EE382M.20: System-on-Chip (SoC) Design

Lecture 8 – HW/SW & Accelerator Interfacing

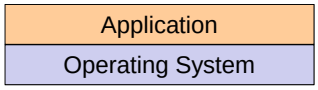
Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 8: Outline

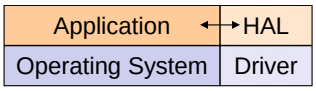
- **Software interfacing**
  - Application mapping
  - Hardware abstraction layer (HAL)
  - Drivers
- **Hardware interfacing**
  - Accelerator & SoC architecture
  - Accelerator coupling
  - Zynq architecture

Application Mapping

- **Identify modules to accelerate**
  - Profiling, refactoring
  - Communication, parallelism



- **Map modules to accelerators**
  - Develop hardware abstraction layer (HAL)
  - Develop drivers
  - Define HW/SW interfaces
    - Address maps
    - Interrupts
    - …



Hardware Abstraction Layer (HAL)

- **Defines/provides an efficient interface to hardware while maintaining application code structure**
  - From application, HW accelerator looks like a function call
  - From HW accelerator, application looks like HW/buffer
- **Encapsulate HW/SW interaction, isolating hardware detail from application software**
  - Synchronization, flow control, status queries
  - Data transfer and communication
- **Enables mixture of hardware and software models**
  - Selective use of hardware modules
  - Debug and emulation
- **Support verification**
  - Instrumentation to capture/replay HW stimuli/responses

Source: Steven Smith

HAL Example

- **Application layer**
  - Maps application function to lower driver layer

```
void appFunction1(int data1, int *data2, int data2Size)
{
#if HAL_ENABLED

#if HAL_INSTRUMENT
    fprintf(pfVStim,"halFunction1InputData1 = %d ;\n", data1) ;
#endif

    Drv_checkReadyFunction1(TRUE);

    Drv_enqueueToFunction1Data1(data1);
    Drv_enqueueToFunction1Data2(data2, data2Size);

    Drv_startFunction1();
    Drv_waitFunction1();
#else

    //  … Existing HLL application function code …

#endif

}
```

Source: Steven Smith

EE382M.20: SoC Design, Lecture 8 © 2018 A. Gerstlauer 5

Low-Level Driver

- **Synchronization**
  - Hardware triggers
    - Write to special hardware address
  - Polling or interrupts to wait for hardware
    - Read hardware flag
    - Interrupt service routine
- **Data transfers**
  - Send input data to hardware and transfer results back
    - Write to/from hardware-accessible registers and/or memory
    - Data packing & formatting (HW vs. SW data structures, e.g. pointers)
    - Map between virtual and physical memory spaces
- ➢ **Kernel-level code for direct hardware/interrupt access**
  - ➢ Linux kernel module
    - ➢ Interrupt service routines (ISRs) and root privileges

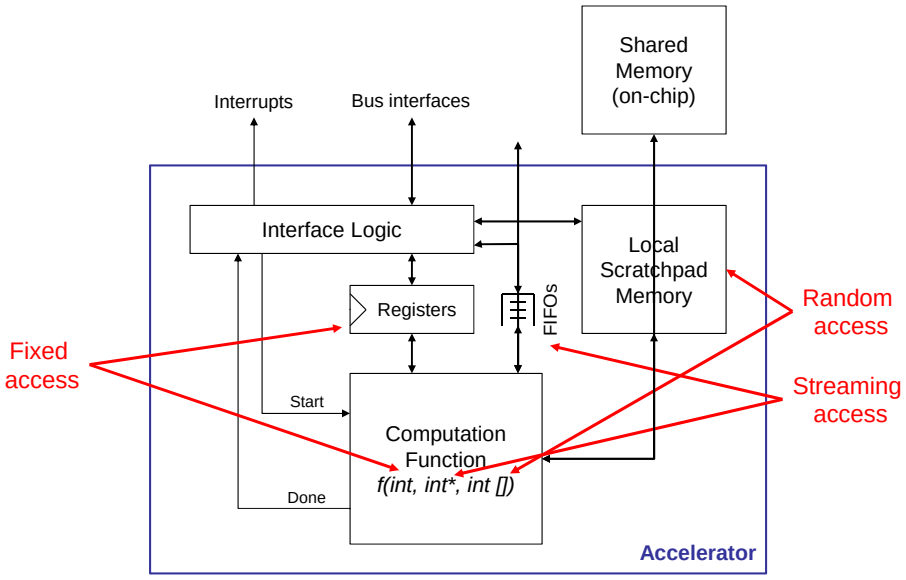
EE382M.20: SoC Design, Lecture 8 © 2018 A. Gerstlauer 6

Driver Example

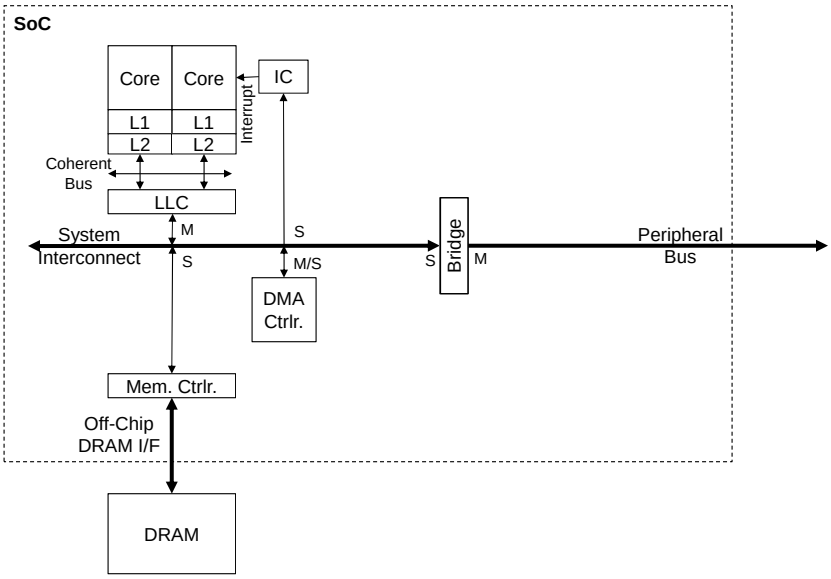
```
volatile void *base;

void Drv_init(void)
{
    base = mmap(…);  // map HW physical into virtual address
}

int Drv_waitReadyFunction1(int waitTillReady)
{
    int result;
    do {
        result = *((int*) base);
    } while (!result && waitTillReady) ;
    return result ;
}

int Drv_enqueueToFunction1Data2(int *data2, int data2Size)
{
    int i;

    for (i=0; i < data2Size ; i++)
    {
       // stream data into accelerator local memory
       *(((int*) base)+4) = data2[i];
    }
}
```

Source: Steven Smith

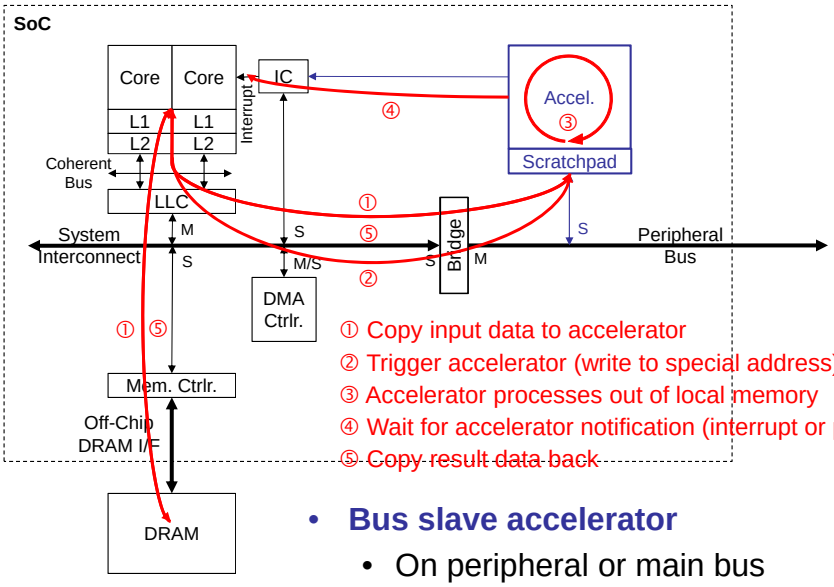
Accelerator Architecture & Interfacing



SoC Architecture



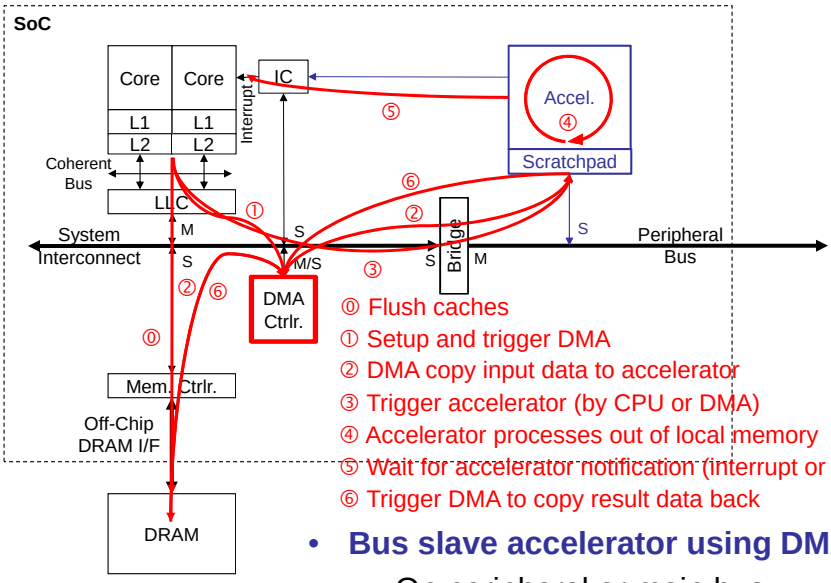
Accelerator Coupling (1)



① Copy input data to accelerator
② Trigger accelerator (write to special address)
③ Accelerator processes out of local memory
④ Wait for accelerator notification (interrupt or polling)
⑤ Copy result data back

- **Bus slave accelerator**
  - On peripheral or main bus

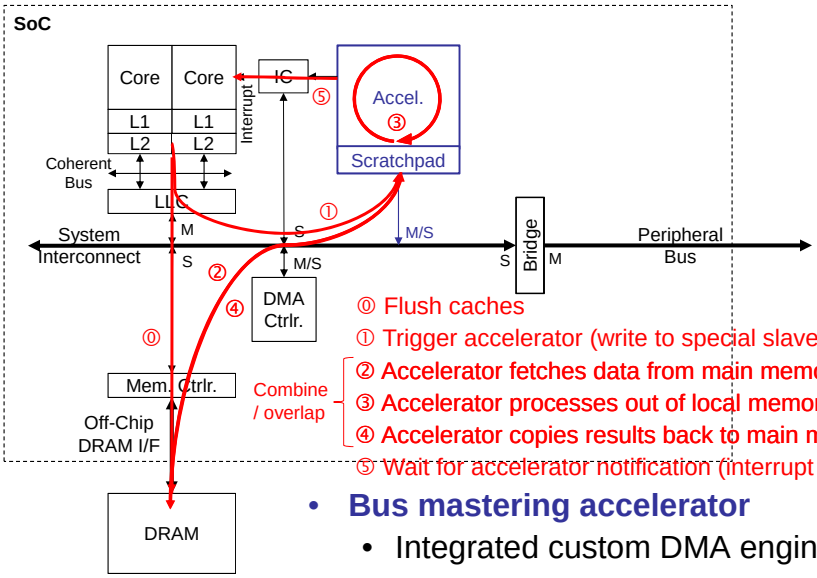
Accelerator Coupling (2)



⓪ Flush caches
① Setup and trigger DMA
② DMA copy input data to accelerator
③ Trigger accelerator (by CPU or DMA)
④ Accelerator processes out of local memory
⑤ Wait for accelerator notification (interrupt or polling)
⑥ Trigger DMA to copy result data back

- **Bus slave accelerator using DMA**
  - On peripheral or main bus

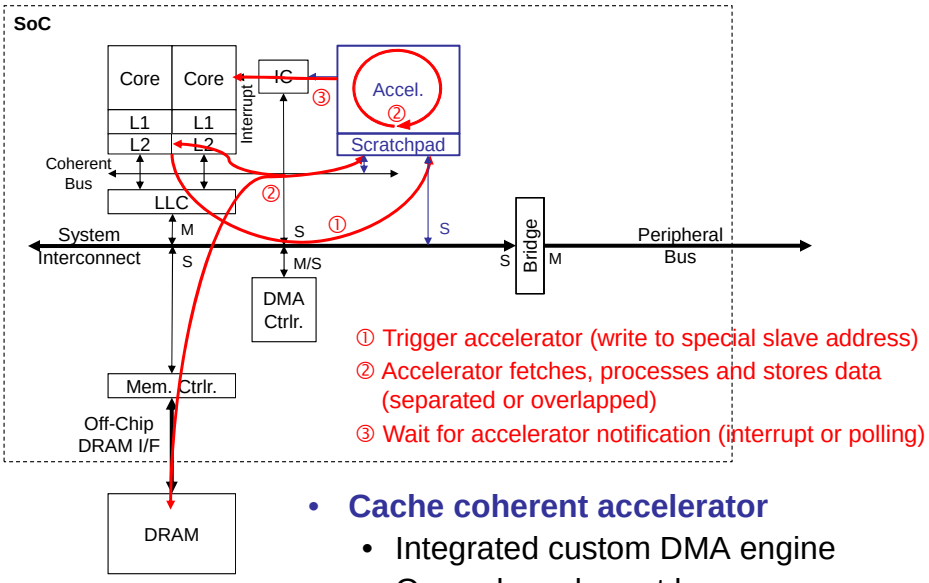
Accelerator Coupling (3)



⓪ Flush caches
① Trigger accelerator (write to special slave address)
Combine / overlap
② Accelerator fetches data from main memory
③ Accelerator processes out of local memory
④ Accelerator copies results back to main memory
⑤ Wait for accelerator notification (interrupt or polling)

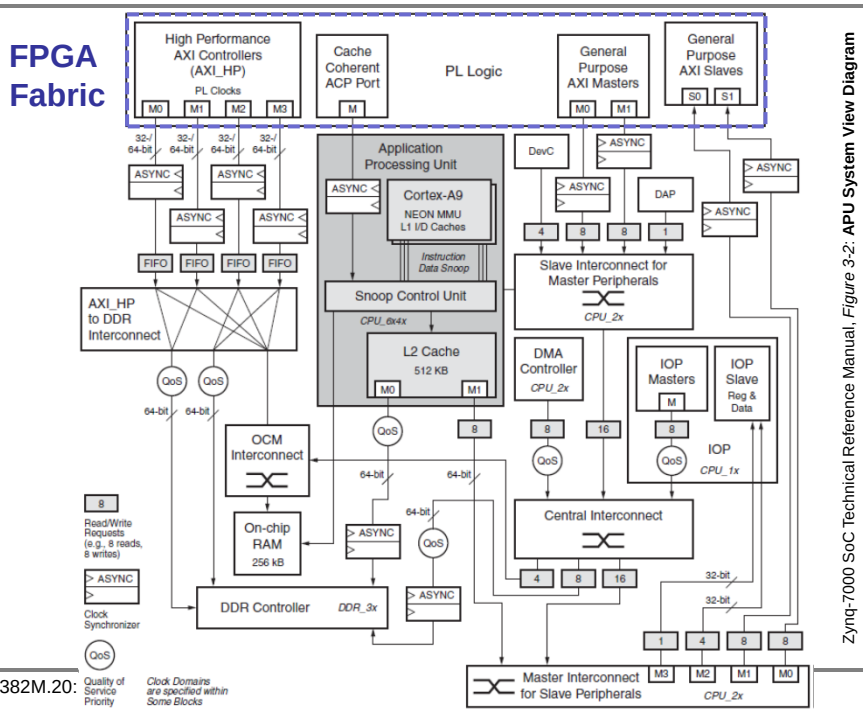
- **Bus mastering accelerator**
  - Integrated custom DMA engine
  - On main bus

Accelerator Coupling (4)



- **Cache coherent accelerator**
  - Integrated custom DMA engine
  - On cache coherent bus

Zynq-7000 Architecture



Zynq-7000 Connectivity

⓪ Flush caches

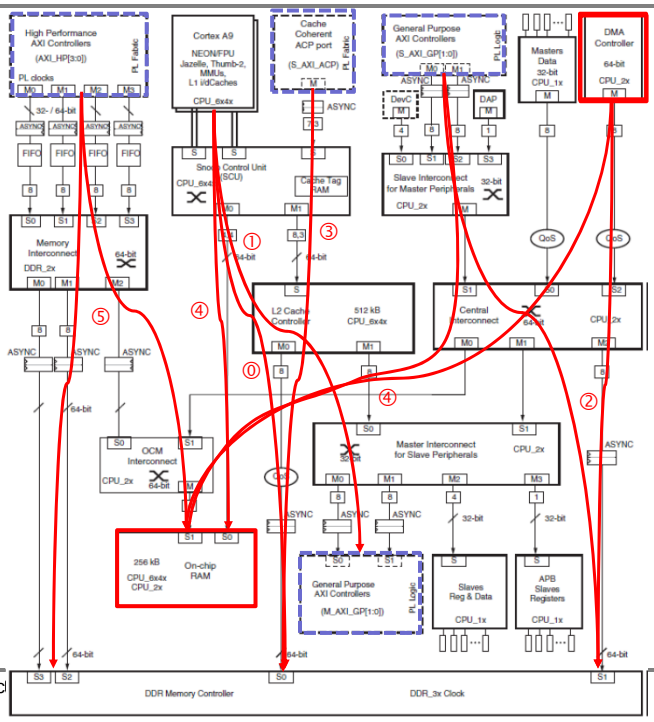
① FPGA slaves

② DMA (via DMA Controller or FPGA masters)

③ Cache-coherent access port (ACP)

④ On-Chip RAM (OCR, un-cached, e.g. accelerator scratchpad)

⑤ High-performance (HP) FPGA master ports (FIFOs)



Zynq-7000 SoC Technical Reference Manual, *Figure 5-1:* **Interconnect Block Diagram**

Zynq-7000 Address 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