EPFL

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

Session 5
Virtual memory, memory for DMA and cache coherence

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

Roadmap

- Previous class: memory management & optimizations
  → **Programmable Logic** (accelerator) point of view
- This class: memory management & optimizations
  → **Processing System** (processor & OS) point of view

Session objectives

Virtual Memory (VM) and address space protection

Virtualization

Accessing peripheral registers with MMIO from the address space of a user-level Linux application

Allocate memory suitable for **Direct Memory Access (DMA) by peripherals**

Memory coherence

- Between the accelerator accessing the DDR and the processors using a cache

Hierarchies

Methods to maintain **cache coherence** in the Zynq FPGAs

Physical memory map of the Zynq 7000

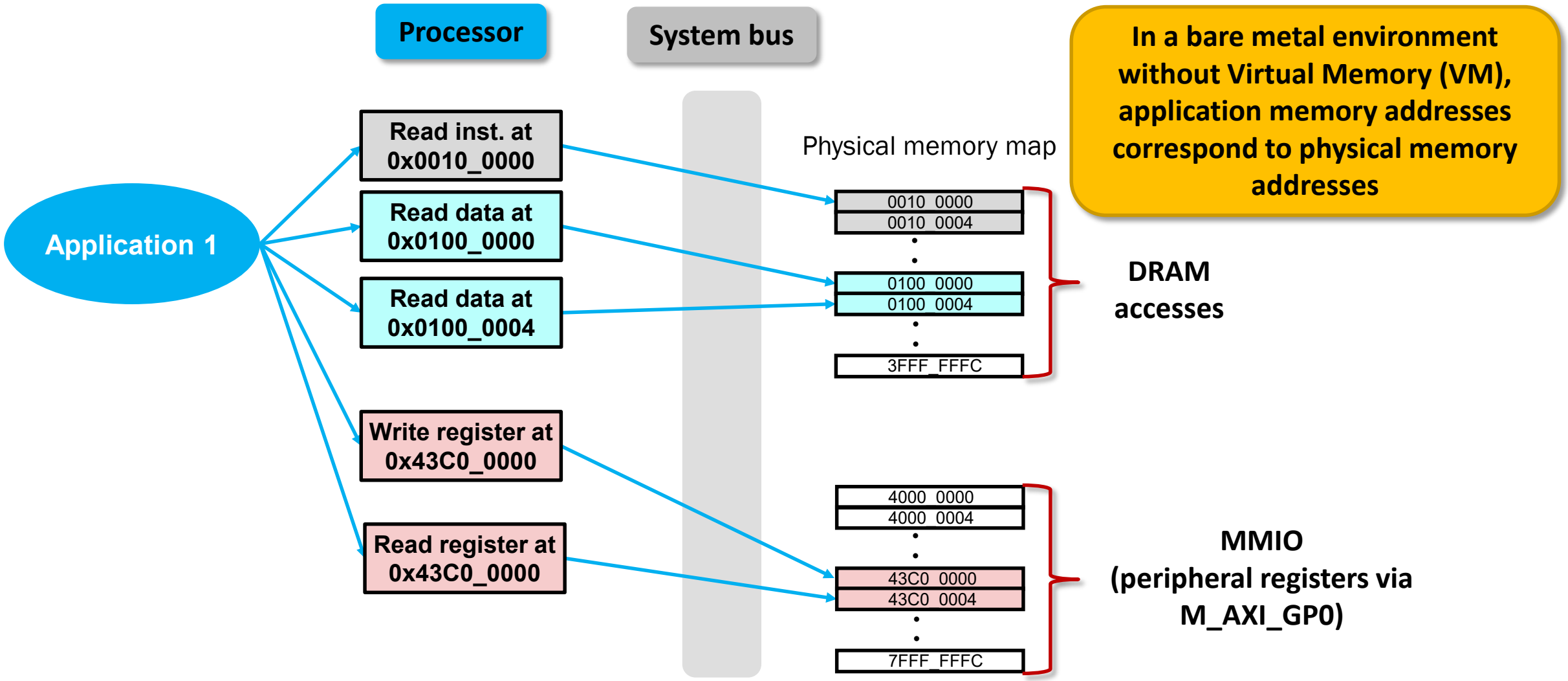
Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and 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

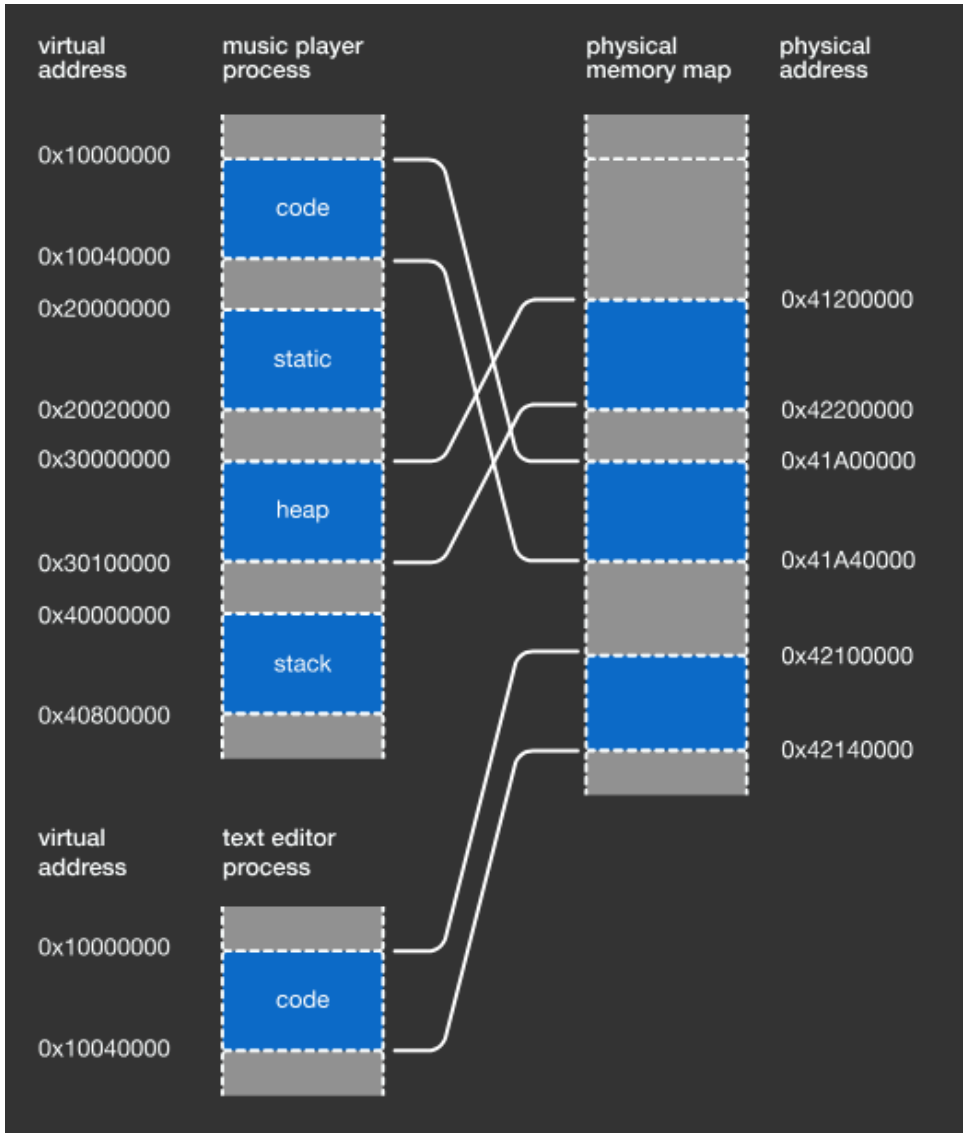
Source: Xilinx UG585

Memory access from the applications (without VM)



Problem: Indiscriminate memory accesses

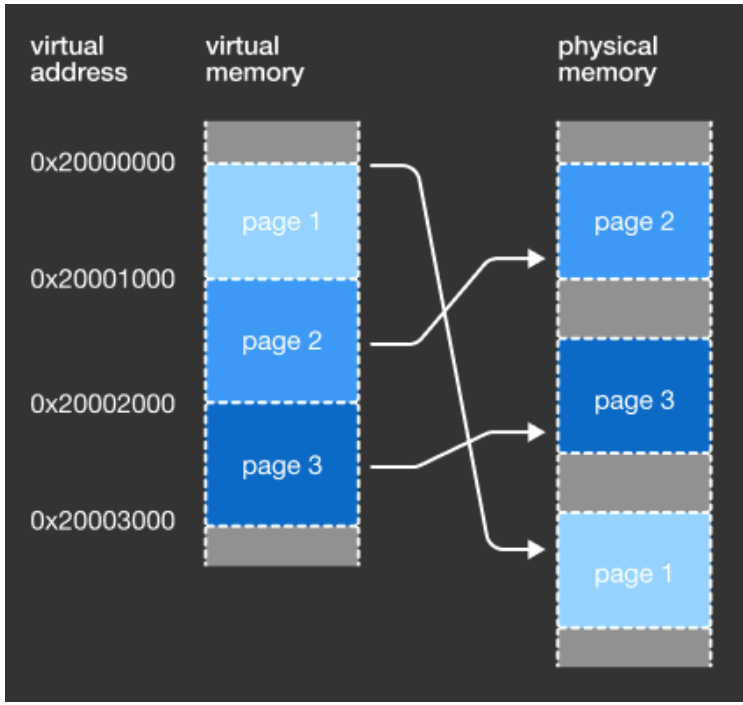
- If all the applications running on a system can access all the memory space…
  - **Any application can modify, spy or corrupt** the state of the others!
  - If one application uses a wrong pointer, the complete system is compromised
- The solution is to isolate each application inside its own *virtual memory space*
  - Implemented in all major processors and operating systems: Windows, Linux, Unix, MacOs, iOS, Android



Source: Brilliant.org

Solution: Virtual memory

- Each application believes to be alone in the system
  - No application can access the memory of another
  - Inter-process communication (IPC) mechanisms
- Addresses of an application **are not related** to the physical memory map
  - An application cannot know the physical location of its data or code!
- **OS** builds a **translation table** for each application
  - Translation is typically based on **pages with fixed sizes** (**4 KiB**, 1 MiB, 1 GiB)
- **Processor** translates **virtual addresses** into **physical addresses**
  - For every application access to data or code!
  - Translation overhead (time, area, energy) for every memory access
- VM enables better management of available memory
  - Allocations from an application can be scattered over the physical address space
  - *Contiguous addresses in application virtual address space do not necessarily correspond to contiguous addresses in the physical address space!*

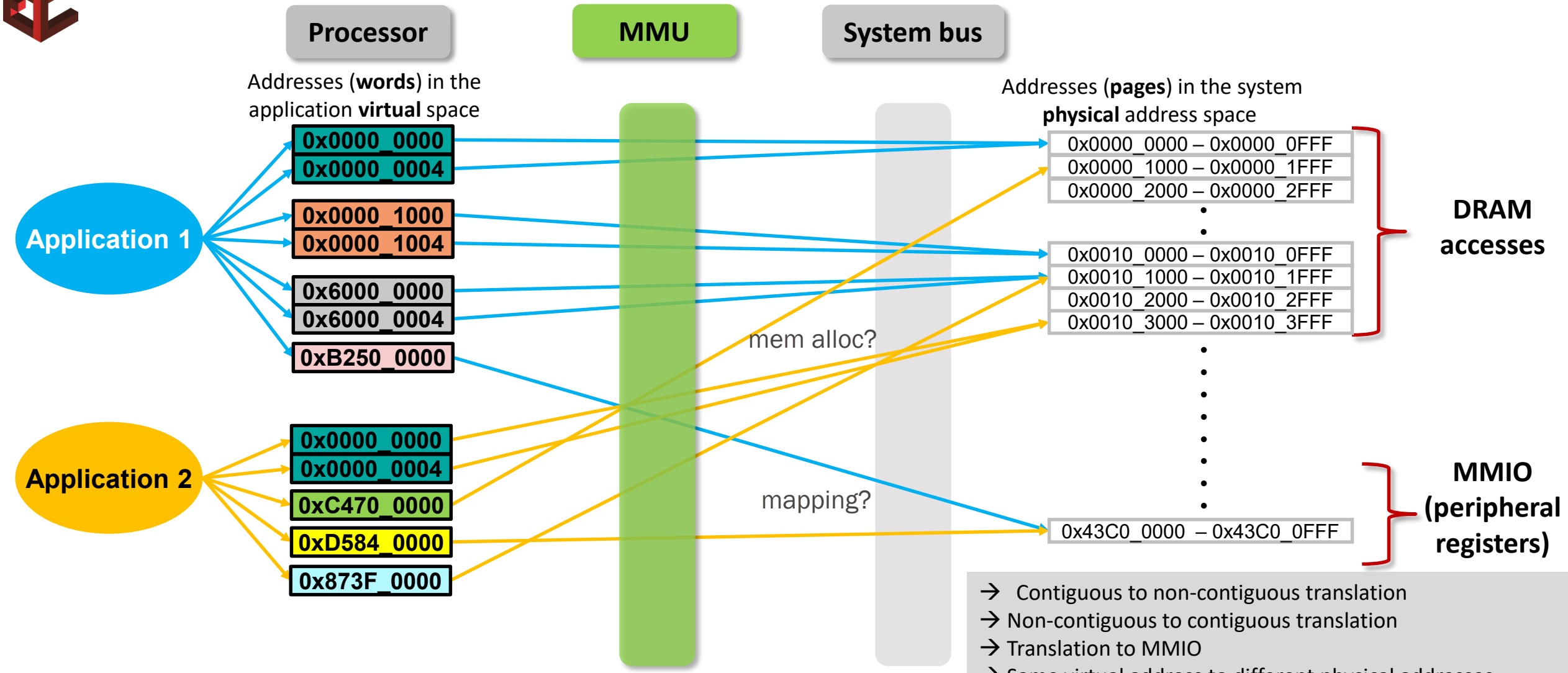


Source: Brilliant.org

Bonus

Can you imagine problems for our peripherals?

Application memory accesses with virtual memory



→ Contiguous to non-contiguous translation
→ Non-contiguous to contiguous translation
→ Translation to MMIO
→ Same virtual address to different physical addresses
→ OS-level memory allocation happens at page level
→ Different virtual addresses from two apps can map to the same physical address (MMIO or shared memory)

Where does the cache of the processors fit in this schema?

Bonus question: Can you imagine how does swapping work?

MMU = Memory Management Unit

Implications when accessing peripherals from Linux

- Our application cannot access physical addresses directly…
  - We cannot access the registers of the slave peripherals!

What do we do?
`cma_mmap(…)`
`cma_get_phy_addr(…)`

- Memory allocated by our application can be **non-contiguous**
  - E.g., page size in the ARM Cortex-A9 is typically 4 KiB
    - `uint32_t inputVector[8192];`
    - `uint32_t * inputVector = (uint32_t *)malloc(8192 * sizeof(uint32_t));`
  - Both methods allocate 8 pages of 4 KiB, but each page is potentially anywhere in RAM!

What do we do?
`cma_alloc(…)`
`cma_free(…)`

- Our application allocates memory in its virtual address space
  - How do we pass the physical address of a data object to the peripherals?

What do we do?
`cma_get_phy_addr(…)`

- In production systems, we need **device drivers**
  - In the Pynq Linux image, we can **use functions provided by Xilinx**

What do we do?
Kernel space drivers

Mapping peripheral registers into the application virtual address space

- With the Xilinx environment, use the function `cma_mmap(physicalAddress, mappingSize)` to access physical addresses
  - This is not a memory allocation! We are just ***mapping*** the registers of the peripherals (MMIO)
  - Use `cma_munmap(physicalAddress, mappingSize)` to release the mapping

Look into the class CAccelDriver to see how this is implemented

- Alternative: Use `mmap()` to project a file into the address space of the app
  - Normally used for fast I/O on large files (the file appears logically as an array)
  - /dev/mem is a special "file" that represents the physical address space of the system
  - The file /dev/mem is projected into the address space of the application with `mmap()`
    - Using as offset the starting address of the region we want to map
- This method:
  - Requires root privileges
  - In most platforms, cannot be used to access areas of memory managed by the OS
    - Use it for the peripheral registers, which are outside of the RAM address range
  - Works only on embedded systems
    - By default, disabled at the kernel level on non-embedded systems (e.g., PCs)

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

Example of using mmap() to access MMIO

```
15 volatile uint32_t * MapMemIO(uint32_t baseAddr, uint32_t mapSize)
16 {
17   memMapSize = 0;
18   memMapAddr = NULL;
19
20   if ( (memMapFileDesc = open("/dev/mem", O_RDWR | O_SYNC )) == -1) {
21     printf("Error opening file\n");
22     return NULL;
23   }
24
25   memMapAddr = (volatile unsigned int *)
26     mmap(NULL, mapSize, PROT_READ | PROT_WRITE, MAP_SHARED, memMapFileDesc, baseAddr);
27   if (memMapAddr == MAP_FAILED) {
28     printf("Memory mapping failed.\n");
29     close(memMapFileDesc);
30     memMapFileDesc = -1;
31     memMapAddr = NULL;
32     return NULL;
33   }
34
35   memMapSize = mapSize;
36   return memMapAddr;
37 }
```

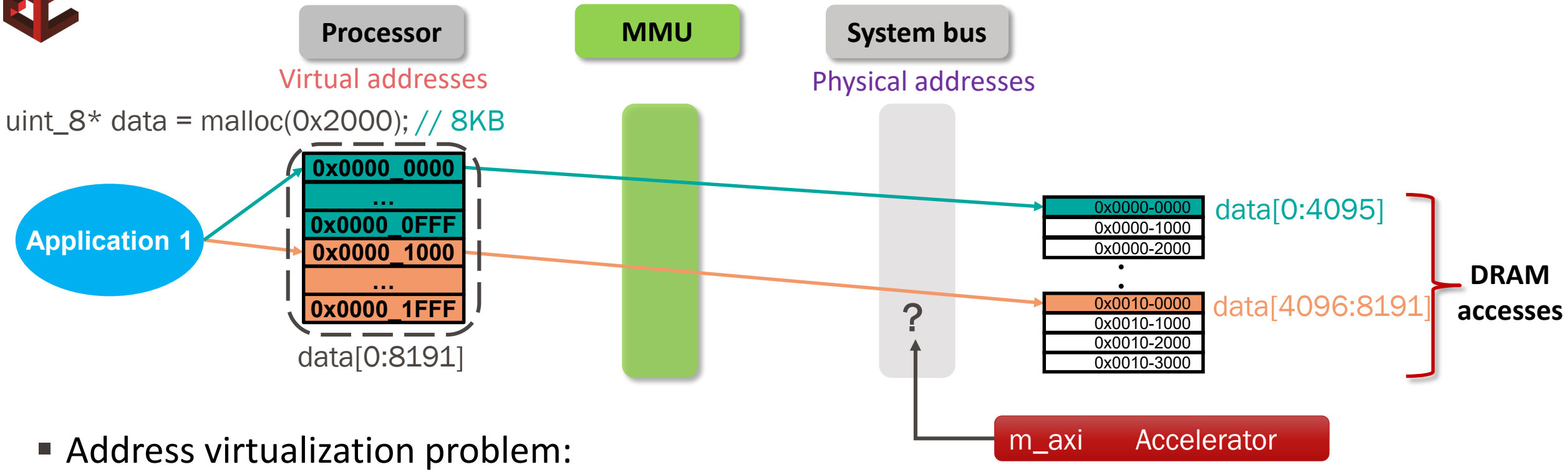
First, open the file /dev/mem

Then, memory-map the file into this application virtual address space

Phy addr to Virt. addr

```
Mmap done. Peripherals mapped from 40000000 to B2DAF000
Buttons at: B2DAF000 - LEDs at: B5DAF000
```

Sharing data with accelerators with VM



- Address virtualization problem:
  - **Applications** see a private **virtual address space**, **not** the **physical address space**
  - **Master peripherals** see the **physical address space**, they **don't know** about the **virtual address spaces**
- Non-contiguity problem:
  - Pages that are consecutive in an application's virtual space may not be consecutive in the physical address space

The MMU is part of the CPU. Our accelerators don't have access to it!

In desktops/servers, peripherals use virtual addresses! → Need for an IOMMU to translate also accesses from the peripherals

MMU = Memory Management Unit

Allocating buffers for data-sharing with accelerators

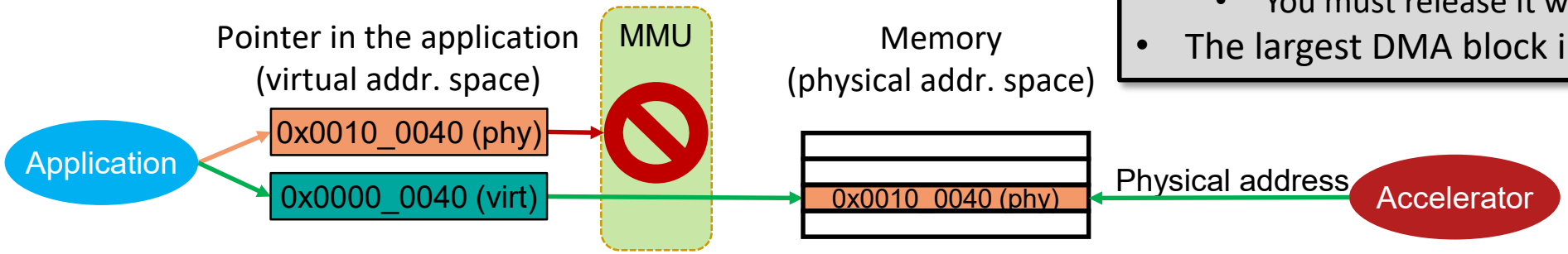
- We use two functions provided by Xilinx to allocate DMA-ready memory
- Allocate a block of ***contiguous*** memory:

```
uint32_t * input1 = (uint32_t *)cma_alloc(LENGTH * 4, MEM_IS_CACHEABLE);
```

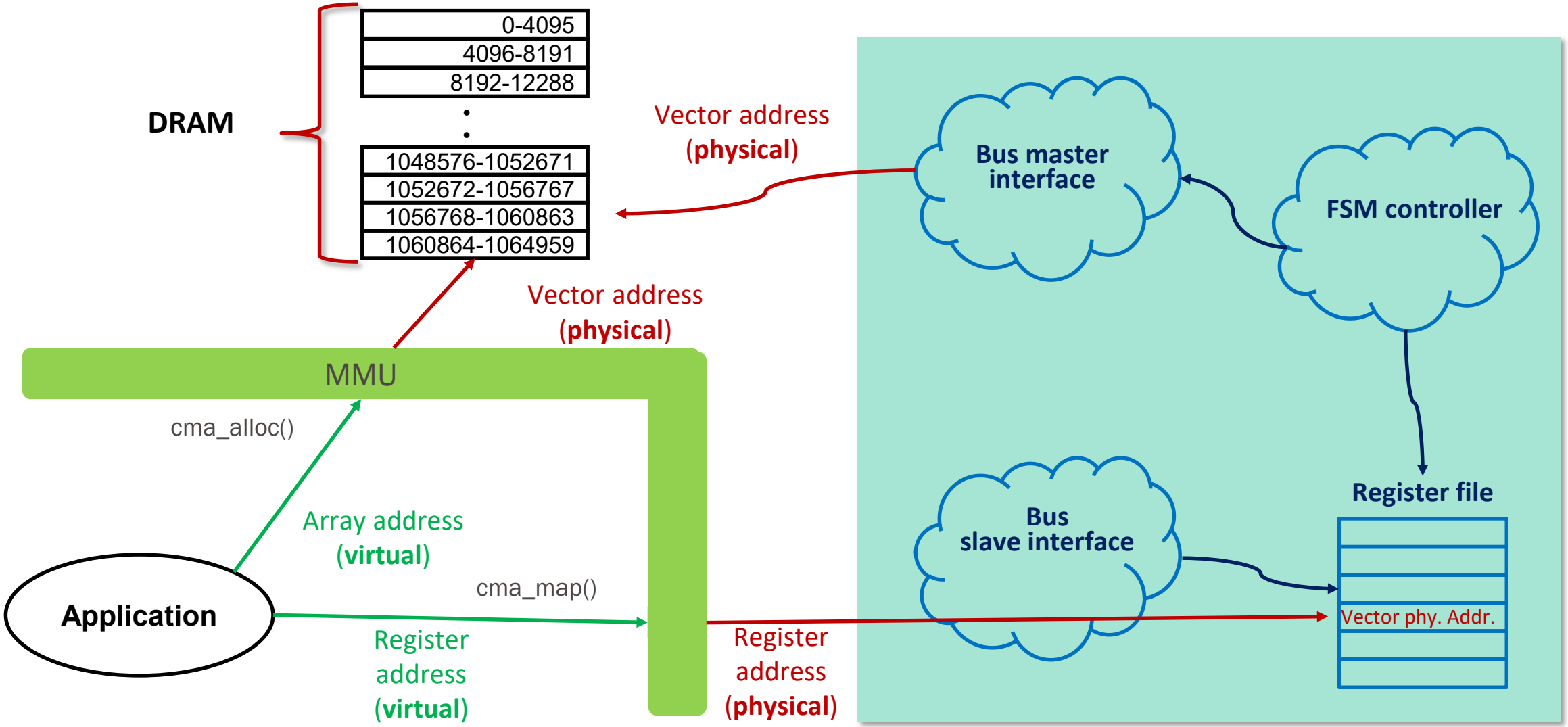
  - input1 is a virtual address suitable *only for the application* allocating the memory
  - Granularity is 4 KiB
- Obtain the ***physical address*** corresponding to the buffer:

```
uint32_t * phyInput1 = (uint32_t *)((uint32_t)cma_get_phy_addr(input1));
```

- The application must never use the physical address!
- The accelerators never use virtual addresses
- DMA memory is a system-wide resource
  - It's not released when the application ends!
  - You must release it with `cma_free()`
- The largest DMA block in the Pynq board is ~32 MiB



How does this look with our accelerators?



Session objectives

Why memory hierarchies?

The “memory barrier” problem reflects the disparity between

- Processor speed
- Memory access speed

In general, the larger a memory, the higher the latency (and energy!) to access it

Concept of memory hierarchy

- Bring the most used data into small and fast memories close to the processor

A memory hierarchy can be

- Transparent: cache memories
- Visible in the programming model: “scratchpad” memories

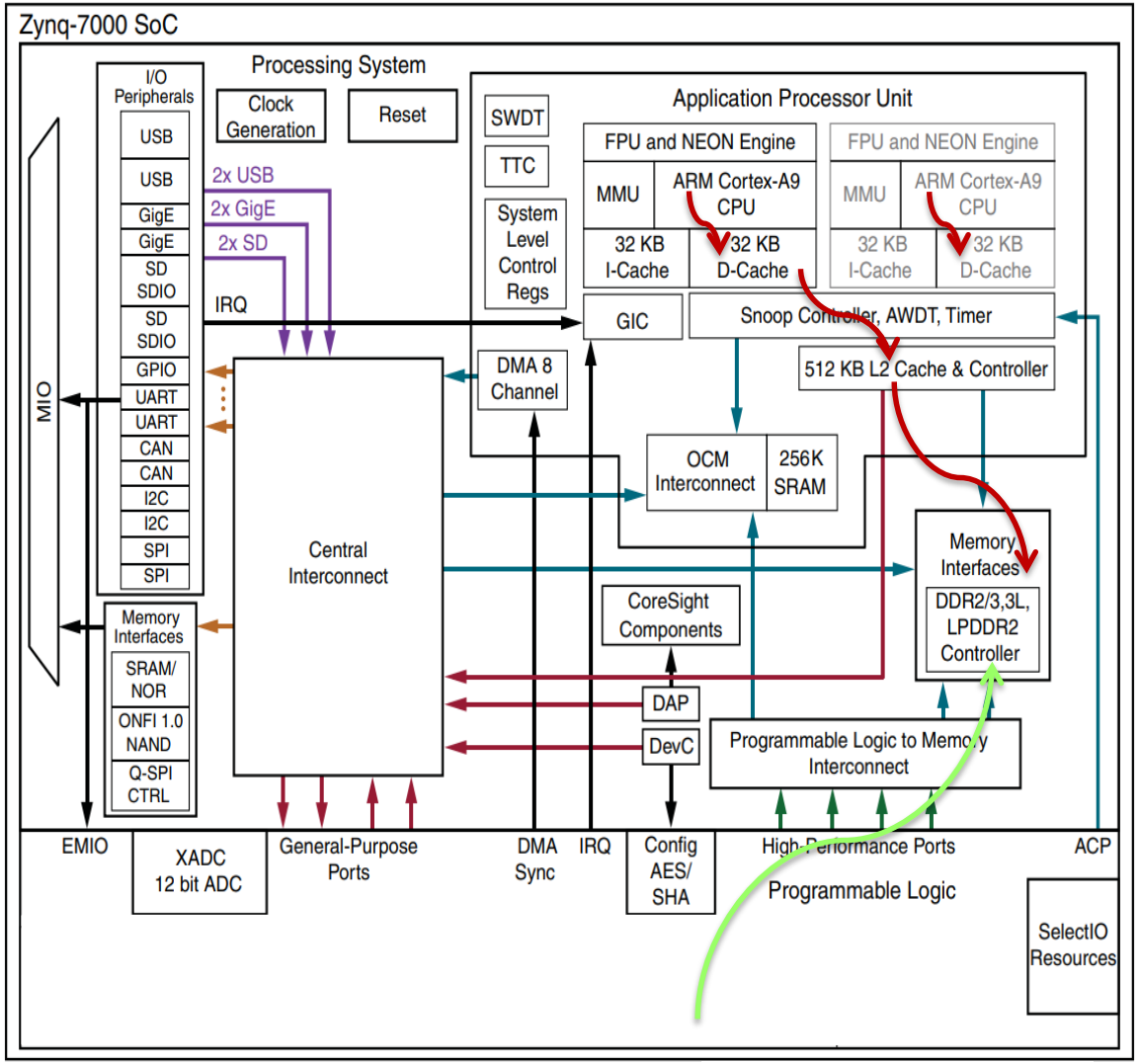
Application processors tend to include instruction and cache memories

- The ARM cores in the Zynq-7000 include two levels of cache memory

Coherence between processor cache and peripherals

- ARM Cortex-A9 cores write data to their caches
  - From the processor registers to the L1 D-cache
  - From the L1 D-cache to the shared L2 D-cache
- Master peripherals access the DRAM directly
  - Memory views can be different!
- Several solutions:
  - Use non-cacheable memory
  - The ACP port connects accelerators in PL to the snoop controller
  - Manage the cache in SW (flush/invalidate)

Why is "`volatile`" not enough?
What is then "`volatile`" for?
Why does "`volatile`" work for MMIO to peripheral registers?



0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Ac
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL M
8000_0000 to BFFF_FFFF	PL		PL	G M_AXI_GP1

Cacheable

Non-cachable

Using non-cacheable memory

- Marks ranges of memory as non-cacheable
  - E.g., the ranges that correspond to the M_AXI_GP0 and M_AXI_GP1 ports for MMIO
- Multiples way of specifying this option
  - In the Pynq boards, specifying `0' to the cacheability parameter of `cma_alloc()`
- Accesses from the processors to non-cacheable mem. are **MUCH** slower
  - Avoid when the processors need to operate heavily on the data
  - May be a good idea to copy into a cacheable buffer, operate, copy back to the non-cacheable one before the HW uses it
- No special measures in the software application
  - Data is always consistent between processors and accelerators
- Particularly useful for intermediate buffers between multiple calls to an accelerator
  - E.g., multiple layers of a neural network

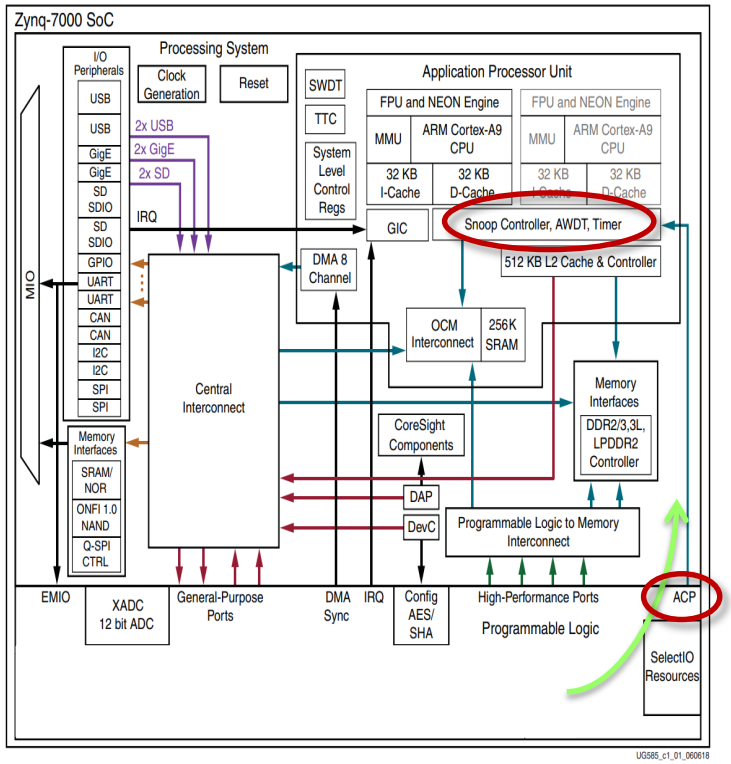
```
./cnnSolver cat.9495.jpg.rgba.planar
OUTPUT: 0.06528854 --> CAT
Conv 0 --> 1764281139 ns (1.764 s)
Conv 1 --> 9103904565 ns (9.104 s)
Conv 2 --> 8283239096 ns (8.283 s)
Conv 3 --> 8171361416 ns (8.171 s)
Conv 4 --> 805582082 ns (0.806 s)
MaxPool 0 --> 26132348 ns (0.026 s)
MaxPool 1 --> 11396990 ns (0.011 s)
MaxPool 2 --> 5817302 ns (0.006 s)
MaxPool 3 --> 2631283 ns (0.003 s)
MaxPool 4 --> 81523 ns (0.000 s)
Dense 5 --> 25690080 ns (0.026 s)
Dense 6 --> 12422 ns (0.000 s)
Total Conv time: 28128368298 ns (28.128 s) 99.7 %
Total MaxPool time: 46059446 ns (0.046 s) 0.2 %
Total Dense time: 25702502 ns (0.026 s) 0.1 %
Total Flatten time: 14676 ns (0.000 s) 0.0 %
Total Sigmoid time: 164517 ns (0.000 s) 0.0 %
Total time: 28200309439 ns (28.200 s) 100.0 %
```

```
sudo ./cnnSolver cat.9495.jpg.rgba.planar
Allocating DMA memory...
DMA memory allocated.
InputImageFxp: Virtual address: 0xB6A16000 (3064029184)
Buffer0: Virtual address: 0xB5A55000 (3047510016)
Buffer1: Virtual address: 0xB5664000 (3043377152)
OUTPUT: 0.06528854 --> CAT
Conv 0 --> 15588497897 ns (15.588 s)
Conv 1 --> 74655428443 ns (74.655 s)
Conv 2 --> 68282866160 ns (68.283 s)
Conv 3 --> 59147606758 ns (59.148 s)
Conv 4 --> 5459415466 ns (5.459 s)
MaxPool 0 --> 265860590 ns (0.266 s)
MaxPool 1 --> 125571553 ns (0.126 s)
MaxPool 2 --> 59560655 ns (0.060 s)
MaxPool 3 --> 25592357 ns (0.026 s)
MaxPool 4 --> 1187089 ns (0.001 s)
Dense 5 --> 270244645 ns (0.270 s)
Dense 6 --> 115104 ns (0.000 s)
Total Conv time: 223133814724 ns (223.134 s) 99.7 %
Total MaxPool time: 477772244 ns (0.478 s) 0.2 %
Total Dense time: 270359749 ns (0.270 s) 0.1 %
Total Flatten time: 433646 ns (0.000 s) 0.0 %
Total Sigmoid time: 128573 ns (0.000 s) 0.0 %
Total time: 223882508936 ns (223.883 s) 100.0 %
```

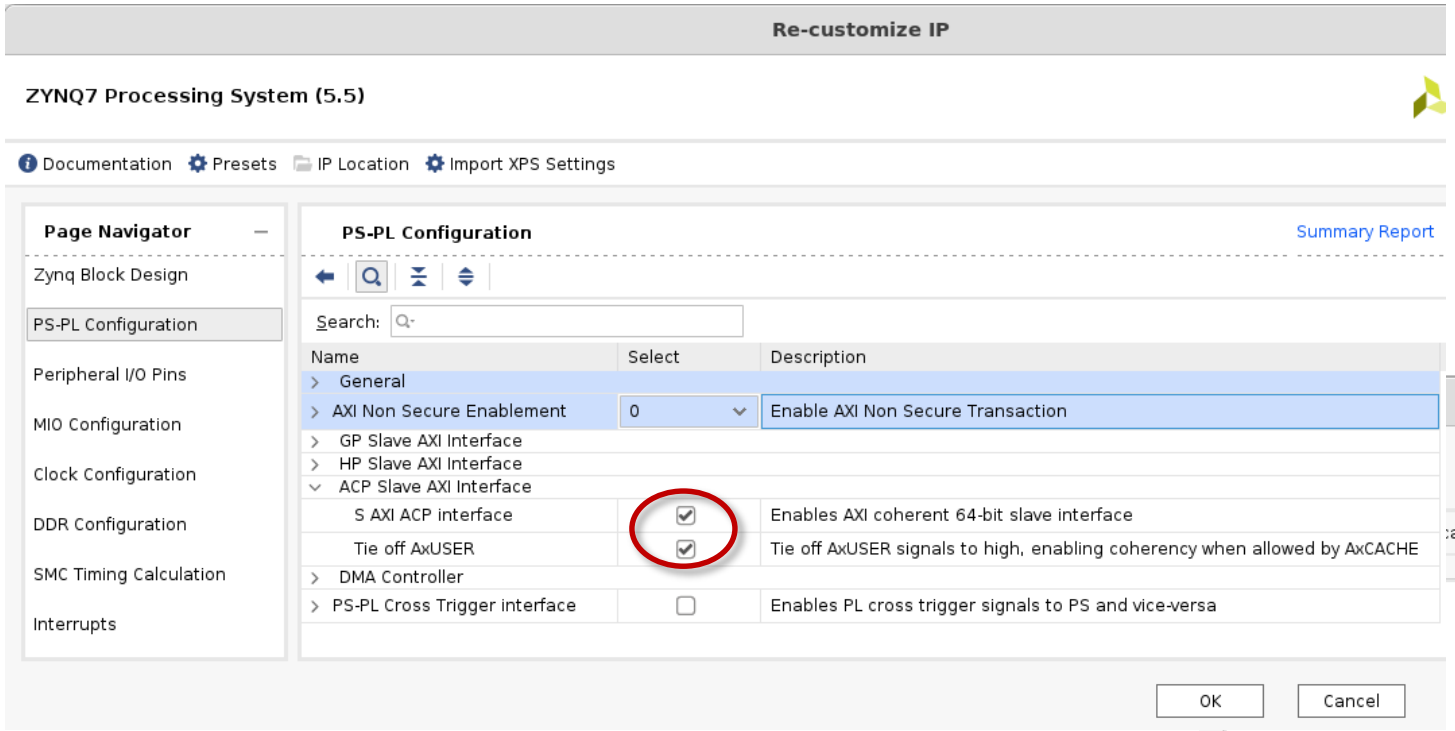
SW execution with non-cacheable RAM: From ~ 28 s to ~ 224 s

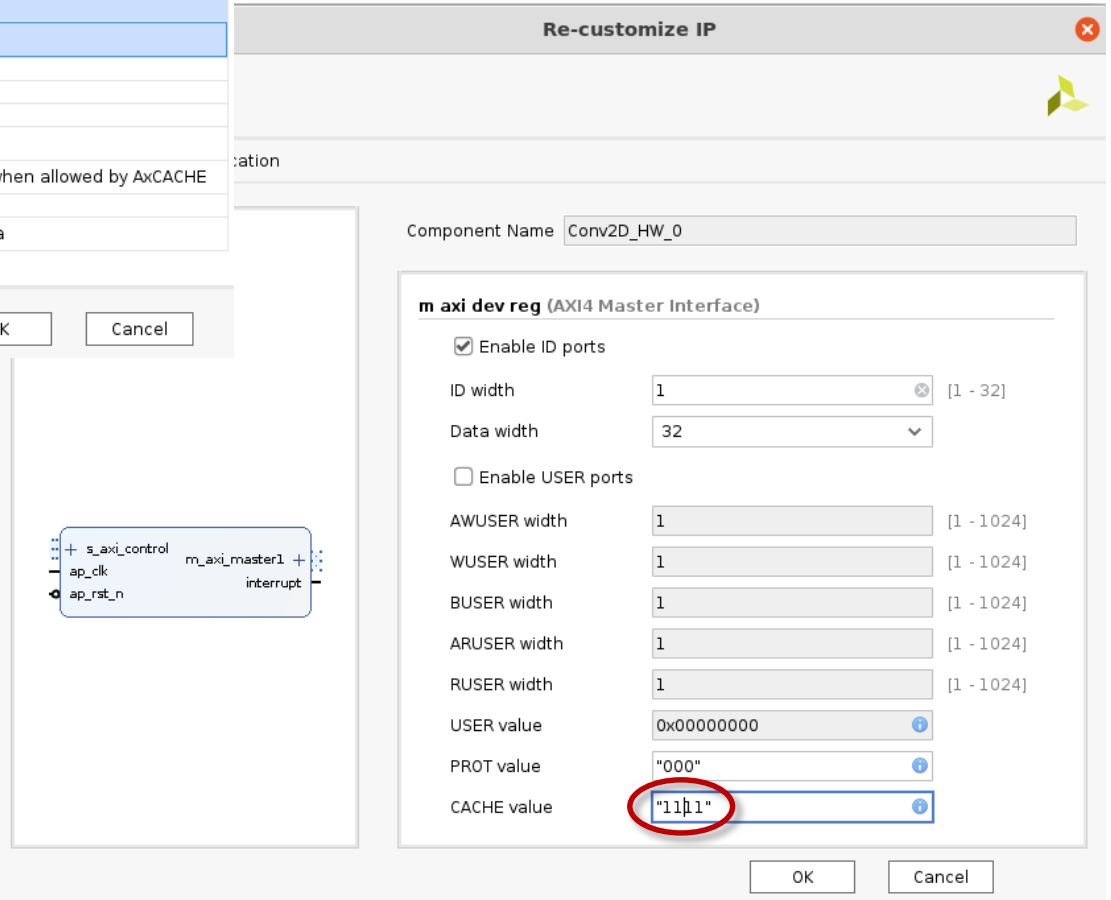
Using the ACP port

- Accelerator Coherent Port (ACP)
  - Allows a peripheral to talk directly to the snoop controller of the processors
- The peripheral accesses directly the cache hierarchy of the processors
  - If the required data is in the caches, the peripheral will access it there
  - If the peripheral writes to data, either it stays in the cache or the cache line is invalidated and the processors know they have to go outside to fetch it
- The process is transparent for the SW application
- The peripherals can enjoy the speed of the cache
- However, high risk of “thrashing” the cache
  - If the peripheral accesses large amounts of data, it will evict all data in the cache hierarchy
  - The processors will have to access main memory for their own (unrelated) data!



Configuring ZYNQ7 and peripherals for ACP caching





To use the ACP port with cacheable transactions:

- In the ZYNQ7 processor system configuration page:
  - Activate the ACP Slave AXI interface
  - Enable the tying off of the AxUSER signals to enable coherence when allowed by the peripherals
- In our accelerator:
  - Change the CACHE value (AxCACHE lines) to “1111” (write-back, read and write-allocate)

Explicitly managing cache

- Applications can explicitly issue **flush** and **invalidate** operations for specific cache lines (based on physical address ranges)
  - **Flush operation** forces the cache hierarchy to **write back a cache line to main memory**
  - **Invalidate operation** marks a **cache line as not valid** (if present)
    - The processors will have to **re-fetch** it from main memory the next time

Note: It is potentially dangerous to invalidate a cache line without flushing it, but flushing may not be safe if the line data is partially in DRAM

- This problem appears if a communication buffer with a peripheral shares a cache line with other variables
- To avoid this situation, **align your allocations to the cache line** (e.g., 32 B or 64 B)
  - In our case, this is fine since DMA memory is allocated with page granularity (>> line cache size)

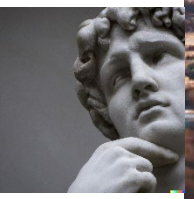
- Drawback:
  - The flush/invalidate operations work on cache lines
  - For large buffers, a potentially high number of operations may be necessary!

Flush/invalidate in the Zynq-7000

- Before passing a data object to an accelerator
  - `cma_flush_cache(input, (uint32_t)phyInput, LENGTH_IN_BYTES);`
  - `cma_invalidate_cache(output, (uint32_t)phyOutput, LENGTH_IN_BYTES);`

- After the peripheral ends, before accessing the data object
  - `cma_invalidate_cache(output, (uint32_t)phyOutput, LENGTH_IN_BYTES);`

Note: We use flush(input)+invalidate(output) to pass data to the accelerator because the invalidate operation is itself implemented as flush+invalidate in our platform. If we didn't invalidate (+flush) the output lines before calling the accelerator, if those cache lines had modified data before, they would overwrite the results stored in the buffer before the processor had the chance to read them again. This situation is possible in our case since we are using both the input and output buffers to compute the bias/ReLU/maxpool operations.



Questions?

Prof. David Atienza

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