ECE382M.20: System-on-Chip (SoC) Design

Lecture 8 – HW/SW & Accelerator Interfacing

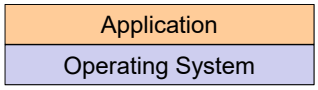
Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 8: Outline

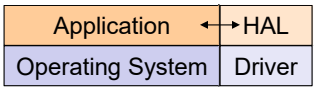
- **Software interfacing**
  - Application mapping
  - Hardware abstraction layer (HAL)
  - Drivers
- **Hardware interfacing**
  - Accelerator & SoC architecture
  - Accelerator coupling
  - Zynq architecture

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

Low-Level Driver

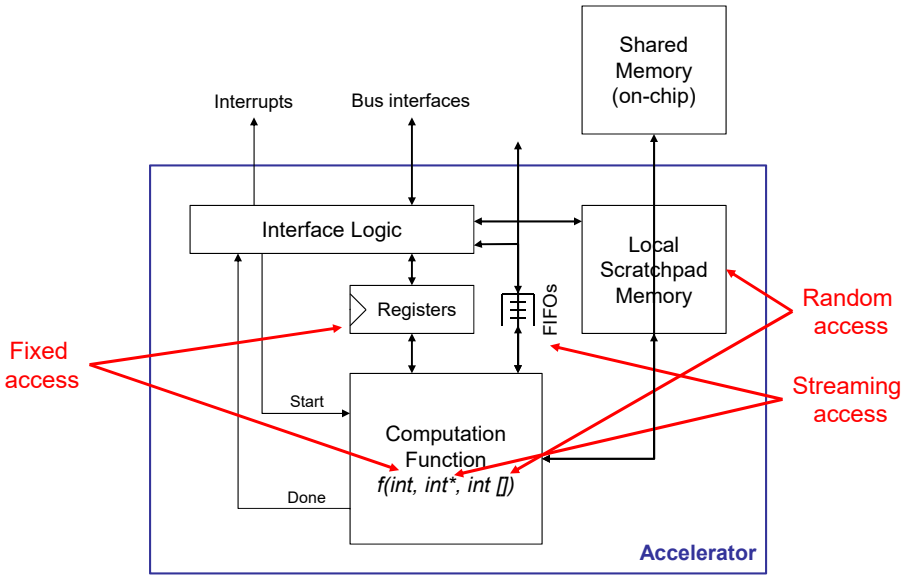
- **Synchronization**
  - Hardware triggers
    - Write to special hardware address
  - Polling or interrupts to wait for hardware
    - Read hardware flag
    - Interrupt service routine
- **Data transfers**
  - Send input data to hardware and transfer results back
    - Write to/from hardware-accessible 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

Driver Example

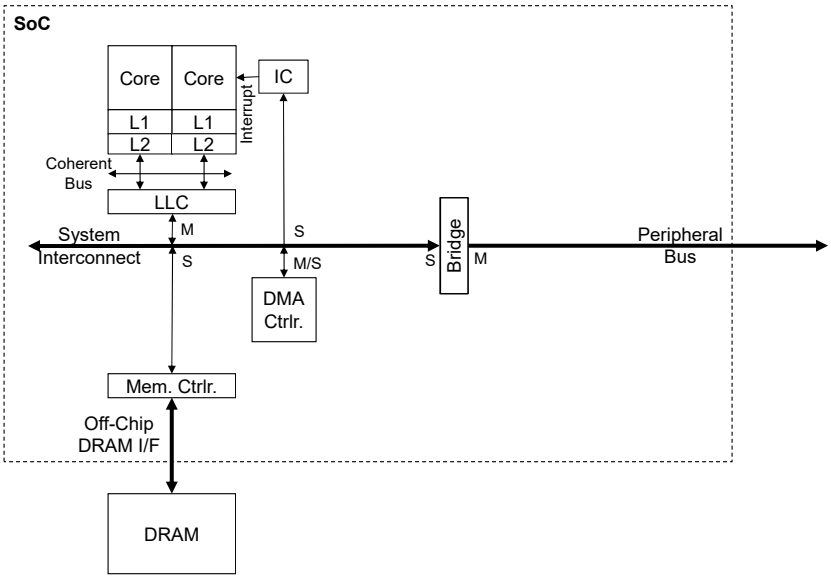
```
volatile void *base;

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

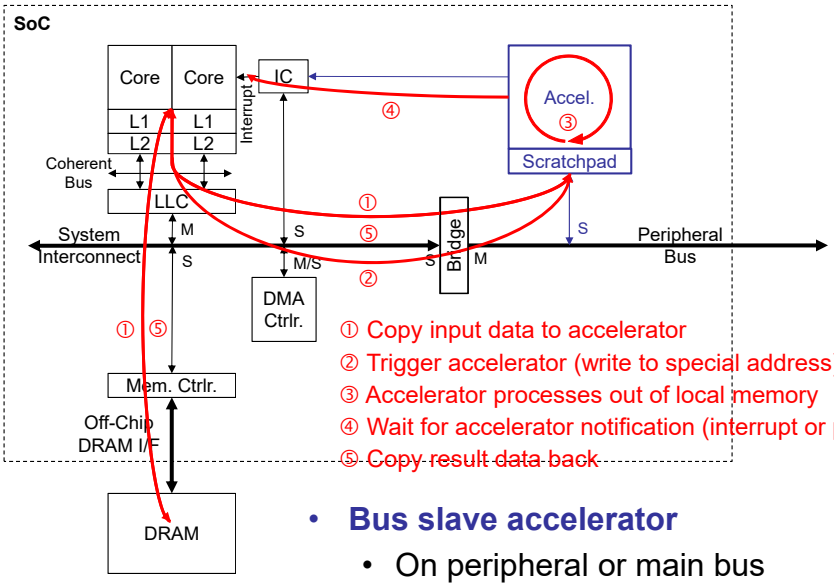
Accelerator Architecture & Interfacing



SoC Architecture



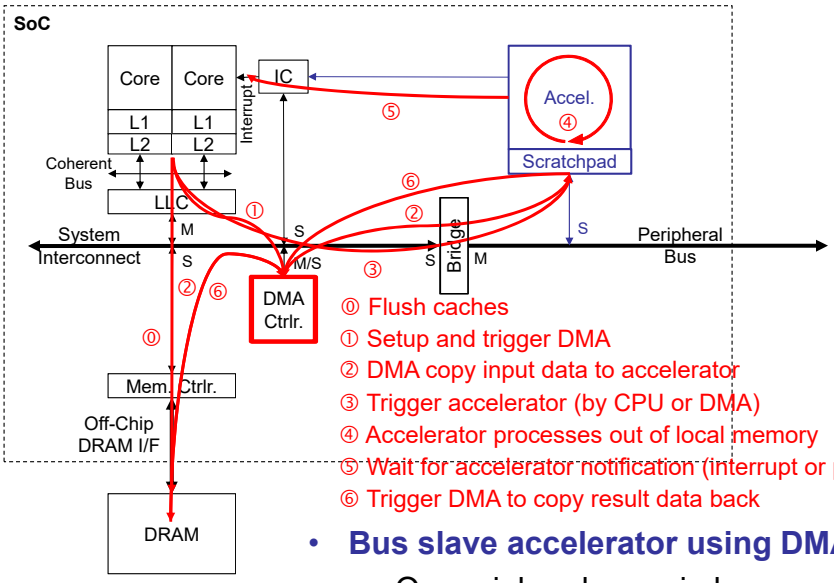
Accelerator Coupling (1)



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

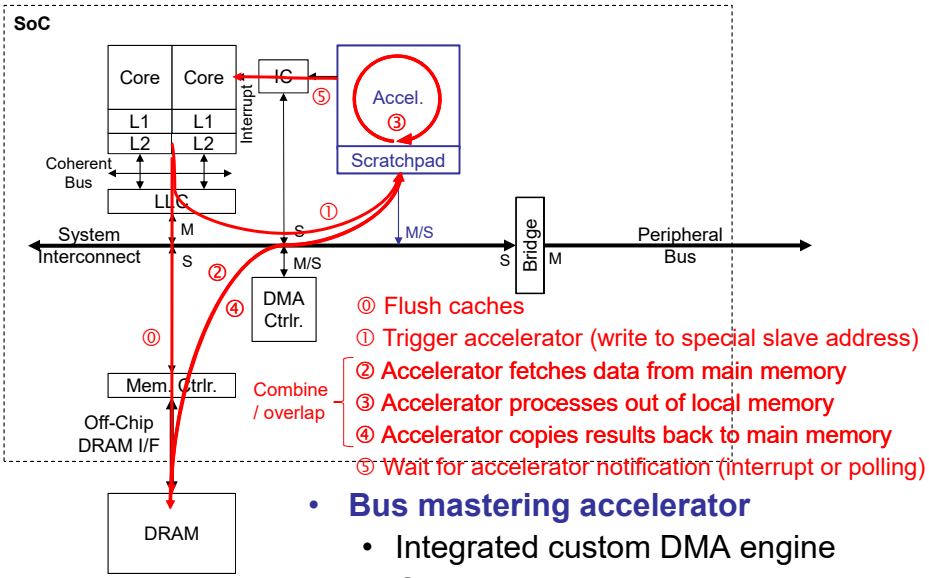
Accelerator Coupling (2)



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

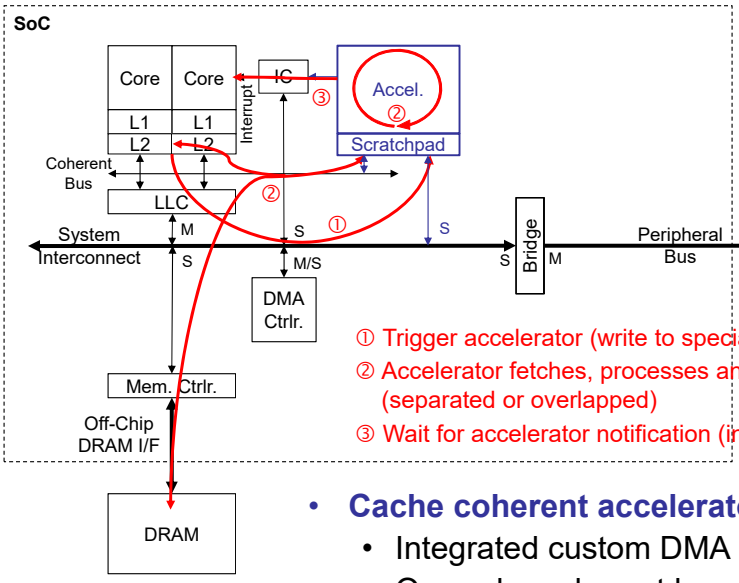
Accelerator Coupling (3)



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)

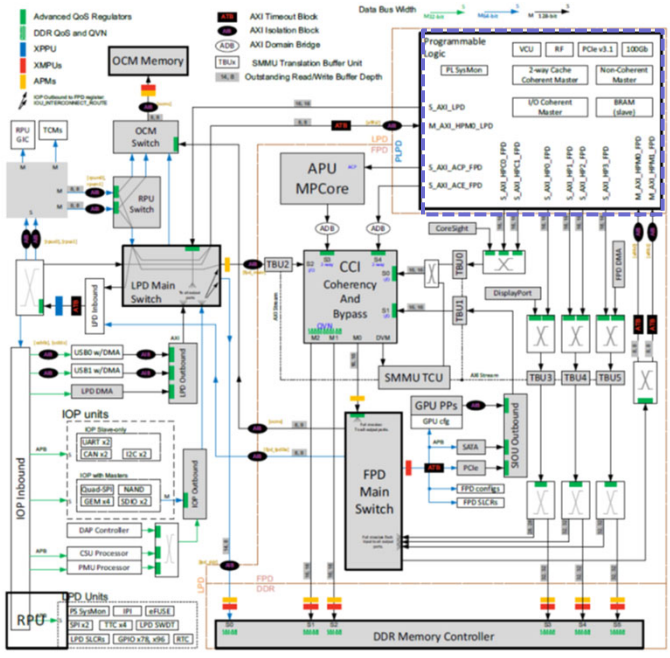
- **Bus mastering accelerator**
  - Integrated custom DMA engine
  - On main bus

Accelerator Coupling (4)



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

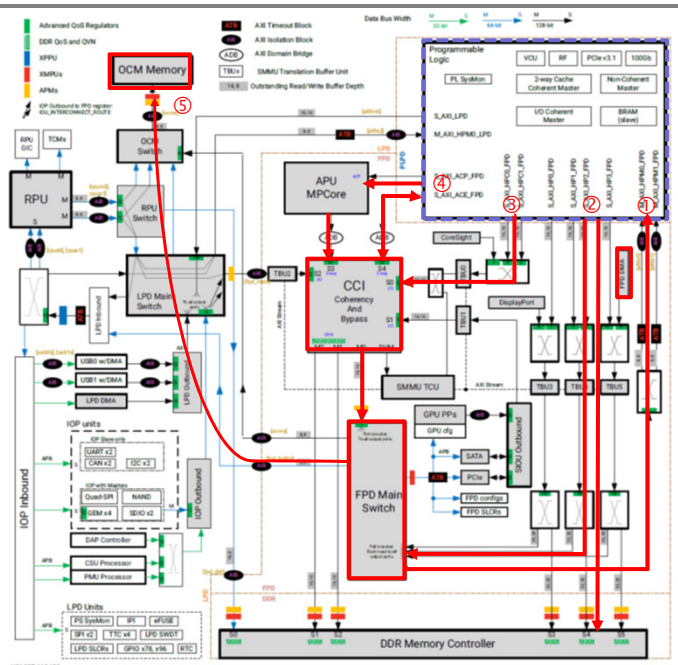
Zynq UltraScale+ Architecture



Zynq UltraScale+ MPSoC Technical Reference Manual, *Figure 1-1:* **AXI Interconnect**

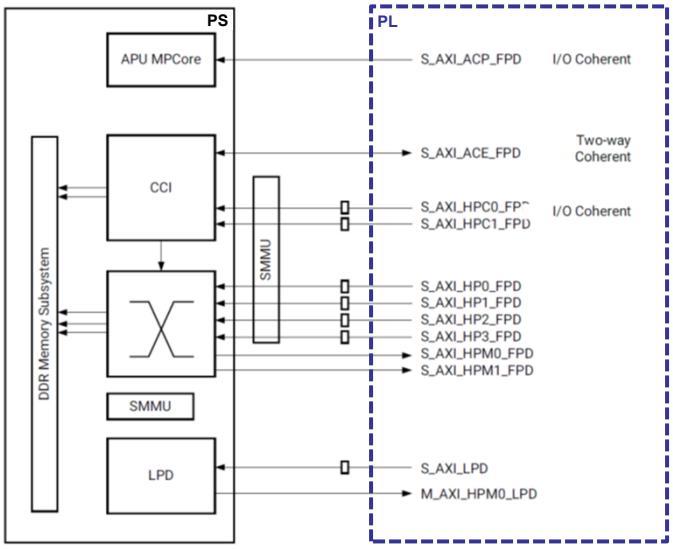
Zynq UltraScale+ Connectivity

① FPGA slaves
② Bus masters (via DMA Controller or FPGA masters)
③ I/O (one-way) coherent FPGA masters
④ Full (two-way) cache-coherent access port (ACP)
⑤ On-Chip RAM (OCM)



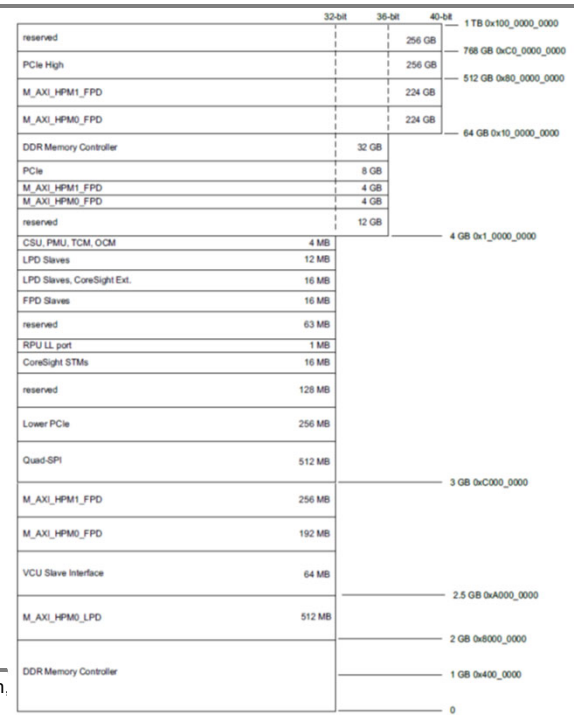
Zynq UltraScale+ MPSoC Technical Reference Manual, *Figure 15-1:* **PS Interconnect**

Zynq UltraScale+ PS-PL Interfaces



Zynq UltraScale+ Technical Reference Manual, *Figure 35-2:* **PS-PL AXI Interface Datapaths**

Zynq UltraScale+ Global Address Map



Zynq UltraScale+ Technical Reference Manual, *Figure 10-1:* **Global System Address Map**