EPFL

Lab. On HW-SW Digital Systems Codesign
EE-390(a)

Session 1
Concept of HW-SW co-design for SoCs

Prof. David Atienza
Dr. Denisa Constantinescu , Dr. Miguel Peón-Quirós
Mr. Rubén Rodríguez-Álvarez, Ms. Stasa Kostic, Mr. Karan Pathk

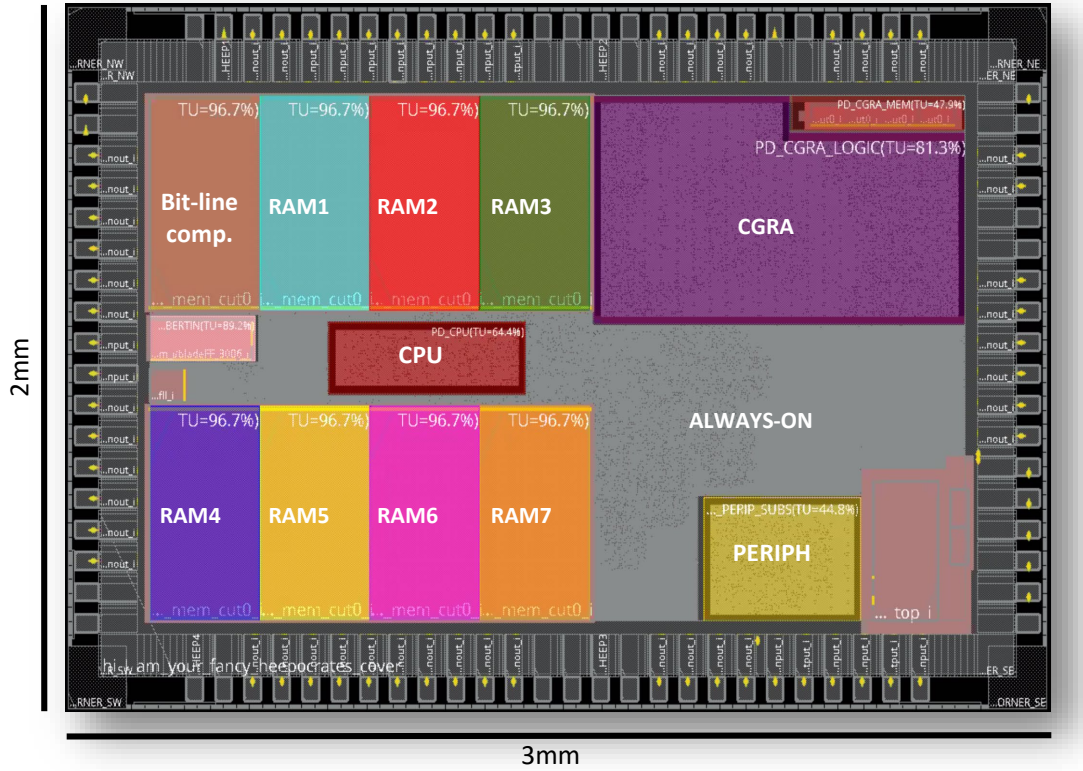
Introduction to the concept of SoC

What is a SoC?

- SoC = System-on-Chip

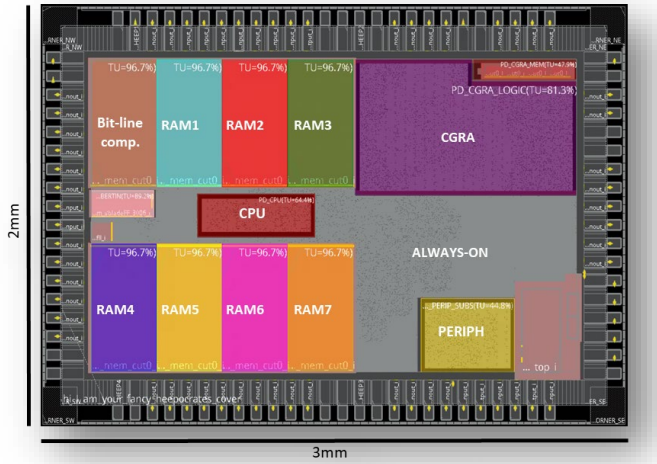
- A computing system that includes all its components in a single integrated circuit:
  - Processors
  - Memory controllers
  - I/O peripherals: UARTs, GPIOs, SPI, I²C, …
  - Some form of internal memory, e.g., SRAM or eDRAM
  - Accelerators for specific functions

- SoC design is normally modular
  - Based on intellectual property (IP) blocks



Example of SoC: ESL's HEEPocrates

- Tiny RISC-V CPU: Core-V[1]
- Memory: 256 KiB of SRAM (8 banks)
- Max frequency: up to 470 MHz
- Power: ~100 µW/MHz (at 1.2 V)
- Accelerators
  - Coarse-grained reconfigurable array (CGRA)
  - In-memory (bit-line) computing



https://www.epfl.ch/labs/esl/research/2d-3d-system-on-chip/x-heep/

[1] OpenHW group github: https://github.com/openhwgroup

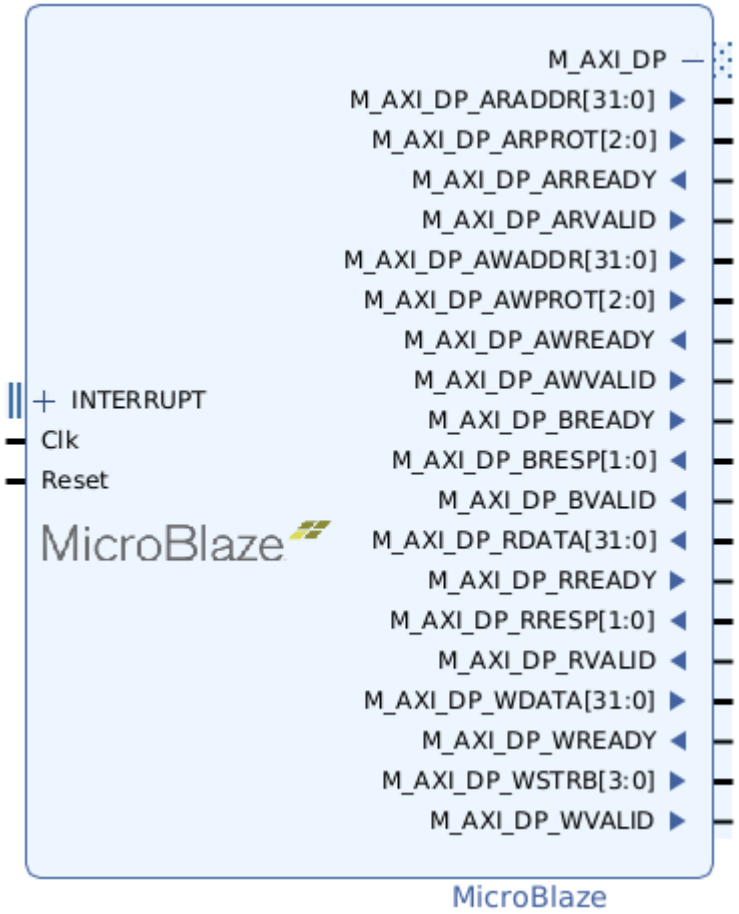
Access to peripherals in a SoC

- In a HW-only (e.g., VHDL) design, modules interface via direct ports:

- However, SoCs are typically processor-based…

… and normally processors cannot directly access peripheral ports!

  - How do we communicate processors with the rest of the system?

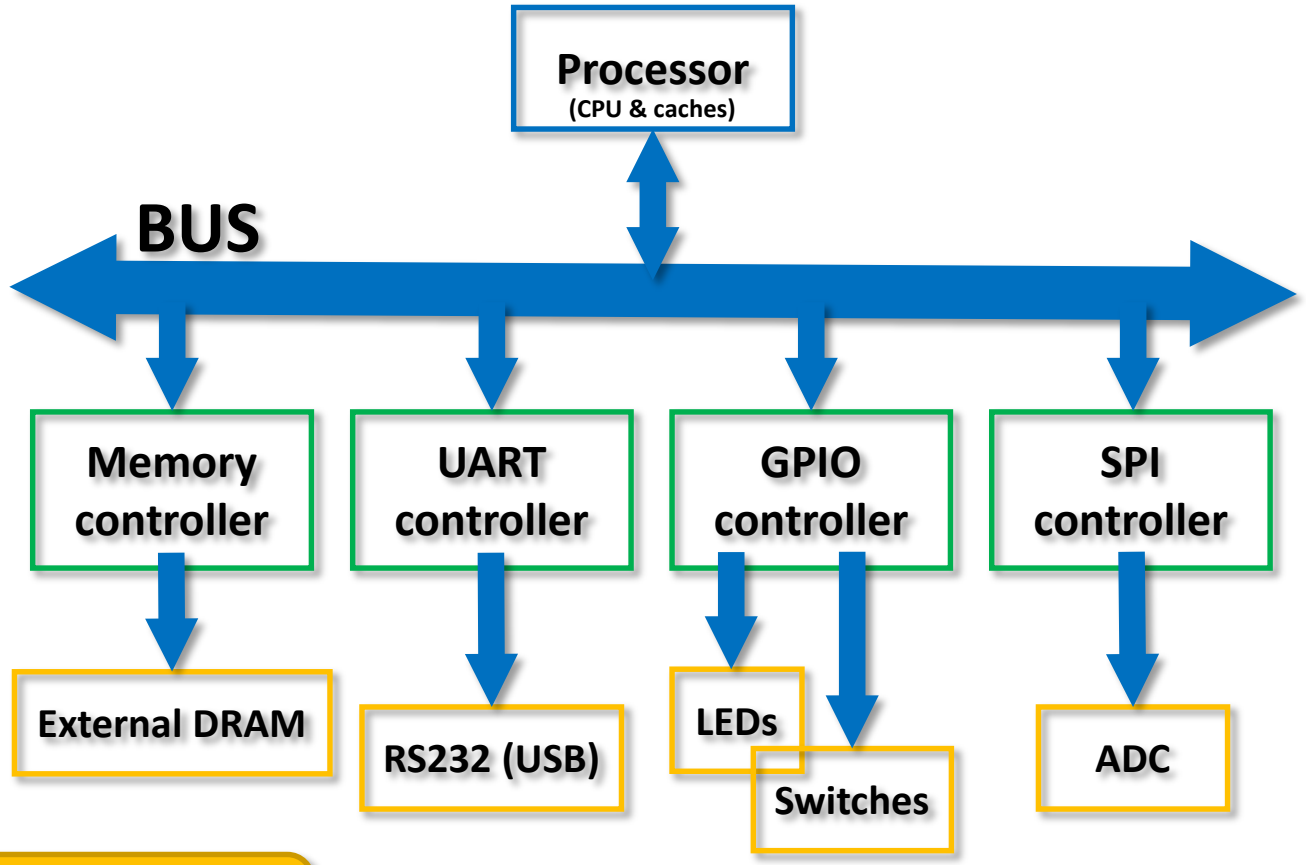


- Processor-based systems use buses

Buses & types of peripherals

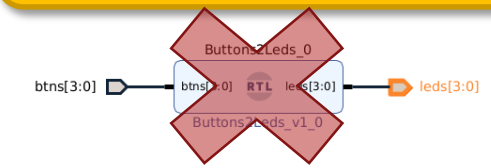
- A bus is a set of wires that allows system components to communicate with each other
  - E.g., a bus allows a processor to read/write from/to a memory
- In general, a bus connects two types of devices:
  - **Masters** initiate transactions. Example: A processor initiates reads/writes towards a memory
  - **Slaves** respond to transactions. Example: A memory returns the contents of a position or stores a new value
- Generalizing the concept, a bus may connect multiple slaves and multiple masters
  - For example, an accelerator can be a master
  - In principle, all masters can initiate transactions towards any of the slaves

Example of communication in a bus-based system



Which are the masters and which the slaves in this system?

How does an application running on the processor change the LEDs to reflect the status of the switches?



The GPIO module offers the processor access to the LEDs and buttons via a bus

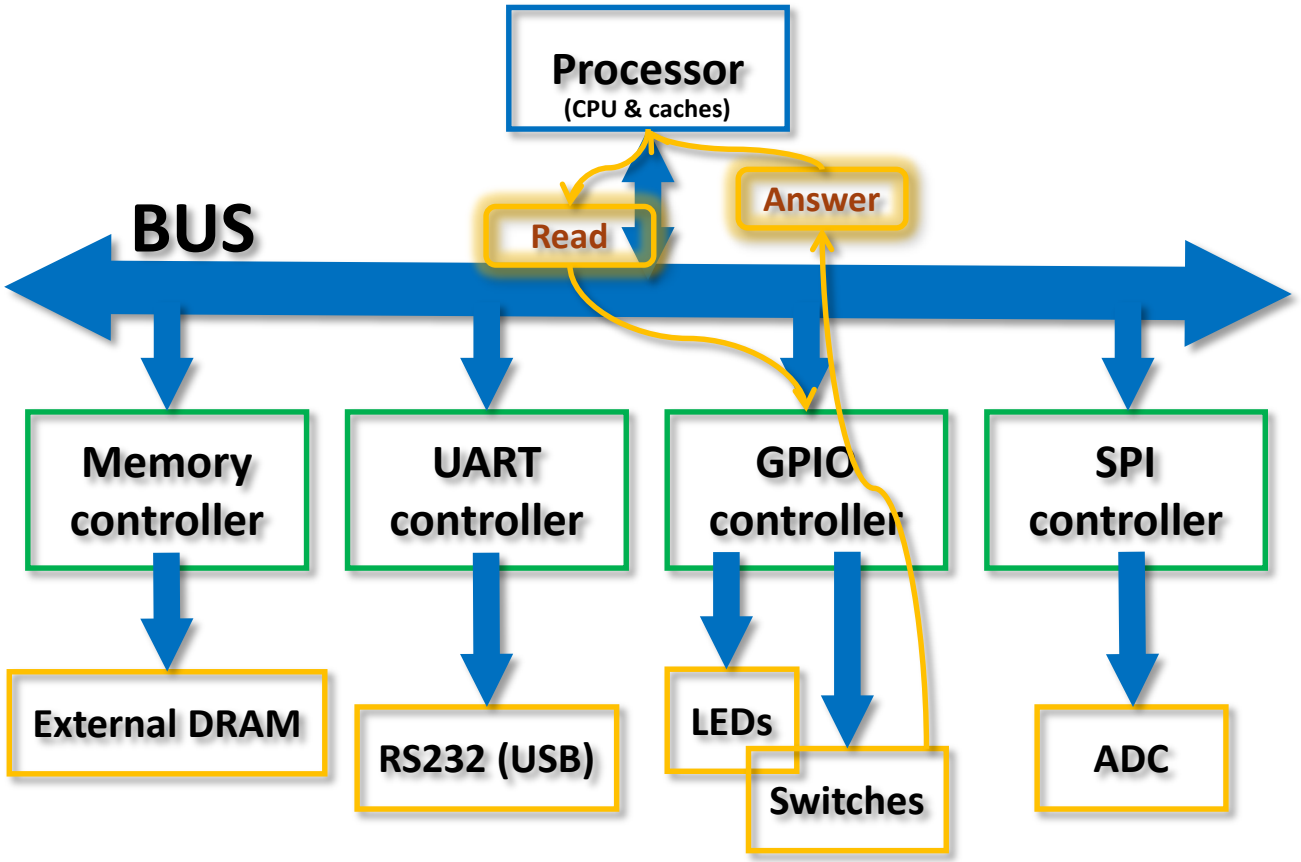
How does the processor contact the peripherals over the bus?

Key idea: memory-mapped IO (MMIO)

Example of communication in a bus-based system

- The processor reads the status of the switches
  1. Proc. sends a read command over the bus to the general purpose I/O controller (GPIO)
  2. The GPIO replies with a transaction on the bus containing the status of the switches

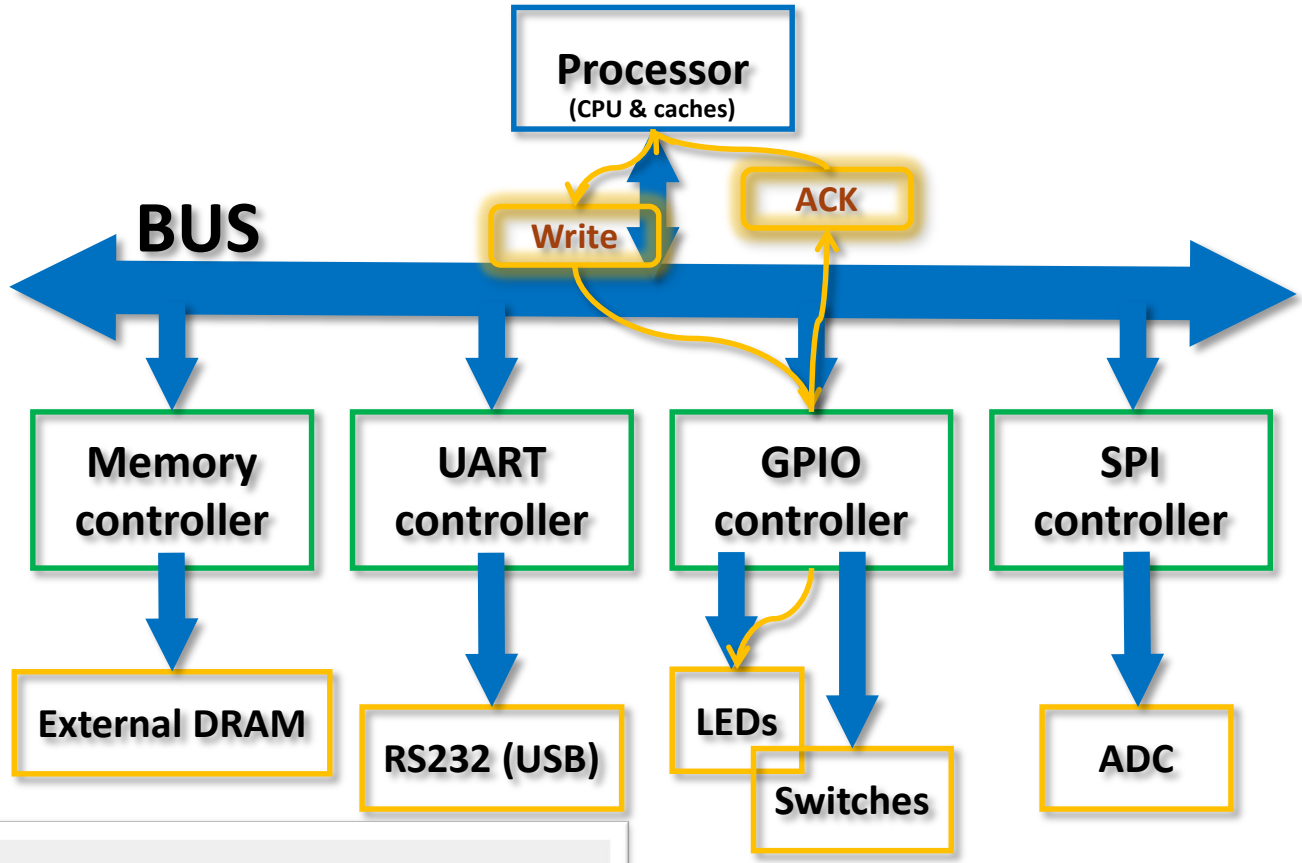
- The processor writes the new status to the LEDs
  1. Proc. sends a write command over the bus to the GPIO controller
  2. The GPIO updates the outputs to the LEDs
  3. The GPIO confirms with a transaction on the bus that the write operation was completed successfully



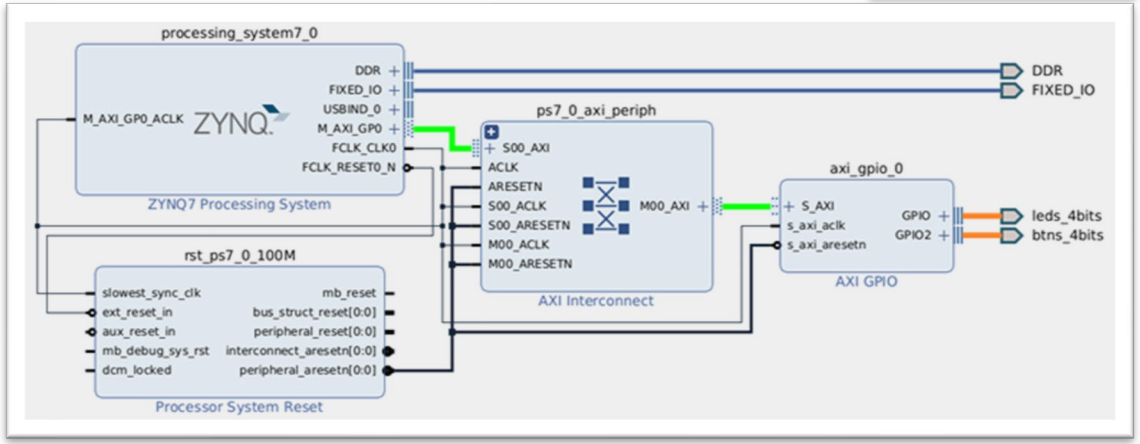
Example of communication in a bus-based system

- The processor reads the status of the switches
  1. Proc. sends a read command over the bus to the general purpose I/O controller (GPIO)
  2. The GPIO replies with a transaction on the bus containing the status of the switches

- The processor writes the new status to the LEDs
  1. Proc. sends a write command over the bus to the GPIO controller
  2. The GPIO updates the outputs to the LEDs
  3. The GPIO confirms with a transaction on the bus that the write operation was completed successfully

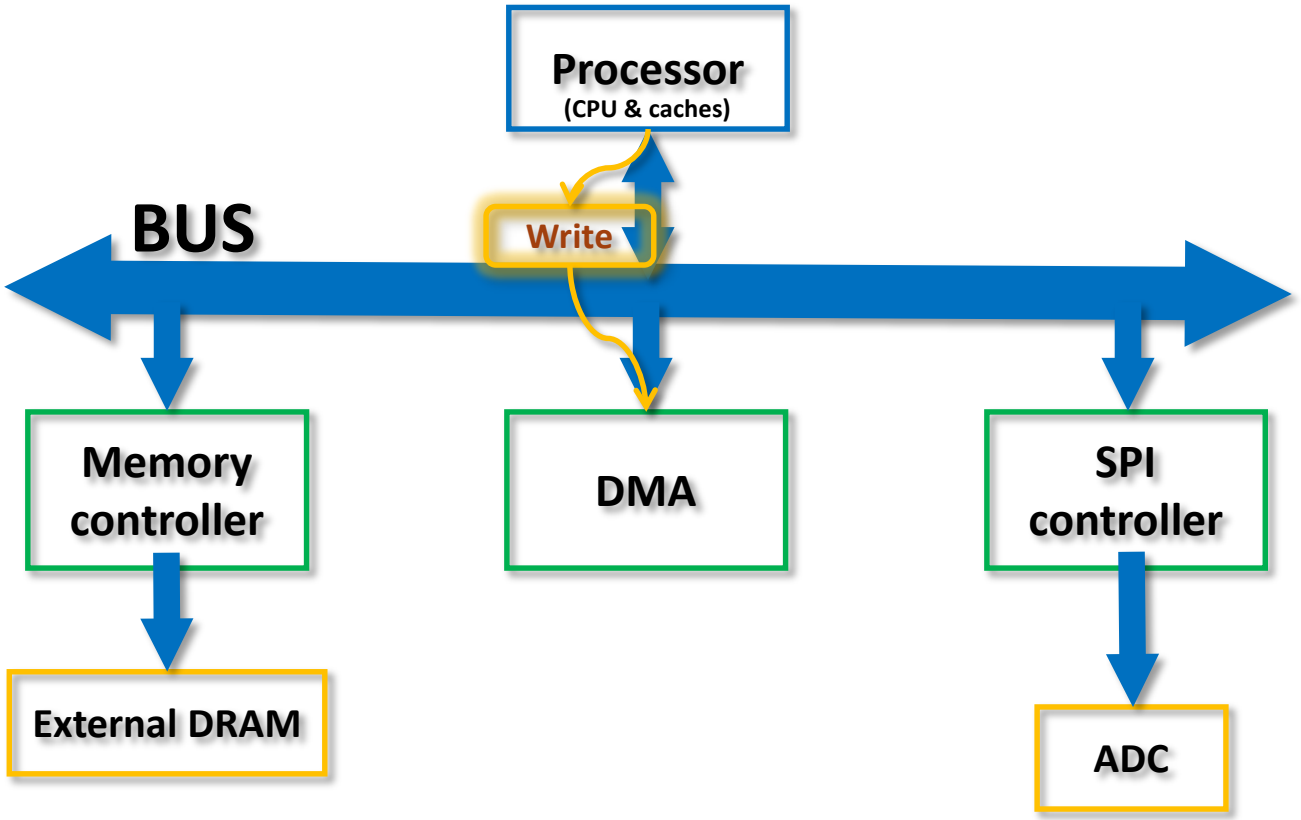


This is how a system with a processor, buttons and LEDs looks like in Vivado



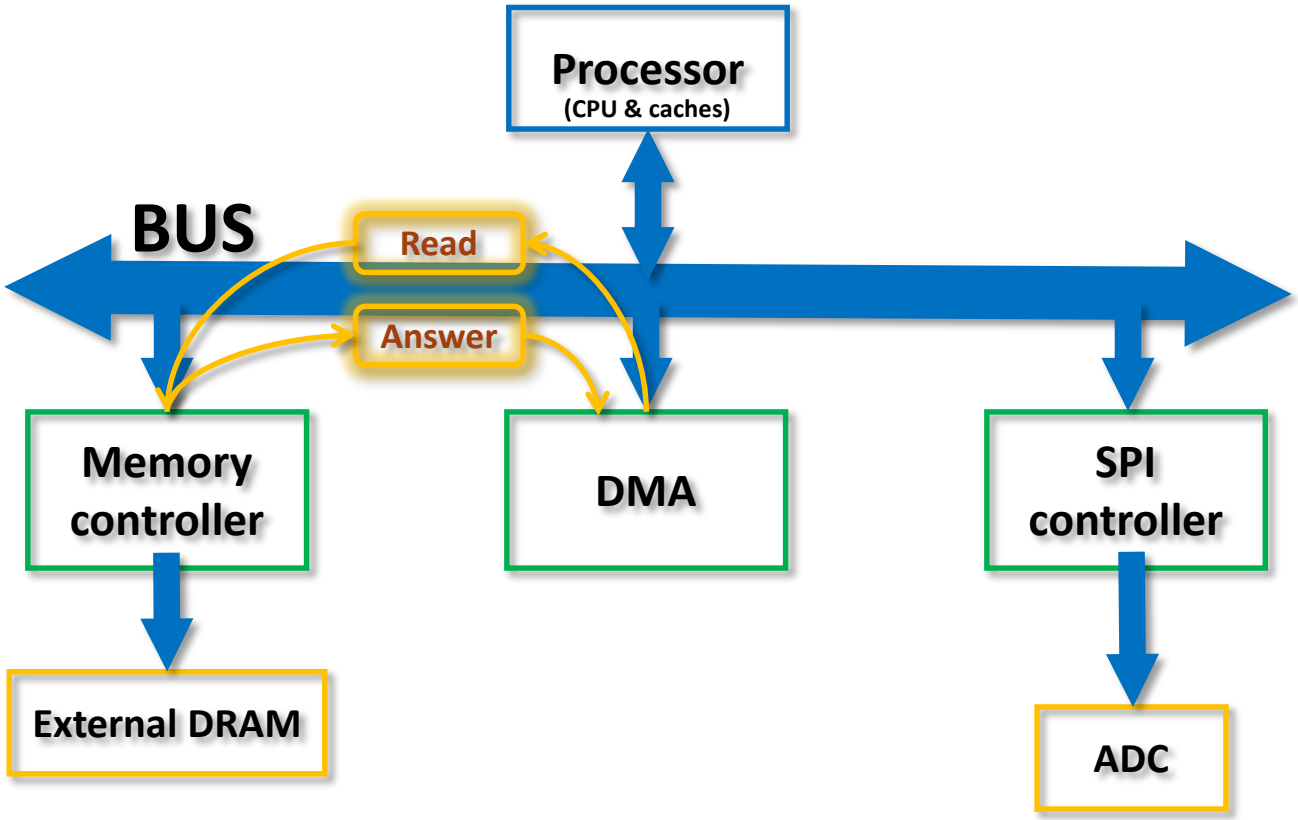
Multiple masters in a bus-based system

- The processor programs a direct memory access (DMA) module to copy an area of memory into another
  1. Proc. sends the transfer parameters to the DMA over the bus
- The DMA sends read/write commands to the memory over the bus
  1. DMA sends read command to the memory
  2. Memory replies with the value in the requested address
  3. DMA sends write command to the memory



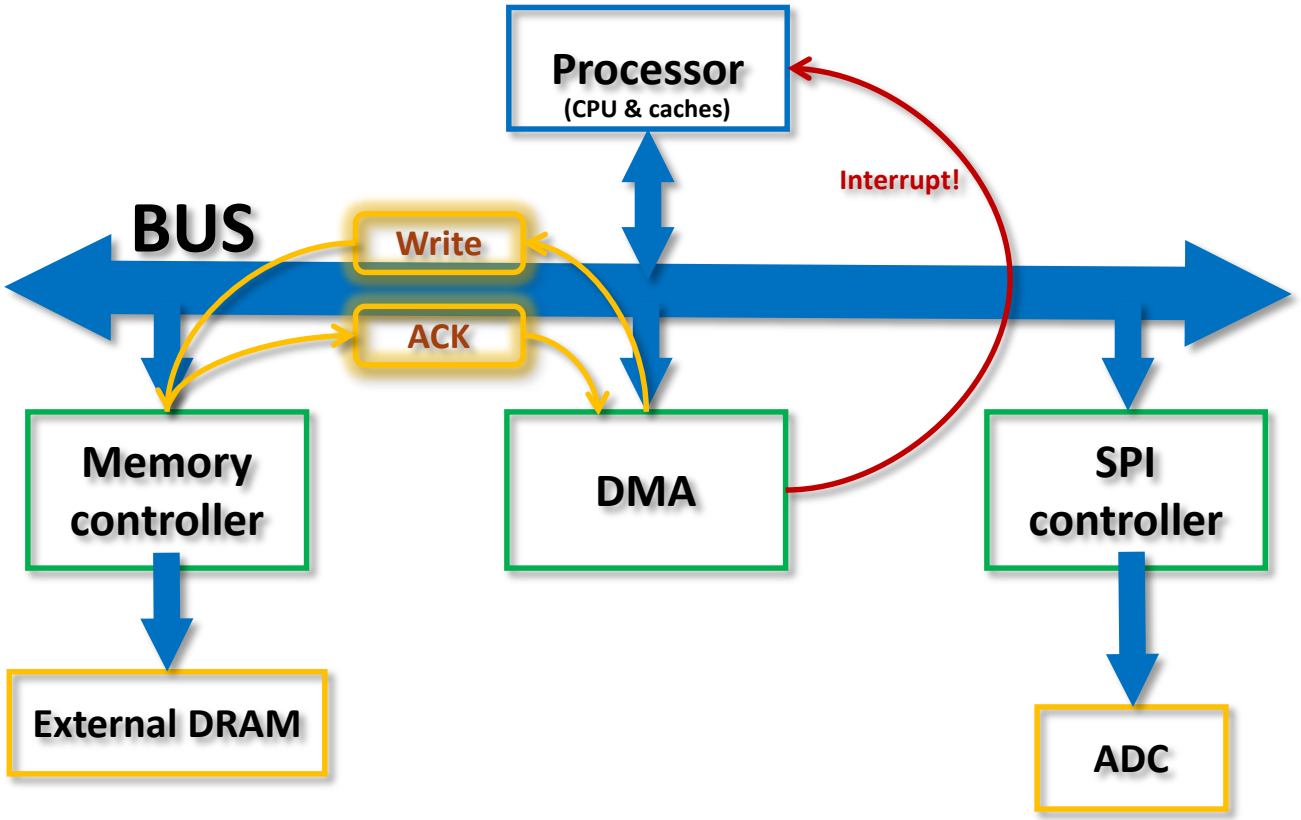
Multiple masters in a bus-based system

- The processor programs a direct memory access (DMA) module to copy an area of memory into another
  1. Proc. sends the transfer parameters to the DMA over the bus
- The DMA sends read/write commands to the memory over the bus
  1. DMA sends read command to the memory
  2. Memory replies with the value in the requested address
  3. DMA sends write command to the memory



Multiple masters in a bus-based system

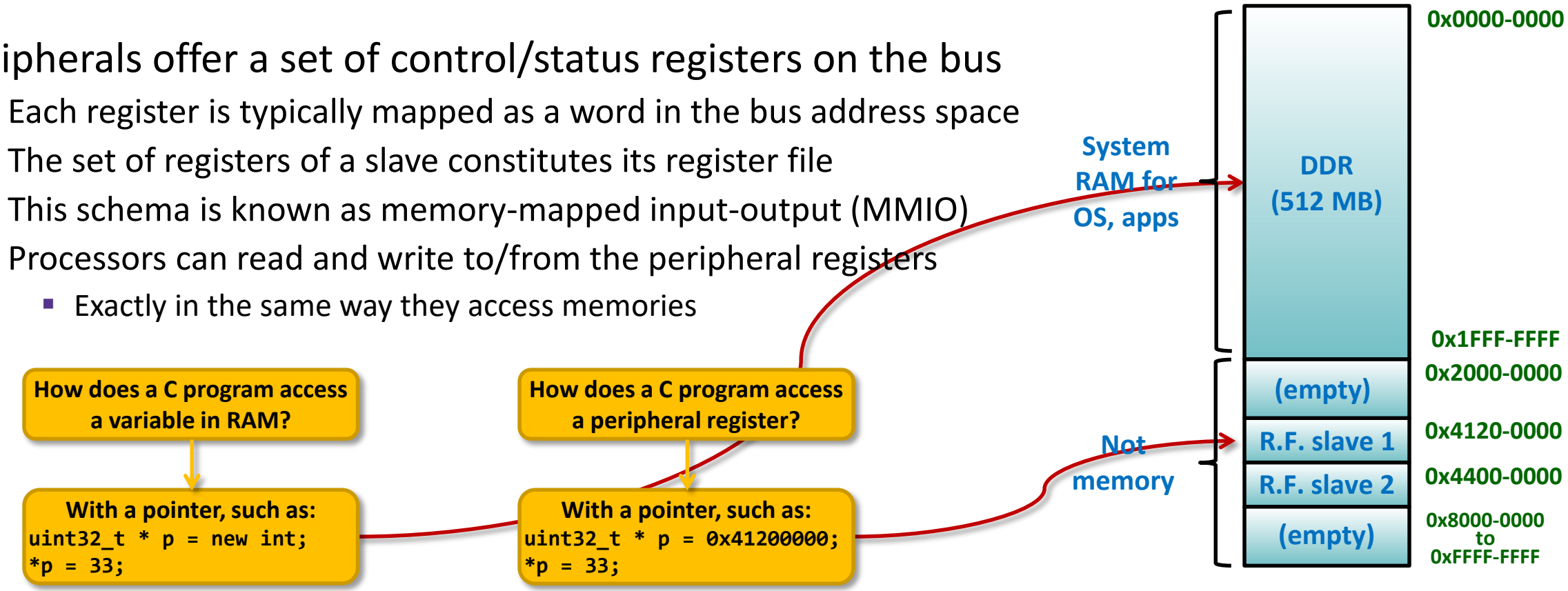
- The processor programs a direct memory access (DMA) module to copy an area of memory into another
  1. Proc. sends the transfer parameters to the DMA over the bus
- The DMA sends read/write commands to the memory over the bus
  1. DMA sends read command to the memory
  2. Memory replies with the value in the requested address
  3. DMA sends write command to the memory



The DMA is both:
- a slave (with respect to the processor)
- a master (with respect to the memory controller)

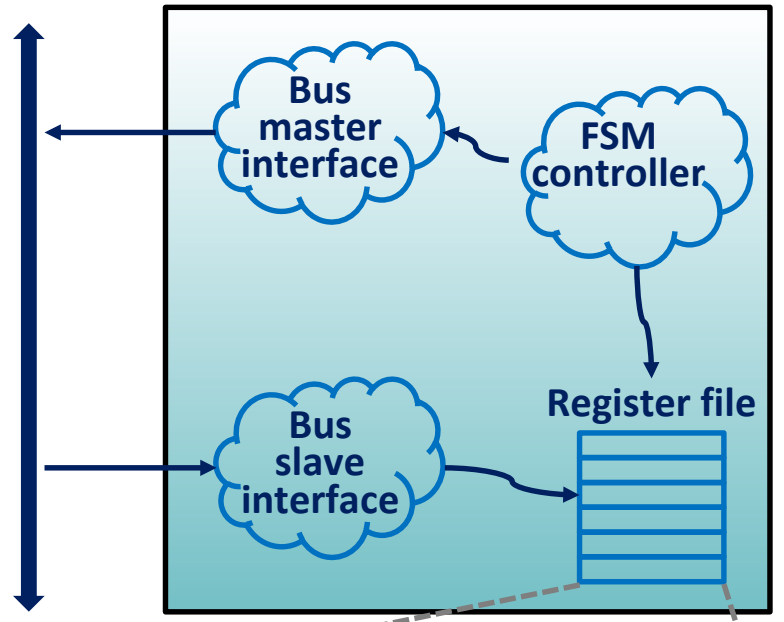
Address space & memory-mapped I/O

- A bus represents an address space
  - All the slaves connected to the bus respond to a range of addresses
  - Memories are mapped on a range of addresses
    - A 512 MiB DRAM will occupy a range of 536 870 912 bus addresses
- Peripherals offer a set of control/status registers on the bus
  - Each register is typically mapped as a word in the bus address space
  - The set of registers of a slave constitutes its register file
  - This schema is known as memory-mapped input-output (MMIO)
  - Processors can read and write to/from the peripheral registers
    - Exactly in the same way they access memories



Anatomy of a bus-based peripheral

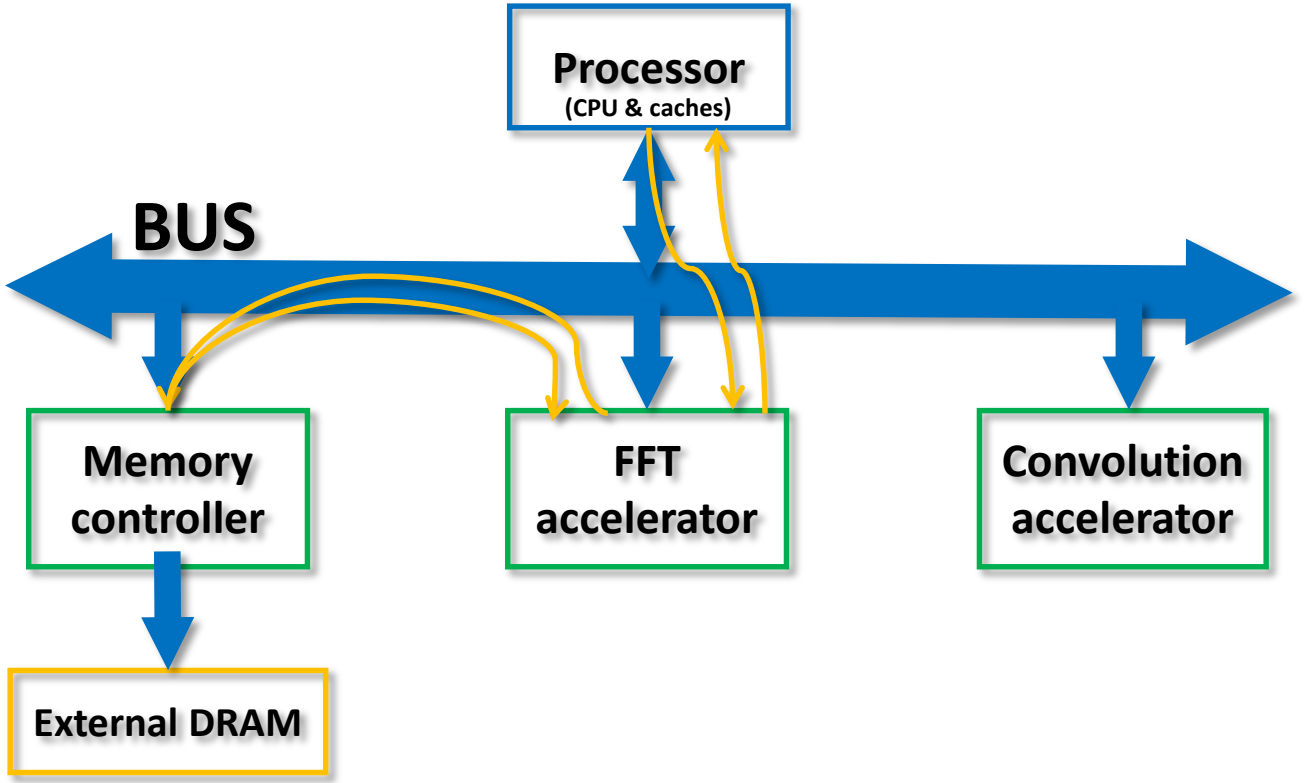
- A memory-mapped peripheral contains a register file
  - And a slave interface accessible to the processor thru the bus
- The processor can read and write the register file
  - To control the peripheral and retrieve its status
- The peripheral has some control logic
  - The controller is connected to the registers and reacts to commands
    - The registers can be connected to the FSM as wires
  - The controller can write into the registers, which will be read by the processor in the future
  - The controller implements the task of the peripheral
    - Using a master interface to access memories or other peripherals
- A peripheral that sums the elements of a vector could have registers for:
  - The starting address of the vector, its length, and the result
  - A bit for the processor to start a computation
  - A bit to signal the end of the computation



	Index	Name	Direction
(0x4120-0000)	0	Address	Input
(0x4120-0004)	1	Length	Input
(0x4120-0008)	2	Result	Output
(0x4120-000C)	3	Start	Input
(0x4120-0010)	4	Done	Output

Accelerators as a special type of peripheral in a SoC

- Accelerators are a type of peripheral that synchronize with a processor to execute specific tasks
  - Often they present a slave and a master interface on the bus
  - The slave interface allows the processor to send commands to the accelerator (e.g., start a task with this data)
  - The master interface allows the accelerator to access the resources required for its computation
    - For example, an accelerator may access data in the memory without processor intervention
- Accelerators speed up specific tasks and/or reduce energy consumption



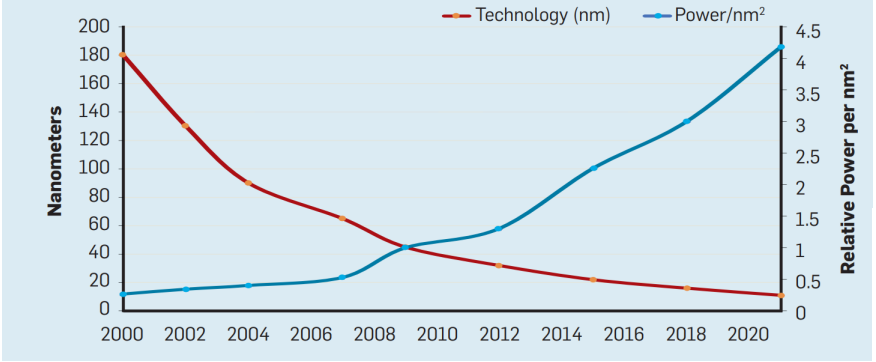
1. Processor sends command to FFT accelerator
2. Accelerator sends read/write requests to the memory
3. The memory responds to the queries from the accelerator
4. The accelerator signals to the processor the end of the task (via interrupts or a status register)

Performance and energy efficiency challenges

Current performance and energy efficiency challenges: An opportunity for domain-specific architectures (DSAs)

- End of Moore's law
  - End to 50 years of exponential density increase
    - Density will still increase at least into early 2030s
- End of Dennard's scaling (power density)
  - We can integrate more transistors than we can afford to power!
  - (Dark silicon era)

Figure 3. Transistors per chip and power per mm².



Source: [1] *"A new golden age for computer architecture"* https://dl.acm.org/doi/10.1145/3282307


DOI:10.1145/3282307

Innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way.

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

A New Golden Age for Computer Architecture

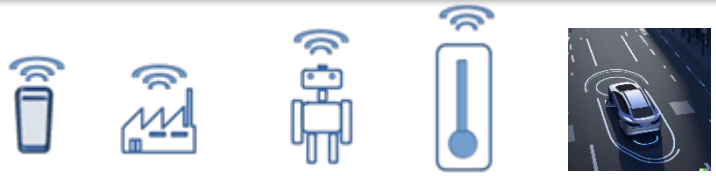
More and larger data centers!

Data centres account for 4% of Swiss electricity usage

swissinfo.ch, January 8, 2023

Significant cost of manufacturing

Plus billions of connected devices!

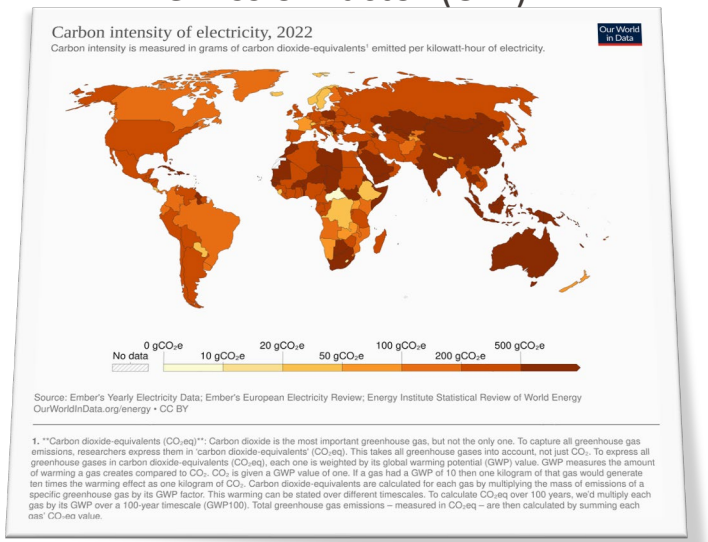


Approximation of CO_2eq emissions cost (rough):

- **For servers: ~50 % manufacturing, ~50 % use**
- **For user devices: ~75 % manufacturing, ~25 % use**

What can be done to reduce the cost of computation?

- Migrate tasks to DCs with lower carbon emission factor (CEF)

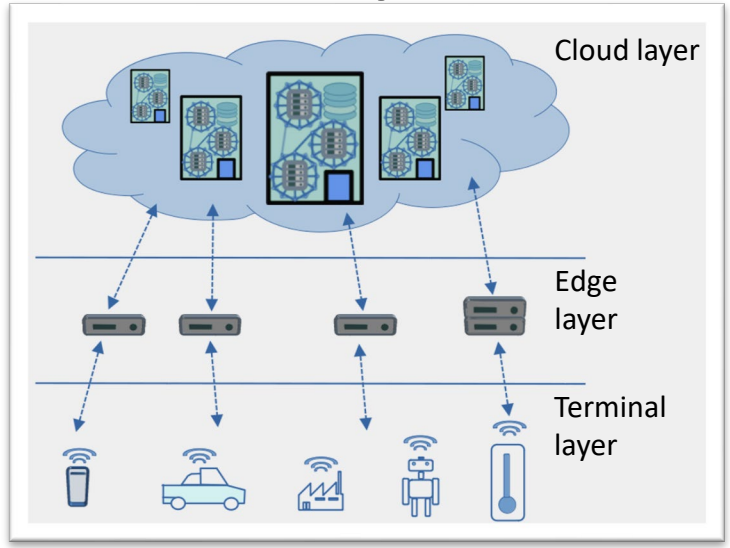


How much overhead from data/task migration?

- Improve DC efficiency

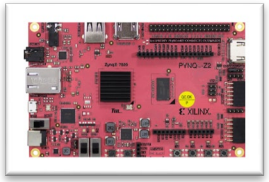
EPFL's new DC in the CCT building with PV generation, water cooling and heat recovery for heating of the campus

- Multi-scale computing systems
  - Distribute workload from terminals to cloud
  - Improved latency
  - Better privacy
  - Avoiding CO2 peaks in the DC



Source: Dr. Xavier Ouvrard, EcoCloud[1]

- Use accelerators for each specific workload (GPUs, FPGAs, ASICs)



Consider shifting from "time-to-completion" to "energy-to-completion"!

Example of accelerator for ultra-low power biomedical devices

Complex-valued FFT (1D)	ARM Cortex-M4 (cycles)	VWR2A* (cycles)	Speed-up
512	47926	7125	6.7 x
1024	84753	12405	6.8 x
2048	219667	30217	7.3 x

	ARM Cortex-M4 (uJ)	VWR2A* (uJ)	Energy Savings
App 1	0.74	0.26	64.7 %
App 2	0.74	0.13	82.9 %
App 3	1.1	0.47	56.0 %

[1] "Special session: Challenges and opportunities for sustainable multi-scale computing systems," X. Ouvrard, et al. ESWeek, 2023.
[2] "VWR2A: A Very-Wide-Register Reconfigurable-Array Architecture for Low-Power Embedded Devices," B. Denkinger, et al. DAC, 2022.

Domain specific architectures

- Why can DSAs improve performance and energy efficiency? From [1]:

1. DSAs can exploit most efficient forms of parallelism for each domain.
   - E.g., SIMD for vector code or VLIW for DSP algorithms.

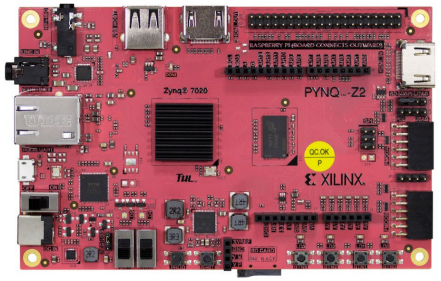
2. DSAs can use memory hierarchies more efficiently.
   - E.g., for known memory patterns, software-controlled memories can be more energy efficient than caches.

3. DSAs can use less precision when adequate.
   - E.g., for DNN inference, often small bit-widths and FxP numbers are enough.
   - In contrast, general-purpose CPUs offer a limited set of numeric formats that may not be optimal for each case.

4. DSAs can benefit from programs written in domain-specific languages (DSLs).

Results of the final projects of last year

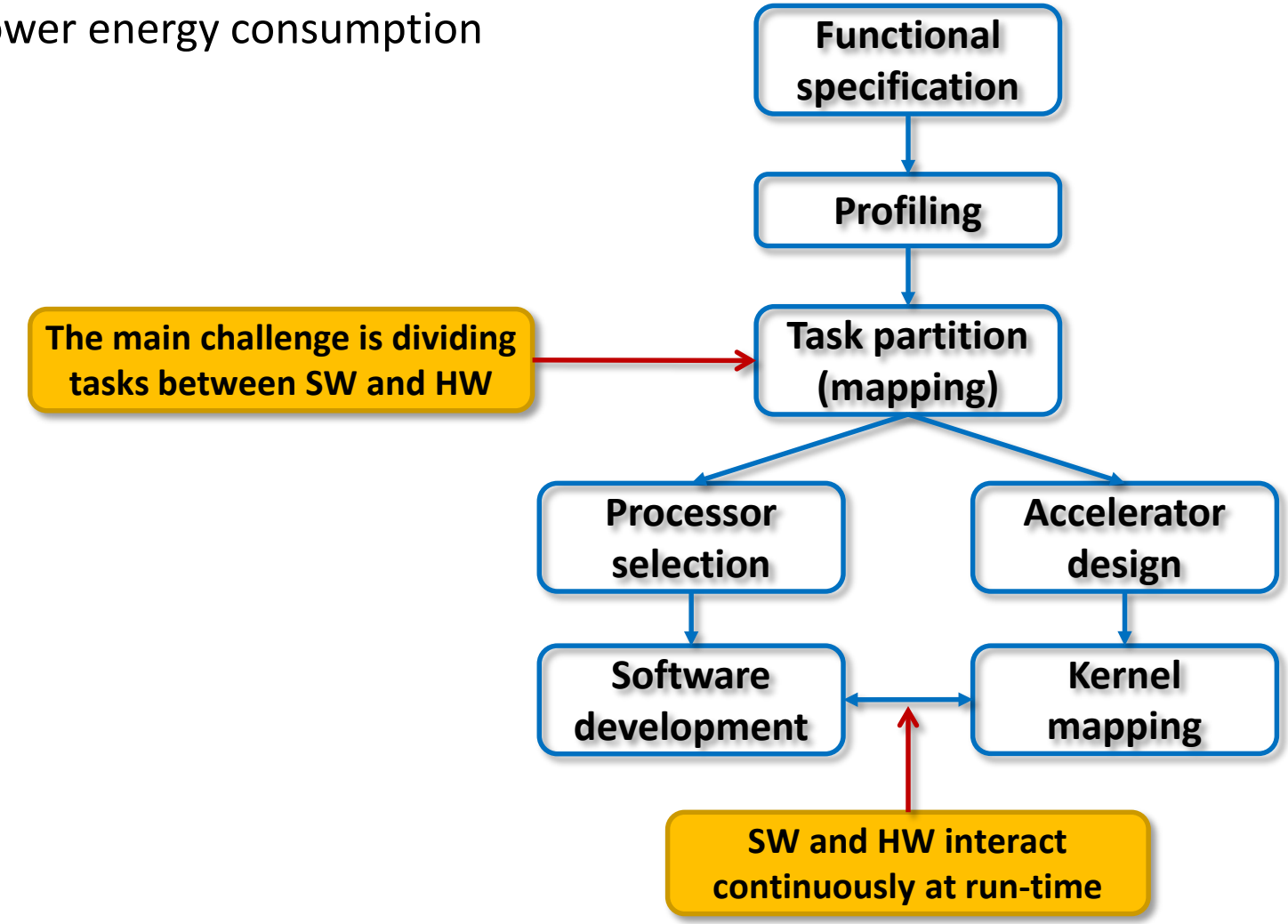
An experiment on genomic sequence alignment



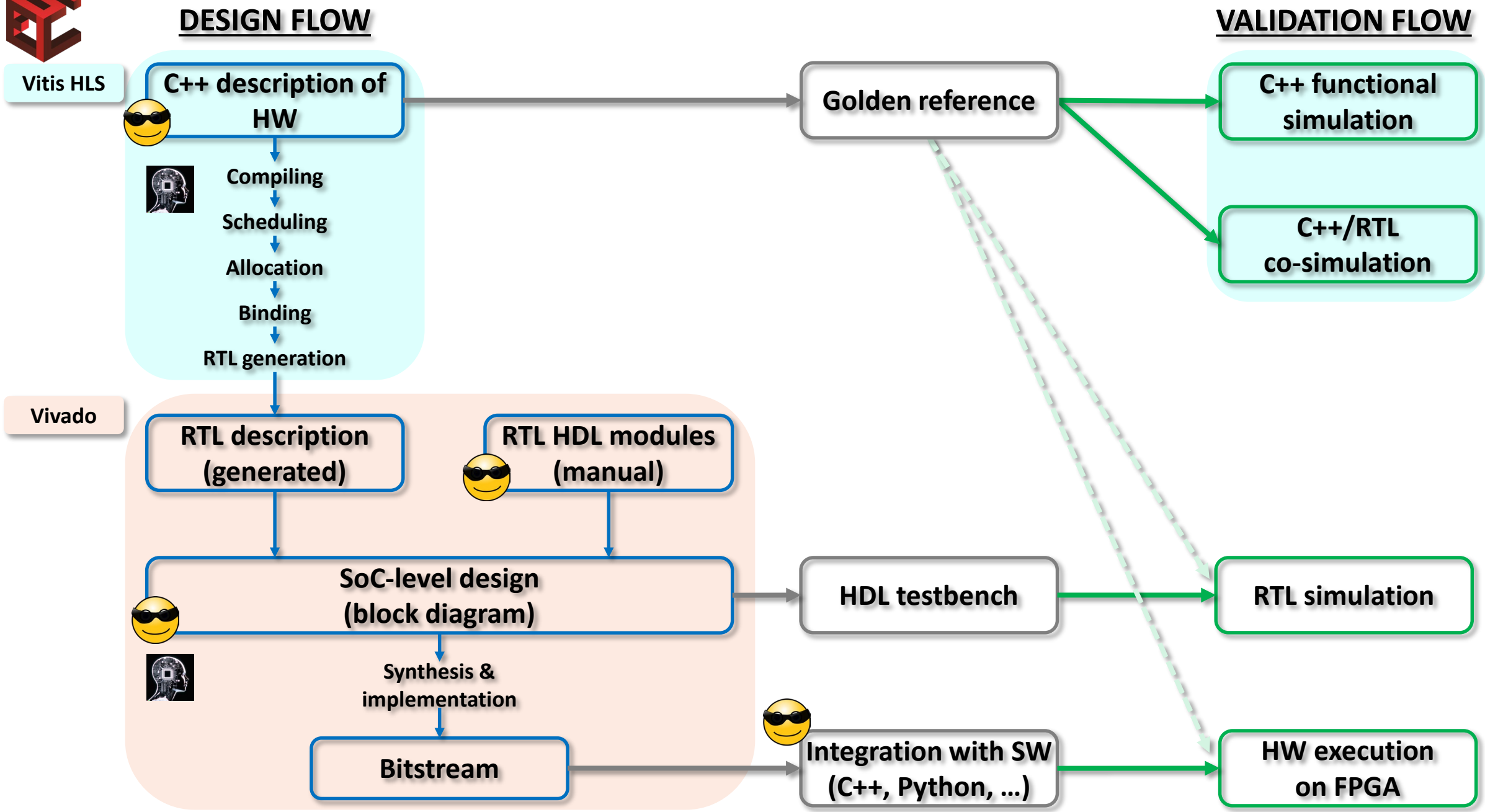
HW-SW co-design flow with the Zynq 7000 SoC

HW-SW co-design flow for SoCs

- SoC design demands careful partition between SW and HW tasks
  - To improve performance
  - To lower energy consumption

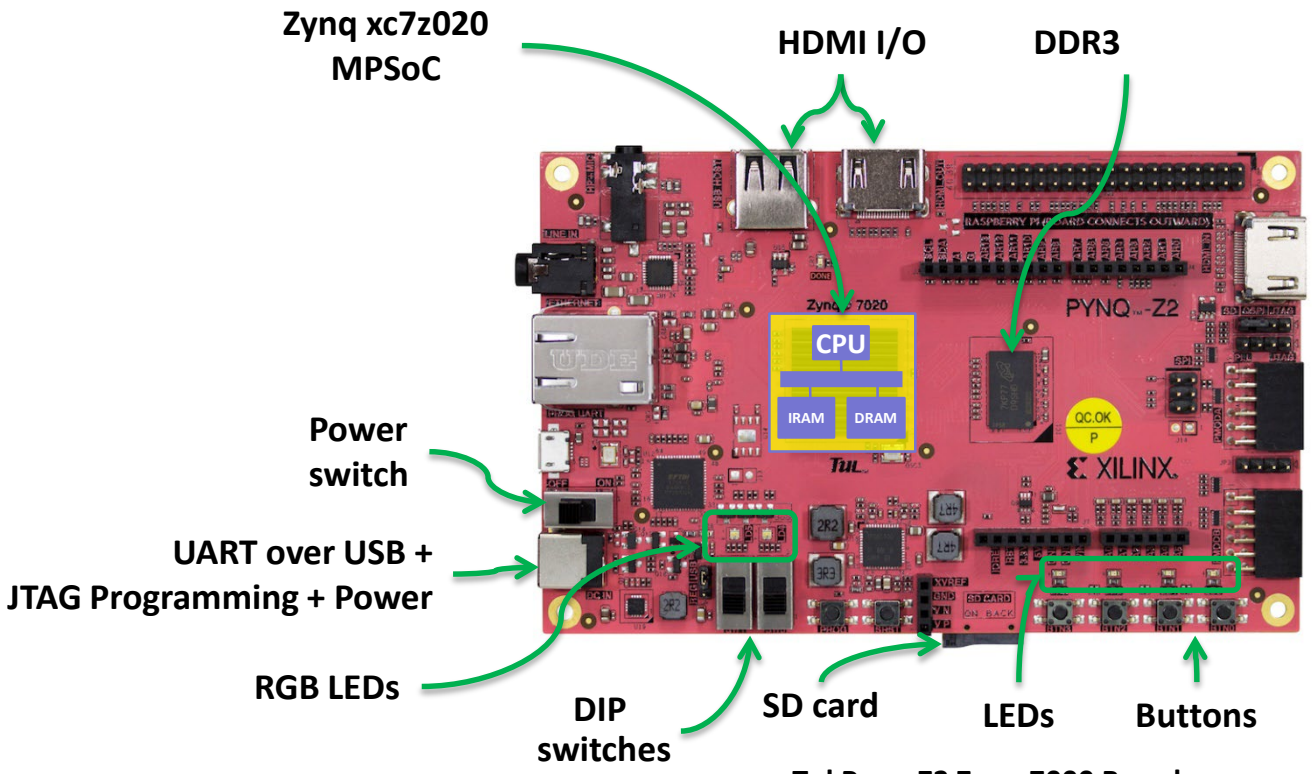


High-level synthesis (HLS) design flow for HW accelerators



Example of SoC: Xilinx Zynq-7000

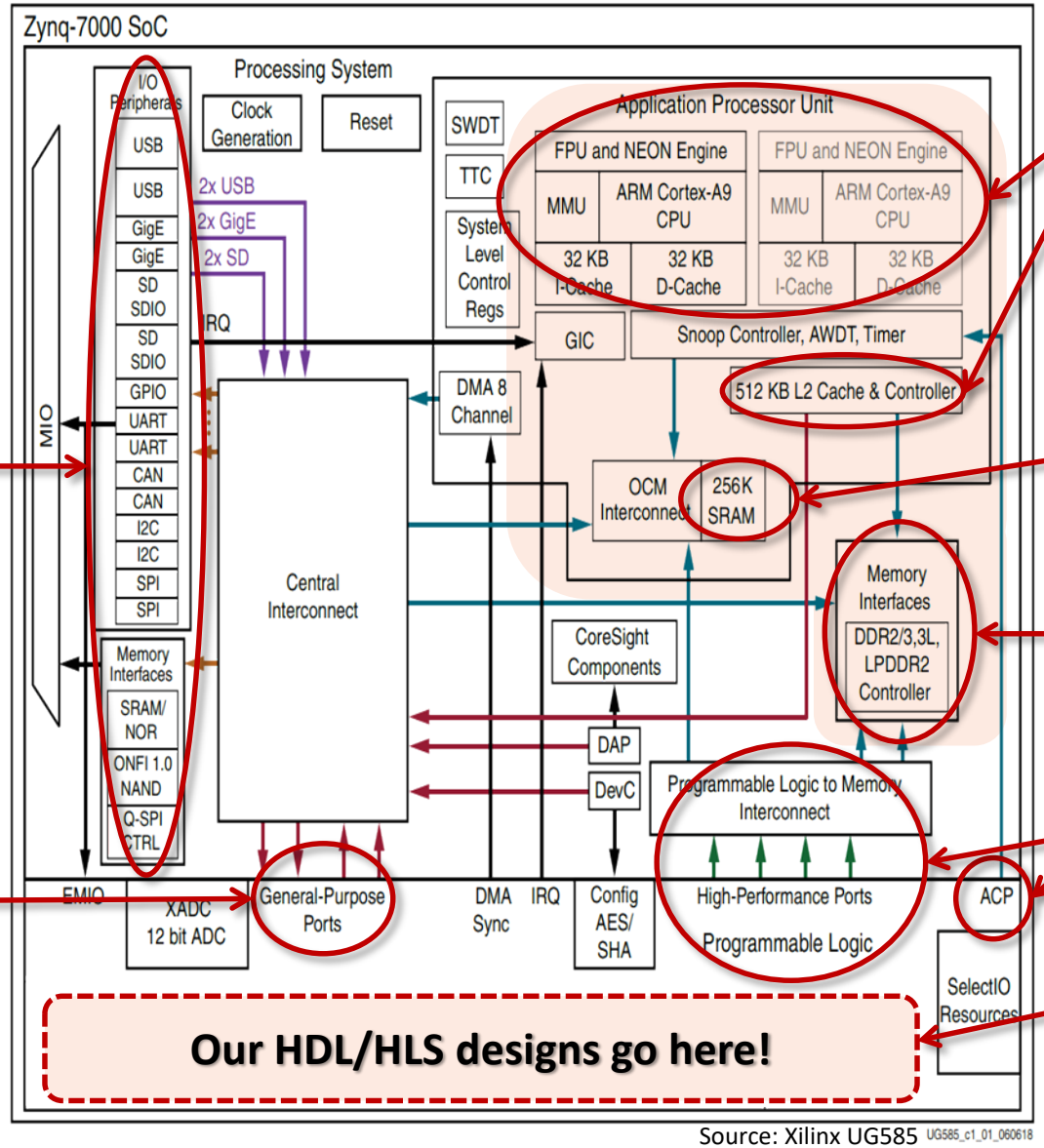
- SoC:
  - 2x ARM Cortex-A9
    - L1 (32 KiB / 32 KiB)
    - L2 512 KiB
  - 256 KiB additional on-chip RAM
  - High-performance AXI4 buses between processors and programmable logic
  - On-chip programming of FPGA logic
  - SPI, I²C, ADC, etc.
- Board:
  - 512 MiB DDR3-SDRAM
  - Serial port + JTAG
  - LEDs, switches and push buttons



Tul Pynq-Z2 Zynq 7000 Board
FPGA: xc7z020clg400-1

During our labs: power, programming and serial port through the single USB port

Architecture of the Zynq-7000 SoC



I/O slow peripherals

Interconnection PS-PL

The processors use these ports to access the registers of our peripherals

Dual-core ARM Cortex-A9
Independent L1 caches
Shared L2 cache

Additional on-chip-memory

Integrated DDR memory controller

Interconnection PS-PL (high-performance)

Programmable logic (FPGA)

Our peripherals use these ports to access the system RAM

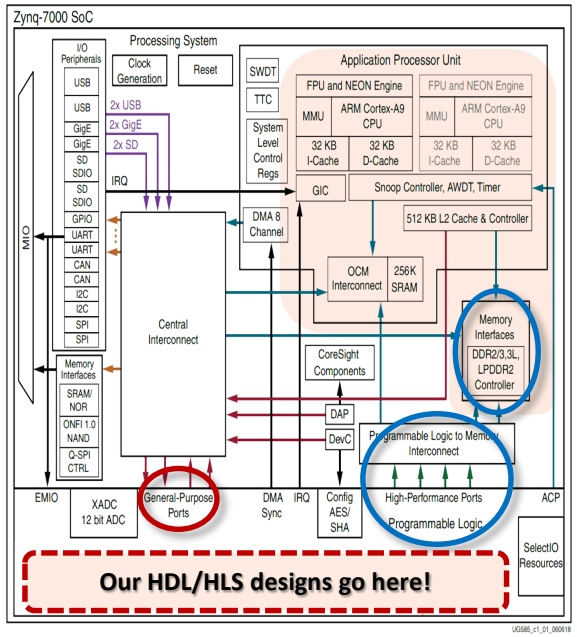
Our HDL/HLS designs go here!

PS: Processing system
PL: Programmable logic

Source: Xilinx UG585 UG585_c1_01_060618

(Physical) Address map of the Zynq-7000 SoC

When we create our first peripheral, check that the address assigned by Vivado belongs to this range



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

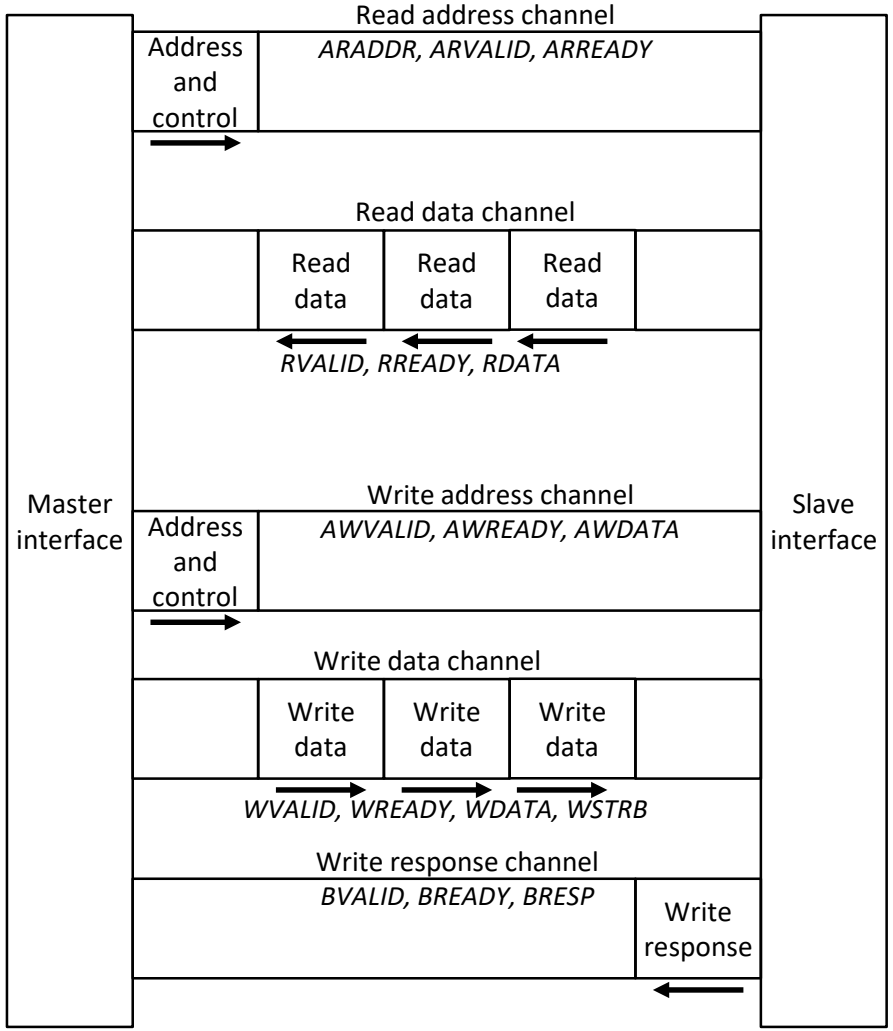
System main memory for use by OS and applications

Mapping of peripherals from the PS side

Source: Xilinx UG585

Introduction to the AXI4 bus

- AXI4 is the native bus for the ARM cores in the Zynq SoCs
  - AXI → Advanced extensible interface
- AXI4 is supported by a large catalog of IP cores:
  - Flexible interconnections
  - Ready-to-use peripherals
- Suitable for high-bandwidth and low-latency applications
  - Multiple outstanding transactions are possible, e.g., to hide high initial latencies
- Synchronous bus
  - All the transactions happen at the rising edge of the clock
  - Separate address/control and data phases
- Separate read and write channels
  - An AXI4 bus can be seen as two independent unidirectional channels linked together
  - Physically: 5 independent channels
- Vitis HLS can generate slave/master peripherals directly from a C/C++ specification

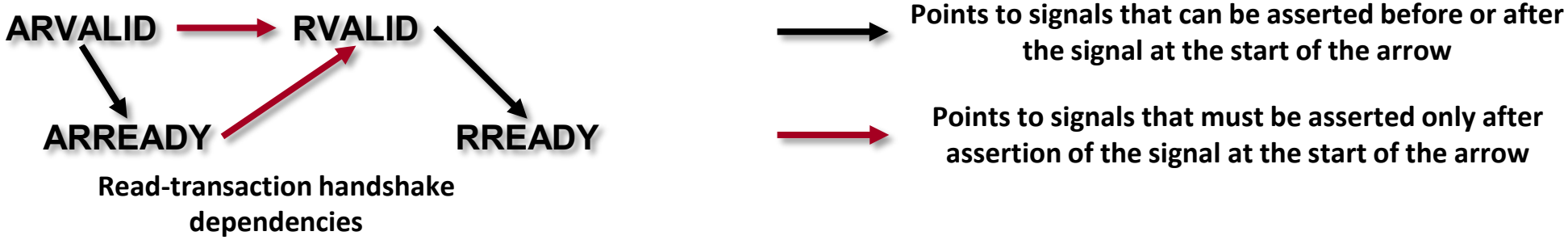


It is possible to have devices that share address channels, but with dedicated data buses!

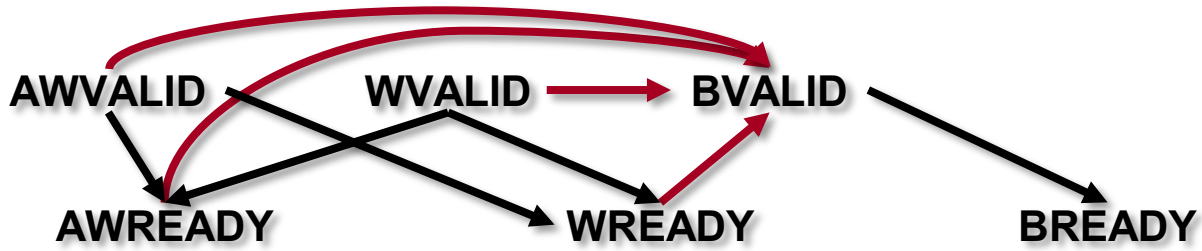
Designing AXI4 peripherals

- All the channels use a two-way handshake mechanism:
  - Based on ready/valid pairs of signals
  - Destination: Assert ready when data can be accepted
  - Source: Assert valid when the information is available
- The transaction ends when both, ready and valid, are HIGH

TRANSACTION CHANNEL	HANDSHAKE PAIR
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY



Read-transaction handshake dependencies



Write-transaction handshake dependencies

Flavors of AXI4 peripherals

- Both masters and slaves can be designed as:
  - ***Lite peripherals***: Low-throughput, memory-mapped
    - Single-access transactions → lower complexity
  - ***Full peripherals***: High performance, memory-mapped
    - Burst transactions. Configurable and flexible
  - ***Stream peripherals***: Point-to-point communication, high-speed streams
    - No explicit memory addresses
      - E.g., an audio stream, a video stream, filter-chain architectures with multiple cascaded devices

- Additional characteristics of AXI4
  - ***Burst transfers***: Only the initial address needs to be issued
    - This saves latency, energy in the lines, and simplifies the logic
  - Multiple outstanding transactions
    - Out of order completion: multiple transaction IDs, completion in-order for each ID
  - Data channels can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide
    - This may be useful to compensate for different device frequencies
    - E.g., FPGA device working at 100 MHz, memory controller at 800 MHz

AXI4 bus hierarchies

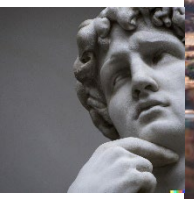
- One single AXI4 Interconnect IP core can create independent paths

- If the design has several high-bandwidth devices:
  - Use several high-performance ports from the PS side
  - Consider using multiple AXI4 Interconnect instantiations to dissect the bandwidth

- Each AXI Interconnect can have multiple slave and master ports
  - Multiple master ports can be connected to multiple slave ports at the PS side
  - But it is (likely) more efficient to instantiate several interconnects
  - If there are more masters than slave ports → shared bandwidth
  - If multiple masters access the same slave → arbitration

Bibliography (in Moodle)

- Pynq-Z2 Reference Manual v1.1
- Zynq-7000 SoC Data Sheet Overview (DS190)
- Zynq-7000 SoC: Technical Reference Manual (UG585)
- AMBA AXI and ACE - Protocol Specification
- Vivado Design Suite – AXI Reference Guide – UG1037 (v4.0)
- Recommended reading:
  - [1] *"A new golden age for computer architecture,"* John L. Hennessy, David A. Patterson. 2019. Communications of the ACM 62, 2, 48–60 https://dl.acm.org/doi/10.1145/3282307



Questions?

Prof. David Atienza

EPFL – Embedded Systems Laboratory
david.atienza@epfl.ch