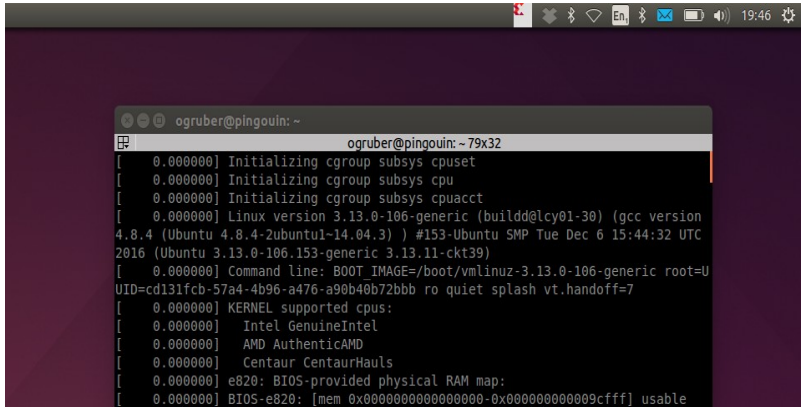




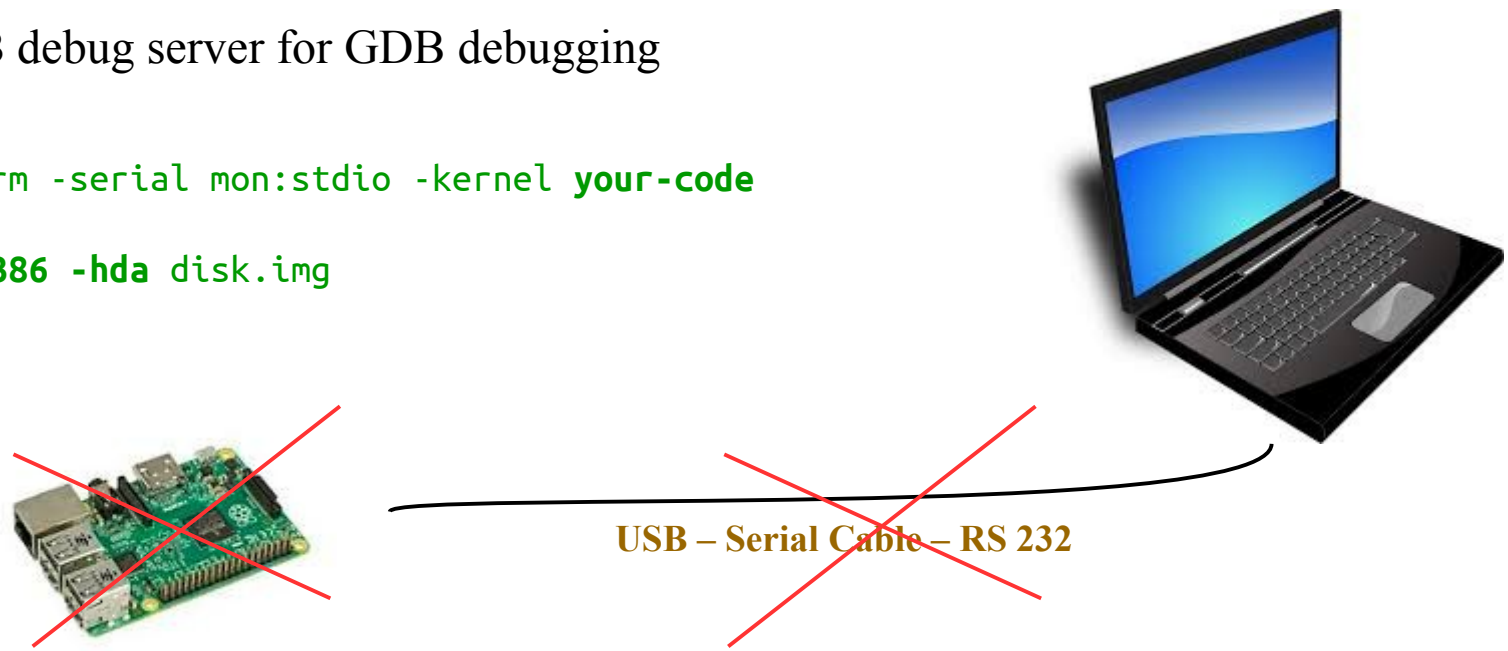
(1) This is the hard part, those are hard to find and not always correct
(2) Avoids having to switching SD cards → will ultimately break the SD socket

Development Environment

- Using an emulator – QEMU
  - A regular Linux process
  - Emulates a machine for your bare-metal software
    - A processor (intel or arm) with memory
    - A serial line for the console
    - One or more mass-storage devices for file systems
  - Support easy debugging
    - Provides a GDB debug server for GDB debugging

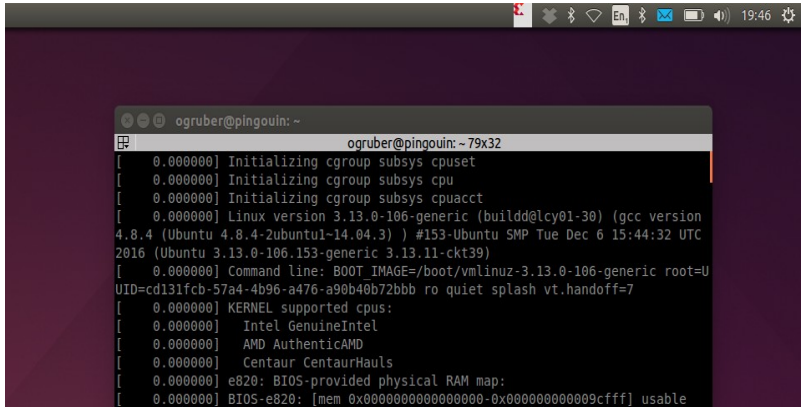



```
$ qemu-system-arm -serial mon:stdio -kernel your-code

$ qemu-system-i386 -hda disk.img
```

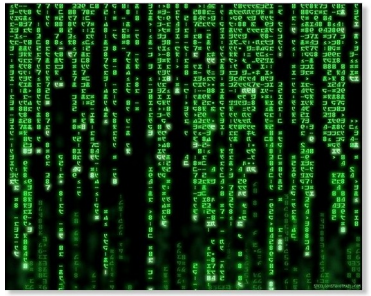


Development Environment

- Using an emulator – QEMU
  - A regular Linux process
  - Emulates a machine for your bare-metal software
  - *But how is possible?*



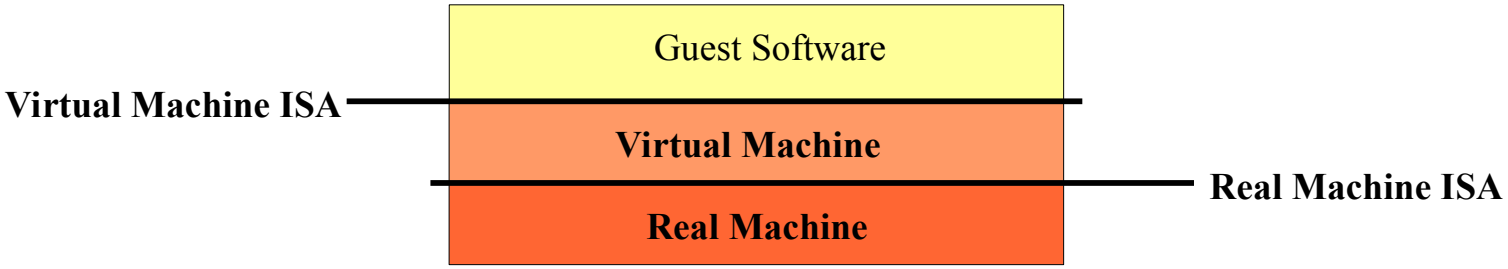
For your code: "The matrix is a prison that you cannot see, taste, or smell." Morpheus



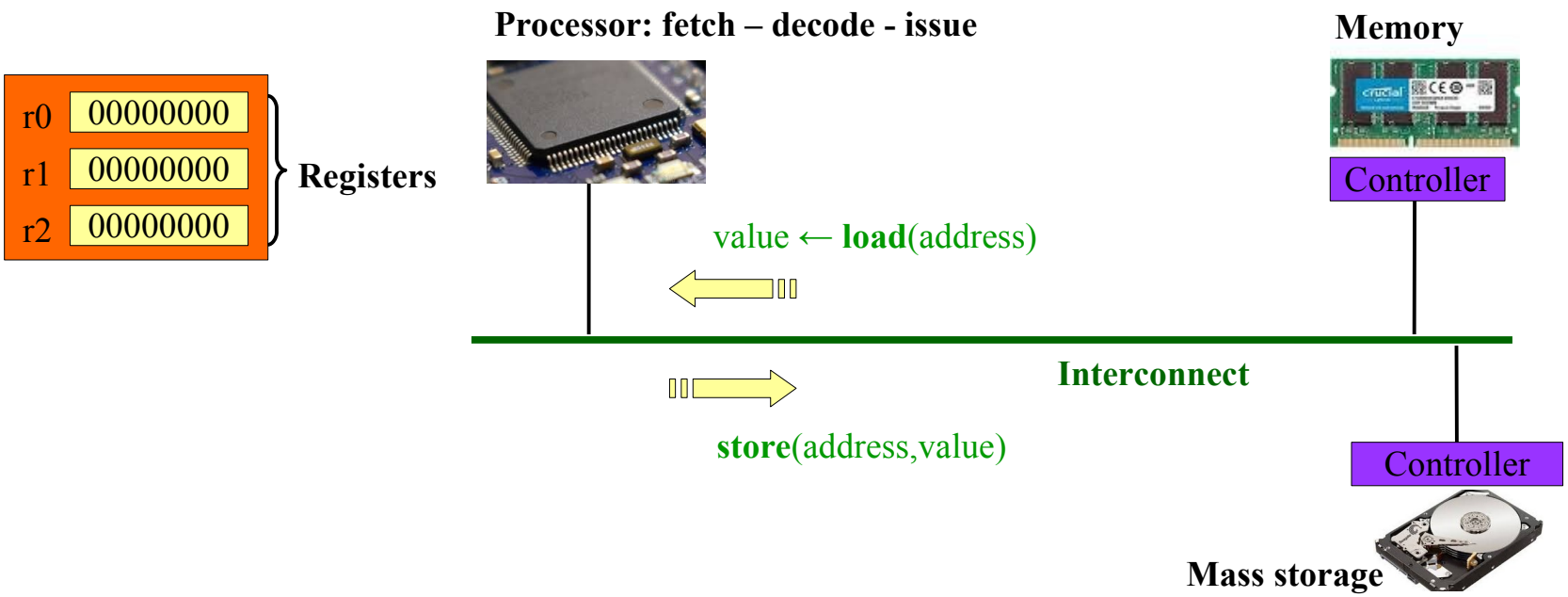
For you: *an all-software-development experience in the confort of your chair!*

USB – Serial Cable – RS 232

Virtual Machines – The Art of Illusions



What is an execution?
Imagine yourself being the code running on the processor, what can you see?



Illusion by Emulation

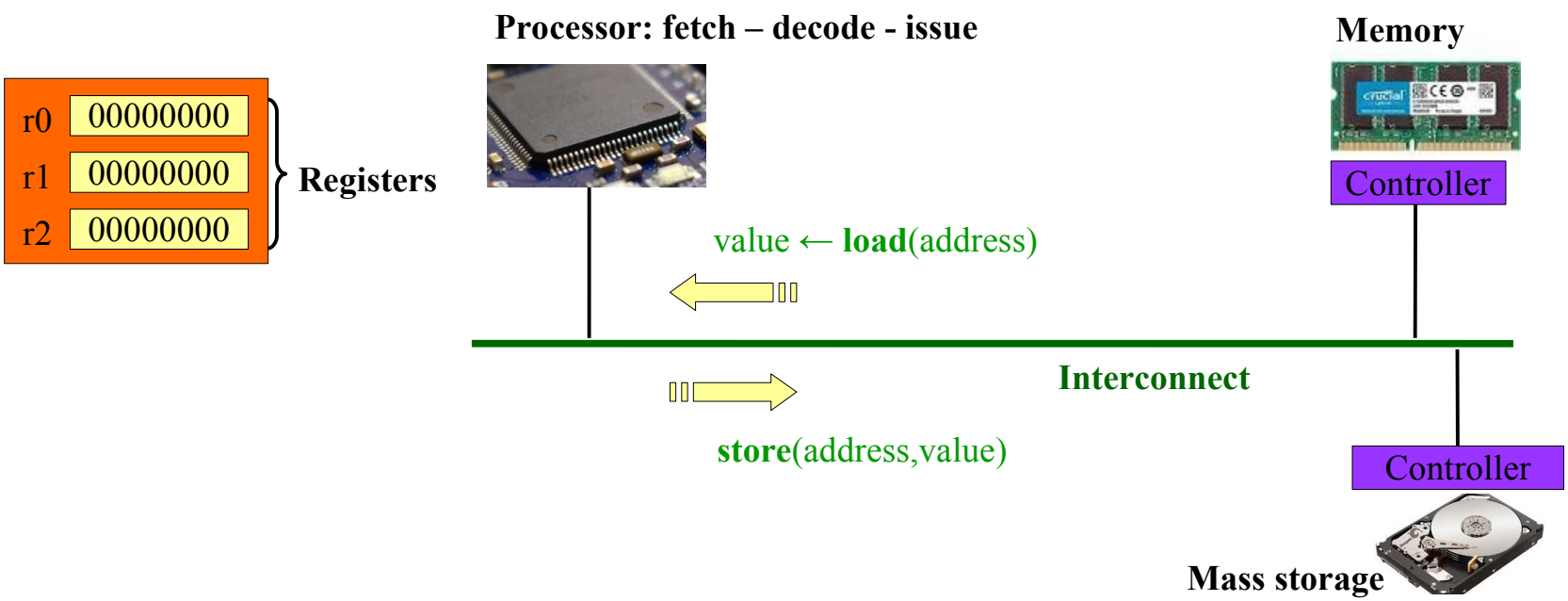
Any execution can be emulated by program

- *Fetch instructions, decode them, and issue them*
- *Maintain the contents of registers*
- *Maintain the contents of memory*

(let's ignore devices for now...)

*After all, a **processor** is only an **interpreter** of instructions, done in **hardware**...*

→ So it can be done in software.



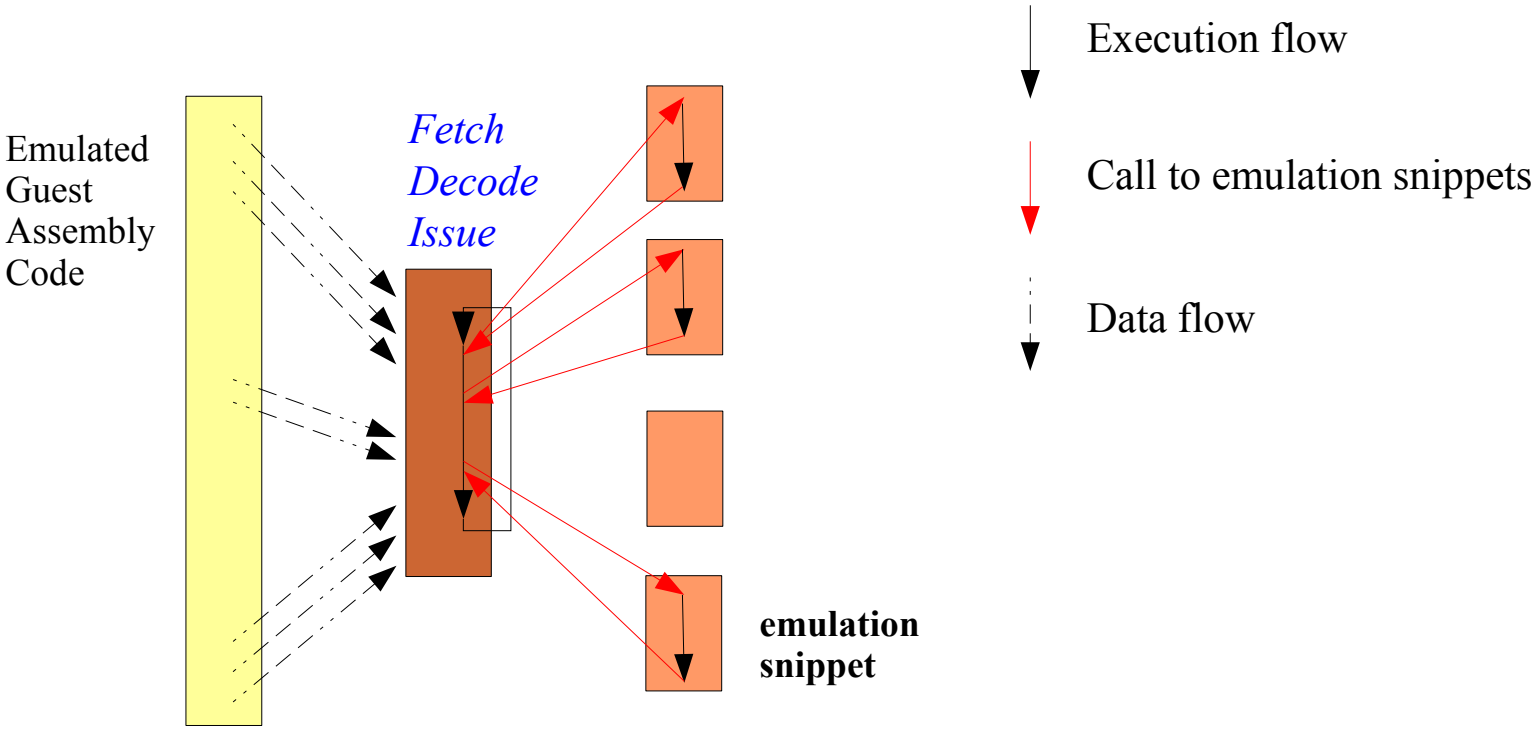
Illusion by Emulation – Architected State

- Architected State
  - Memory
    - A number of physical frames, can be emulated by an array
  - Processor – per core
    - A set of registers (r0-r15 for example)
    - Current status: user or kernel mode
    - Interrupt status
- Core Emulation
  - Just a regular program…
  - Emulating the execution...

```
uint8_t memory[SIZE_512MB];

struct core {
  int32_t regs[16];
  uint32_t flags;
  uint32_t pending_irqs;
  struct mmu *mmu;
};

struct processor {
   struct core cores[4];
   …
};

void wakeup(struct processor *p, int coreno) {
    reset();
    for (;;)
      fetch_decode_issue();
}
```

Emulation by Interpretration

- The emulation is done via an interpreter
  - Like any other interpreter (Shell, JavaScript, or Java)
  - Emulate the Fetch-Decode-Issue loop (FDI loop)
    - Fetch instruction at emulated program counter in emulated memory
    - Decode the instruction → invoke emulation snippet
    - Emulation snippets manipulate the architected state



Emulation Snippets – Examples

mov r1,r0

```
→ core.regs[dst] = core.regs[src];    // actual copy of the emulate register
  core.regs[pc] += 4;                 // update the program counter
```

add r1,r1,r0

```
→ core.regs[dst] = core.regs[left] + core.regs[right];   // the arithmetic operation
  core.regs[pc] += 4;                                    // update the program counter
```

ldr r1,[r0]

```
→ uint32_t* addr = core.regs[src];       // guest address to load from emulated memory
  core.regs[dst] = memory[addr];         // emulate load from memory
  core.regs[pc] += 4;                    // update the program counter
```

bl r1

```
→ uint32_t* addr = core.regs[src];       // guest address to branch to
  core.regs[link] = core.regs[pc];       // save current program counter in link register
  core.regs[pc] = *addr;                 // branch the emulated execution
```

```
struct core {
  int32_t regs[15];
} core;

uint8_t memory[SIZE_64MB];
```

Emulation Snippets – Examples

Set the page table directory (turn on MMU)

```
mov mmu,r0      → core.ptd = core.regs[src];     // actual copy of the emulated register
                  core.regs[pc] += 4;            // update the program counter
```

Memory load/store operations must emulate the MMU now

```
ldr r1,[r0]     → uint32_t* addr = core.regs[src];       // address to load from emulated memory
                  addr = softMMU_translate(addr);        // translate address from virtual to physical
                  core.regs[dst] = memory[addr];         // emulate load from memory
                  core.regs[pc] += 4;                    // update the program counter
```

```
struct page_table_entry {
   int  flags:8;
   int  addr:24;
};

struct page_table {
   struct page_table_entry* entries;
   int depth;
   int page_size;
};

struct core {
  int32_t regs[15];
  struct page_table *pdt;
} core;

uint8_t memory[SIZE_64MB];
```

Virtual Machines – Type I or II

Type-I Hypervisors[1]:
VMware, Xen, …

Installed directly on the hardware.

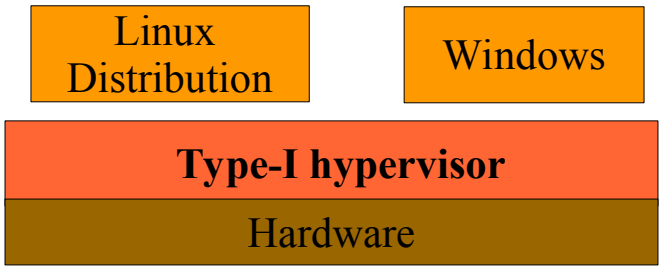
Host multiple guest operating systems that are para-virtualized.

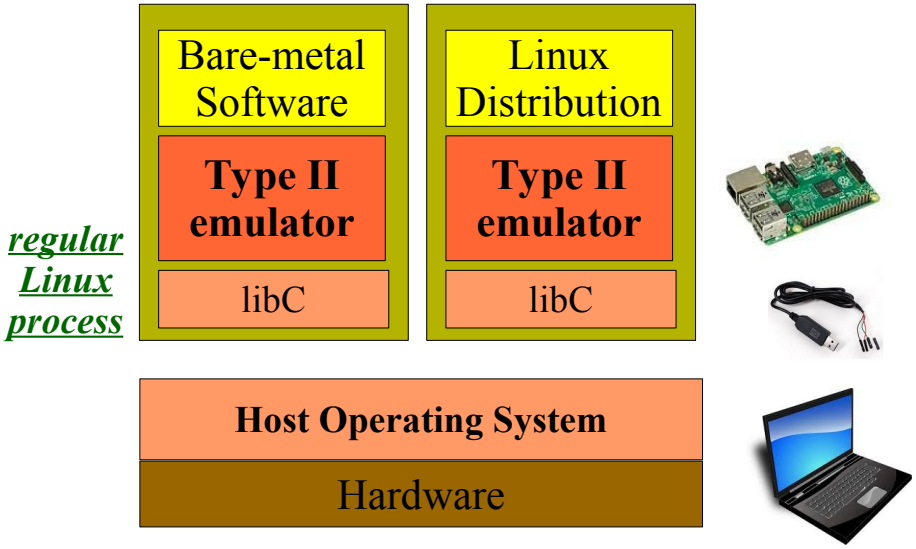
Type-II Virtual Machines:
Qemu/KVM, VirtualBox, …

Installed as a regular application.

Host a single guest operating system, para-virtualized or not.

May be used to host a bare-metal application designed for embedded systems.





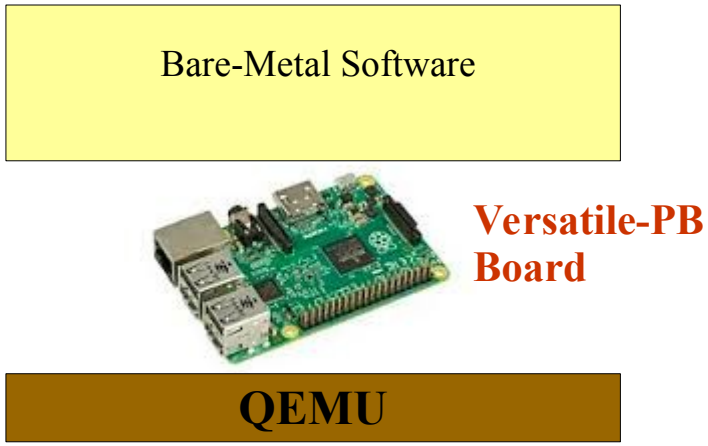
(1) Hypervisor, just another name for a virtual machine, there are many kinds of virtual machines...

QEMU – Booting an VersatilePB Board

- VersatliePB board
  - Small ARM-based board
  - ARM processor: ARM926EJ-S
  - No disk, no display, but a serial line
- QEMU
  - Kernel: a binary image to load in memory
  - Serial: connected to the stdin/stdout of the shell used to launch QEMU

Requires an arm-none-eabi toolchain

Requires to know the UART controller and its mmio registers



$ qemu-system-arm -M **versatilepb** **-serial** mon:stdio -kernel **your-code**

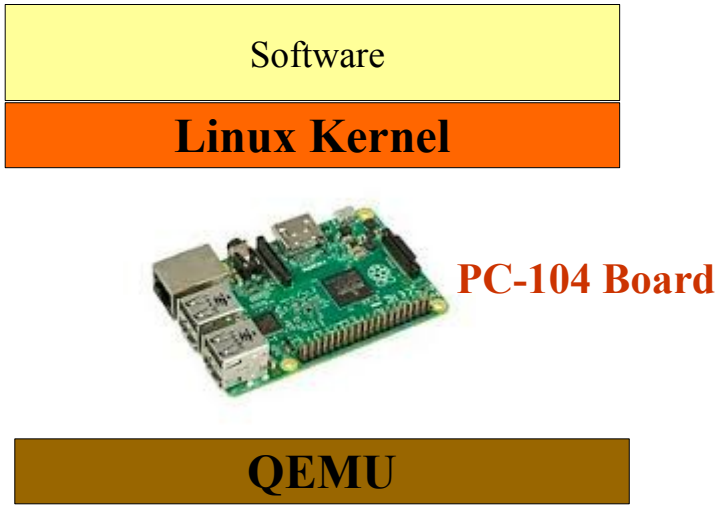
UART0:
Base Address: 0x101F1000

Data register: 0x00
Flag register: 0x18

But, how do we know?

QEMU – Emulated Hardware

- QEMU Emulated Hardware
  - Information via the QEMU monitor
  - Example with a PC-104 board




```
$ qemu-system-i386 -serial mon:stdio
Crtl-A c
(qemu) info qtree
…
dev: i440FX-pcihost, id ""
   irq 0
   bus: pci.0
     dev: PIIX3, id ""
       class ISA bridge, addr 00:01.0,
              pci id 8086:7000 (sub 1af4:1100)
       bus: isa.0
         type ISA
         dev: isa-serial, id ""
           index = 0 (0)
           iobase = 1016 (0x3f8)
           irq = 4 (0x4)
           chardev = "serial0"
           wakeup = 0 (0)
           isa irq 4
```

UART: COM1
Base Address: 0x3F8

Status: 0x3F8 + 0x05
Bit 0x20 → can write a character
Bit 0x01 → can read a character

Out register: 0x3F8
In register: 0x3F8
IRQ: 4

What‘s next?

- Hands-on Learning
  - The worklog in Step0…
- Software Installation for Ubuntu/Debian
  - Step0: support the ARM processor toolchain

    $ sudo apt-get install *qemu-system-arm* *qemu-system-x86*

    $ sudo apt-get install *gcc-arm-none-eabi* *gdb-arm-none-eabi* *binutils-arm-none-eabi*

    Note: if you cannot install the gdb-arm-none-eabi, go for the gdb supporting multiple architectures

    $ sudo apt-get install *gdb-multiarch*

  - Step1: support for 32-bit code

    $ sudo apt-get update -y
    $ sudo apt-get install -y multiarch-support
    $ sudo apt-get install libncurses5:i386