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

Lecture 10 – System Integration & Accelerator Interfacing

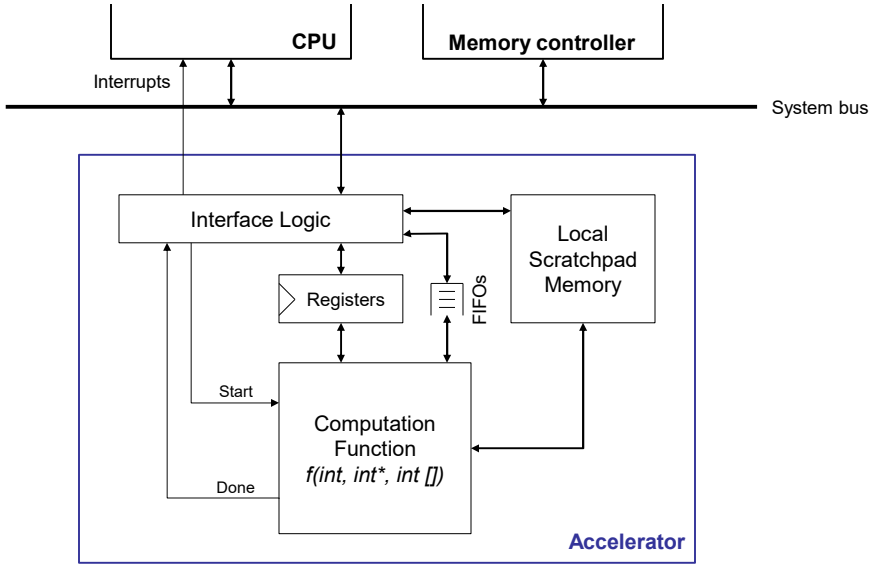
Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu



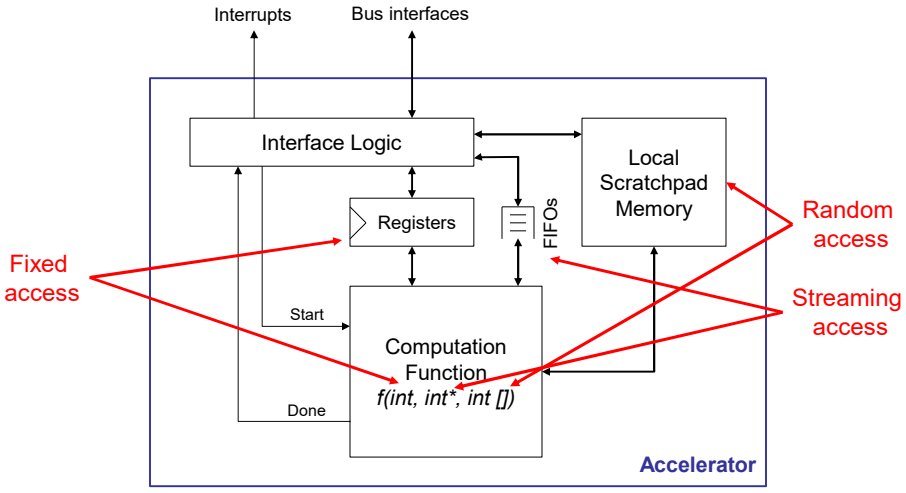
Lecture 10: Outline

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

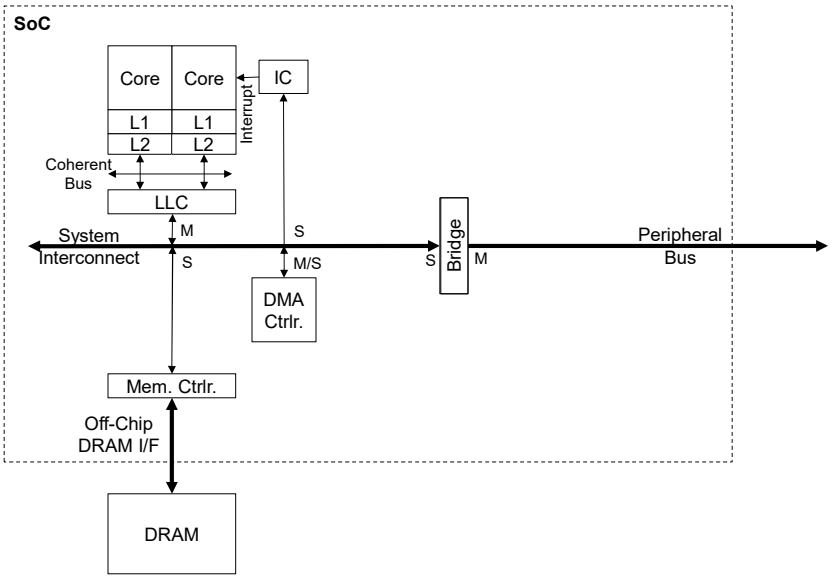
Accelerator Architecture & Interfacing



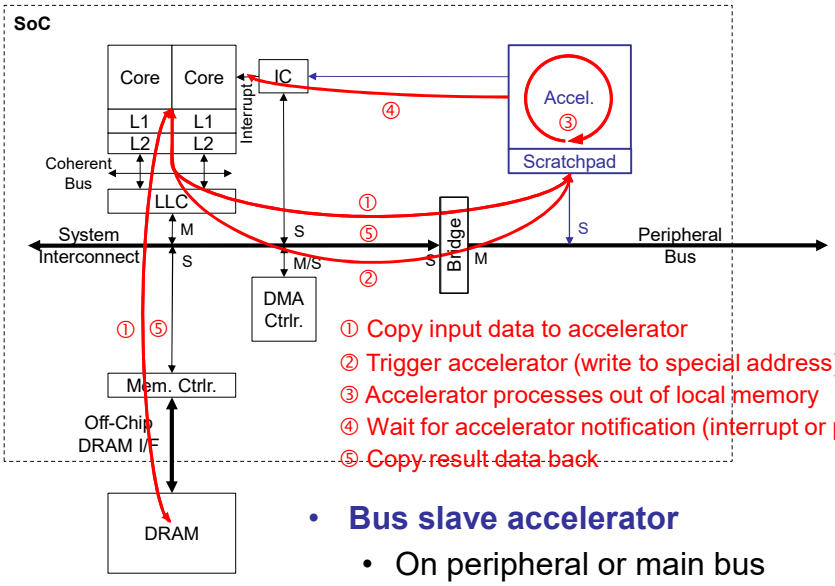
Accelerator Architecture & Interfacing



SoC Architecture



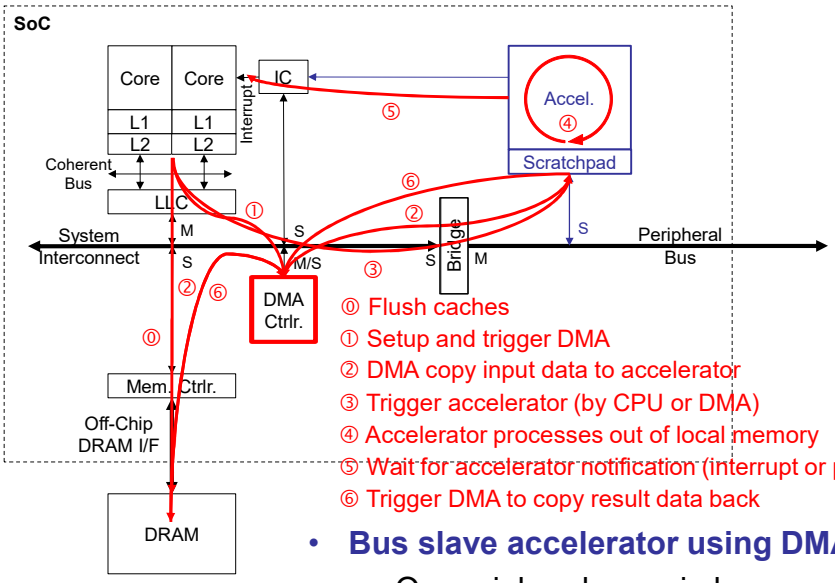
Accelerator Coupling (1)



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

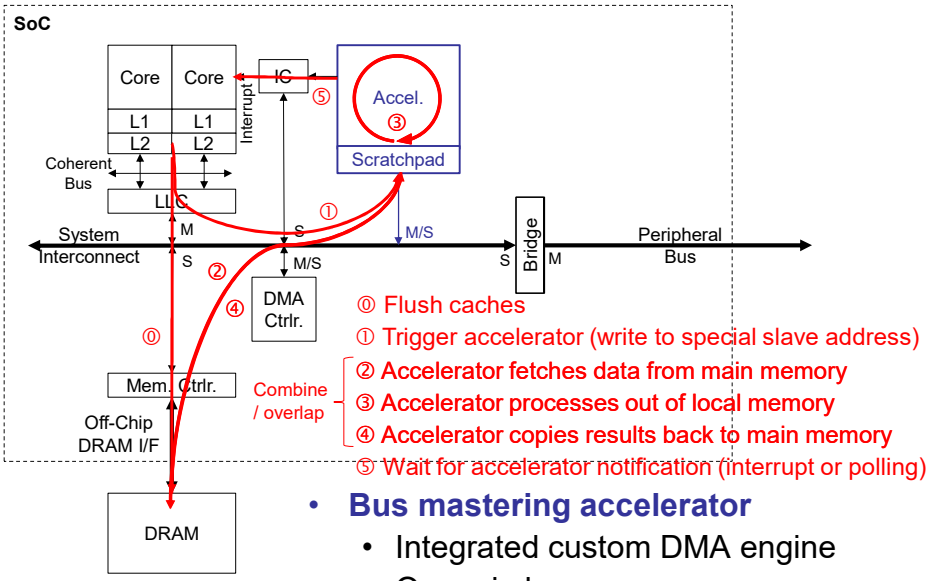
Accelerator Coupling (2)



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

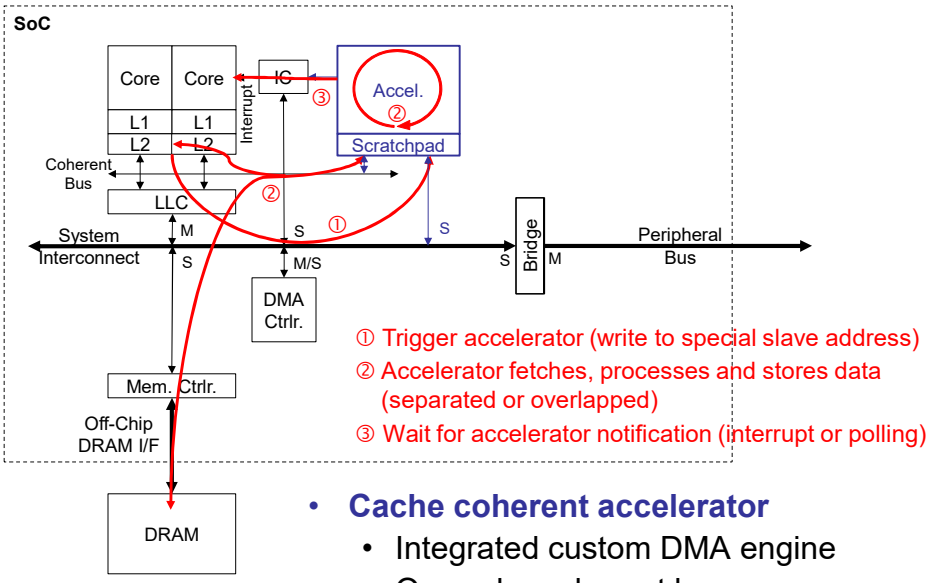
Accelerator Coupling (3)



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)

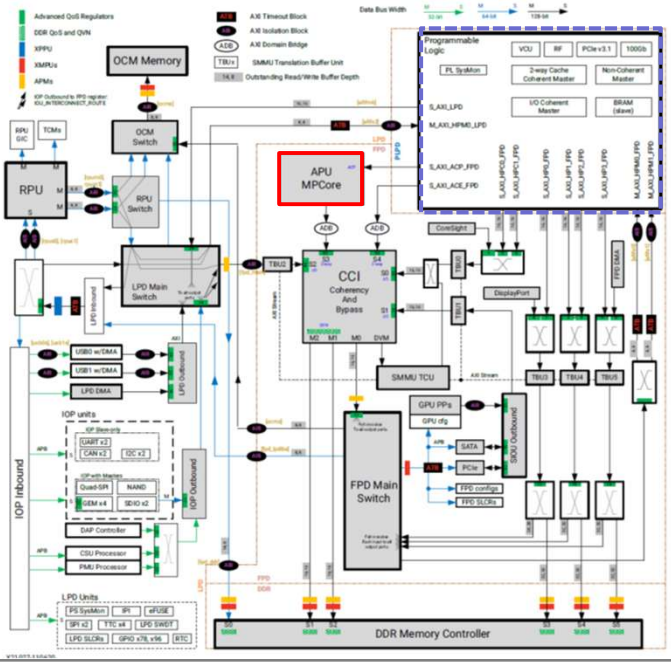
- **Bus mastering accelerator**
  - Integrated custom DMA engine
  - On main bus

Accelerator Coupling (4)



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

Zynq UltraScale+ Architecture



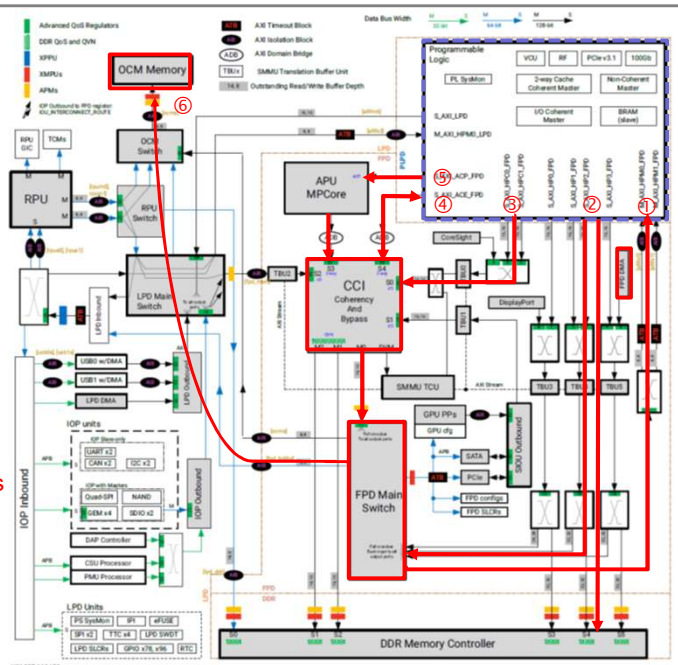
FPGA Fabric

CPU Subsystem

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

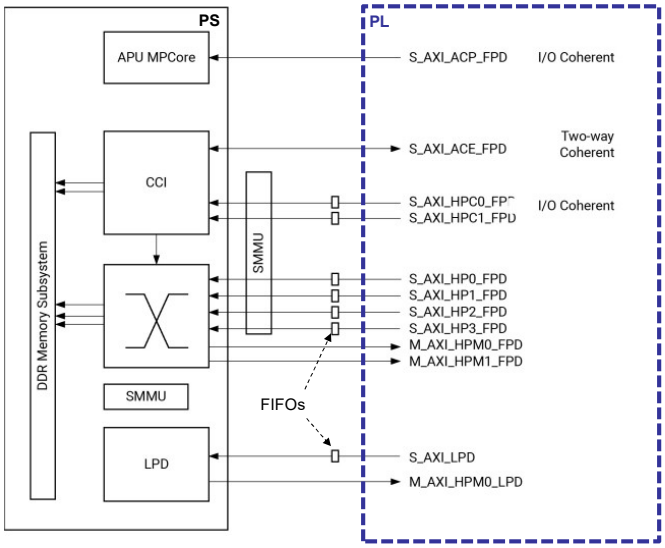
Zynq UltraScale+ Connectivity

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 extensions (ACE)
⑤ I/O (one-way) APU-coherent accelerator access port (ACP)
⑥ On-Chip RAM (OCM)



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

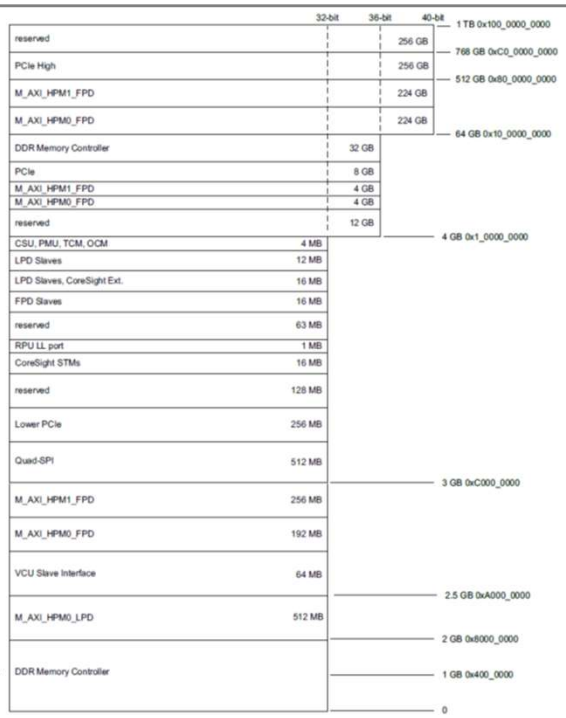
Zynq UltraScale+ PS-PL Interfaces



LPD: Low-Power Domain (peripheral subsystem, can be turned off)

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**

SystemC/QEMU Co-Simulation Interface

libsystemctlm-soc/soc/xilinx/zynqmp/**xilinx-zynqmp.h**:

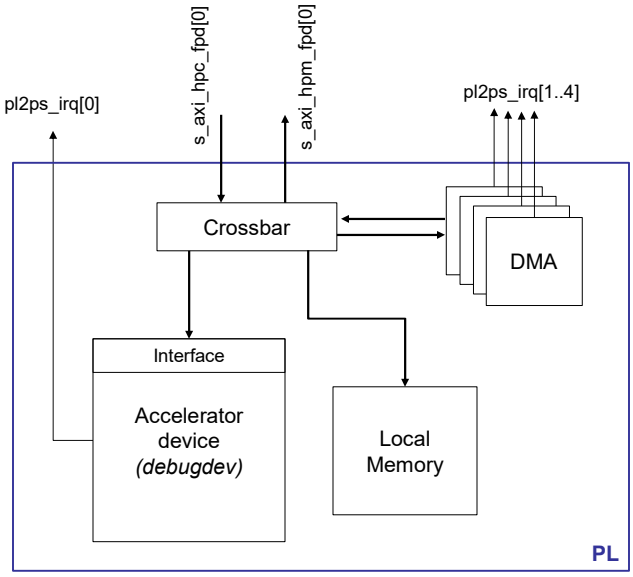
```
class xilinx_zynqmp : public remoteport_tlm { …
public:
/* PS to PL interfaces
 * HPM0 - 1 _FPD.
 * HPM_LPD */
 tlm_utils::simple_initiator_socket<remoteport_tlm_memory_master> *s_axi_hpm_fpd[2];
 tlm_utils::simple_initiator_socket<remoteport_tlm_memory_master> *s_axi_hpm_lpd;
 tlm_utils::simple_initiator_socket<remoteport_tlm_memory_master> *s_lpd_reserved;

/* PL to PS interfaces
 * HPC0 - 1.
 * HP0 - 3.
 * PL_LPD
 * ACP */
 tlm_utils::simple_target_socket_tagged<xilinx_zynqmp> *s_axi_hpc_fpd[2];
 tlm_utils::simple_target_socket_tagged<xilinx_zynqmp> *s_axi_hp_fpd[4];
 tlm_utils::simple_target_socket_tagged<xilinx_zynqmp> *s_axi_lpd;
 tlm_utils::simple_target_socket_tagged<xilinx_zynqmp> *s_axi_acp_fpd;
 tlm_utils::simple_target_socket_tagged<xilinx_zynqmp> *s_axi_ace_fpd;

/* Interrupts */
 sc_vector<sc_signal<bool> > pl2ps_irq;
 sc_vector<sc_signal<bool> > ps2pl_irq;

…
```

SystemC/QEMU Co-Simulation Demo



SystemC/QEMU Co-Simulation Demo

systemctlm-cosim-demo/zynqmp_demo.cc:

```
SC_MODULE(Top){
  iconnect<NR_MASTERS, NR_DEVICES> bus;
  xilinx_zynqmp zynq;
  memory mem;
  debugdev debug;
  demodma *dma[NR_DEMODMA];
  sc_signal<bool> rst, rst_n;

  Top(sc_module_name name, const char *sk_descr, sc_time quantum) : … { …
    zynq.rst(rst);                                                    Reset

    bus.memmap(0xa0000000ULL, 0x100 - 1, ADDRMODE_RELATIVE, -1, debug.socket);
    for (i = 0; i < (sizeof dma / sizeof dma[0]); i++)                Crossbar slaves
      bus.memmap(0xa0010000ULL+0x100*i, 0x18-1, …, dma[i]->tgt_socket);
    bus.memmap(0xa0800000ULL, 64 * 1024 - 1, ADDRMODE_RELATIVE, -1, mem.socket);

    bus.memmap(0x0LL, 0xffffffff - 1, ADDRMODE_RELATIVE, -1, *(zynq.s_axi_hpc_fpd[0]));
    zynq.s_axi_hpm_fpd[0]->bind(*(bus.t_sk[0]));                      PS<->PL connections

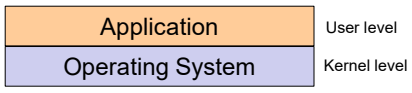
    for (i = 0; i < (sizeof dma / sizeof dma[0]); i++) {              Crossbar masters
      dma[i]->init_socket.bind(*(bus.t_sk[1 + i]));
      dma[i]->irq(zynq.pl2ps_irq[1 + i]);
    }                                                                 Interrupts
    debug.irq[0](zynq.pl2ps_irq[0]);
  …
```

Lecture 10: Outline

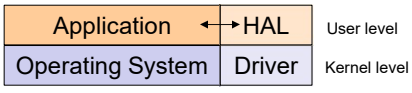
- ✓ **Hardware interfacing**
  - ✓ Accelerator & SoC architecture
  - ✓ Accelerator coupling
  - ✓ Zynq architecture
- **Software interfacing**
  - Application mapping
  - Hardware abstraction layer (HAL)
  - Drivers
- **System optimization**
  - Data movement

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
    - Repeatedly poll hardware flag
    - Wait for asynchronous hardware interrupt
- **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**
  - Interrupt service routines (ISRs)
  - Kernel-level I/O

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

Linux Drivers

- **Built-in drivers and kernel mechanisms**
  - Memory-mapped I/O
  - Access to physical memory addresses via `/dev/mem`
    - Use *mmap*() to map physical into virtual address space
- **Custom driver as loadable kernel module**
  - Interrupt service routines (ISRs)
  - Root privileges for special (e.g. non memory-mapped) I/O

Anatomy of a Linux Kernel Driver

- **Device tree**
  - Read by the kernel on boot
  - Allows adding devices into pre-compiled kernel
  - Each device supported by a driver
    - Compiled-in or loaded dynamically as kernel module
  - Device tree tells kernel what drivers to load
    - If a driver is not available, device gets ignored
- **Kernel module**
  - Dynamically loadable kernel code (driver)
  - Manually (`insmod`) or automatically (`modprobe`) loaded
  - Standard interface for registering with kernel

Device Tree

- **Device tree entry (`fpga0` device)**

```
amba_pl: amba_pl@0 {

    #address-cells = <0x2>;
    #size-cells = <0x2>;
    compatible = "simple-bus";
    ranges;

    fpga0: fpga0@a0000000 {
        compatible = "ece382m,fpga";
        reg = < 0x0 0xa0000000 0x0 0x00000100
                0x0 0xa0800000 0x0 0x00010000 >;
        interrupt-parent = < &gic >;
        interrupts = < 0x0 89 0x4 >;
    };
};
```

64-bit (2x32) addresses

Kernel driver match

2 address ranges

1 interrupt (IRQ 89+32=121)

Kernel Module

- **Module initialization and registration**

```
static struct of_device_id fpga_drv_of_match[] = {
  { .compatible = "ece382m,fpga", }, { },
};
MODULE_DEVICE_TABLE(of, fpga_drv_of_match);

static struct platform_driver fpga_driver = {
  .driver = {
    .name = "fpga",
    .owner = THIS_MODULE,
    .of_match_table = fpga_drv_of_match,
  },
  .probe  = fpga_drv_probe,
  .remove = fpga_drv_remove,
};

static int __init fpga_init_module(void) {
#ifdef DEBUG
   printk("FPGA Interface Module\n");
   printk(KERN_INFO "\nfpga_drv: FPGA Driver Loading.\n");
#endif

   return platform_driver_register(&fpga_driver);
}

module_init(fpga_init_module);
```

Device tree match

Driver data structure

Load module

Register with kernel
-> Triggers probe if device tree/module match

Register module

Kernel Module

- **Device probing, part 1 (install `/dev/fpga` device)**

Device operations

```
struct file_operations fpga_fops = {
   .owner=   THIS_MODULE,
   .read    = fpga_read1,
   .write   = fpga_write1,
   .unlocked_ioctl = fpga_ioctl1,
   .open    = fpga_open1,
   .release = fpga_release1,
   .fasync  = fpga_fasync1,
};

static struct miscdevice fpga_miscdev = {
   .minor =        MISC_DYNAMIC_MINOR,
   .name =         "fpga",
   .fops =         &fpga_fops,
};

static int fpga_drv_probe (struct platform_device *pdev) {
   int rv;
   struct resource *r_reg; /* IO register resources */
   struct resource *r_mem; /* IO memory resources */

   if (misc_register(&fpga_miscdev)) {
      dev_err(dev, "fpga_drv: unable to register device. ABORTING!\n");
      return -EBUSY;
   }
   …
```

Misc device
(major 10, minor set by kernel)

Register device

Kernel Module

- **Device probing, part 2 (register resources)**

```
…
l._reg_ptr = devm_platform_get_and_ioremap_resource(pdev, 0, &r_reg);
if (IS_ERR(l.reg_ptr)) {
  dev_err(dev, "fpga_drv: Unable to map FPGA registers.\n");
  return PTR_ERR(l.reg_ptr);
}
l.mem_ptr = devm_platform_get_and_ioremap_resource(pdev, 1, &r_mem);
if (IS_ERR(l.mem_ptr)) {
  dev_err(dev, "fpga_drv: Unable to map FPGA memory.\n");
  return PTR_ERR(l.mem_ptr);
}

…

l.irq = platform_get_irq(pdev, 0);
if (l.irq < 0) {
  dev_info(dev, "fpga_drv: no IRQ found\n");
  return l.irq;
}
rv = devm_request_irq(dev, l.irq, &fpga_int_handler, 0, "fpga",  NULL);
if (rv) {
   dev_err(dev, "fpga_drv: Can't get interrupt %d: %d\n", l.irq, rv);
   return rv;
}
…
```

Get and map 1st addr. region from device tree

Get and map 2nd addr. region from device tree

Get 1st IRQ no. from device tree

Install handler

(default is level-sensitive)

Kernel Module

- **Device probing, part 3 (install `/proc/fpga` device)**

```
    …
    l.fpga_interrupt_file = proc_create("fpga", 0444, NULL, &proc_ops);
    if(l.fpga_interrupt_file == NULL)
    {
       dev_err(dev, "fpga_drv: create /proc entry returned NULL. ABORTING!\n");
       return -EBUSY;
    }

    dev_info(dev, "fpga_drv: %s %s Initialized\n", fpga_NAME, fpga_VERSION);
    return 0;
}

static const struct proc_ops proc_ops = {
  .proc_open = proc_open_fpga_interrupt,
  .proc_read = seq_read,
  .proc_lseek = seq_lseek,
  .proc_release = single_release,
};

static int proc_read_fpga_interrupt(struct seq_file *f, void *v) {
  seq_printf(f, "Total number of interrupts %19i\n", interruptcount);
  return 0;
}
static int proc_open_fpga_interrupt(struct inode *inode, struct file *file) {
  return single_open(file, proc_read_fpga_interrupt, NULL);
}
```

`/proc` operations

Register special file on open

Kernel Module

- **`ioctl` operations**

```
static long fpga_ioctl1(struct file *file, unsigned int cmd, unsigned long arg){
   int retval = 0;
   unsigned long value;
   unsigned int command_type;
   unsigned int offset;
   volatile unsigned int *access_addr;

#ifdef DEBUG
   printk(KERN_INFO "\nfpga_drv: Inside fpga_ioctl1 \n");
#endif

   offset = ~COMMAND_MASK & cmd & FPGA_MASK;
   if(offset > FPGA_SIZE) retval=-EINVAL;

   command_type = COMMAND_MASK & cmd;
   switch(command_type)
   {
      case 0:  // read
         if(!access_ok((unsigned int *)arg, sizeof(int)))
            return -EFAULT;

         value = readl((volatile unsigned int *)&l.reg_ptr[offset]);
         put_user(value, (unsigned long*)arg);
         break;

      …
```

Read from register and copy to user space

Kernel Module

- **`/dev/fpga` file operations**

```
static ssize_t fpga_write1(struct file *f, const char __user *buf,
                           size_t count, loff_t *off) {
    int not_copied;

#ifdef DEBUG
    printk(KERN_INFO "\nfpga_drv: receive write command to fpga \n");
#endif

    not_copied = copy_from_user((void *)l.mem_ptr, buf, count);
    return count - not_copied;
}

static ssize_t fpga_read1(struct file *filp, char __user *buf,
                          size_t count, loff_t *offp) {
    int not_copied;

#ifdef DEBUG
    printk(KERN_INFO "\nfpga_drv: receive read command from fpga \n");
#endif

    not_copied  = copy_to_user(buf, (void *)l.mem_ptr, count);
    return count - not_copied;
}
```

File write operation

Copy from user space and write to memory

File read operation

Read from memory and copy to user space

Kernel Module

- **Interrupt handling**

```
static irqreturn_t fpga_int_handler(int irq, void *lp) {
   interruptcount++;

#ifdef DEBUG
   printk(KERN_INFO "\nfpga_drv: Interrupt detected in kernel \n");
#endif

   writel(0ul, (volatile unsigned int *)&l.reg_ptr[3]);

   kill_fasync(&l.fasync_fpga_queue, SIGIO, POLL_IN);

   return IRQ_HANDLED;
}

static int fpga_fasync1 (int fd, struct file *filp, int on) {
#ifdef DEBUG
   printk(KERN_INFO "\nfpga_drv: Inside fpga_fasync \n");
#endif
   return fasync_helper(fd, filp, on, &l.fasync_fpga_queue);
}
```

Interrupt handler

Acknowledge interrupt

Signal user application

File operation to register user-level signal notifications with `/dev/fpga` device

Application

- **Open device**

```
fd = open("/dev/fpga", O_RDWR);
if(fd < 0) {
      printf("Unable to open /dev/fpga.  Ensure it exists!\n");
      return -1;
}

if(fcntl(fd, F_SETOWN, getpid())) err(1, "Set owner");
if(fcntl(fd, F_SETFL, fcntl(fd, F_GETFL)|O_ASYNC)) err(1, "Set flags");
```

Open `/dev/fpga` device

Set owner & flags
(O_ASYNC enables signals)

- **`ioctl` operations**

```
#define READ_CMD  (0x0 << 31)
#define WRITE_CMD (0x1 << 31)

if(ioctl(fd, WRITE_CMD + 0, &val)) err(1, "Writing");

if(ioctl(fd, READ_CMD + 0, &result)) err(1, "Reading SystemC time");

if(ioctl(fd, READ_CMD + 4, &result)) err(1, "Reading SystemC clock");

if(ioctl(fd, READ_CMD + 3, &result)) err(1, "Reading interrupt");

val = 1;
if(ioctl(fd, WRITE_CMD + 3, &val)) err(1, "Trigger interrupt");
```

Read/write device registers
(via kernel driver)

Application

- **File operations**

```
write(fd, &val, sizeof(val));

read(fd, &result, sizeof(result));
```

Write/read to/from device memory
(via `/dev/fpga` file I/O)

- **Interrupt handling**

```
void sighandler(int signo) {
  if(signo==SIGIO){
      det_int++;
      printf("\nInterrupt detected\n");
  }
}
```

Signal handler for receiving events
(interrupts signaled via driver)

```
sigemptyset(&action.sa_mask);
sigaddset(&action.sa_mask, SIGIO);
action.sa_handler = sighandler;
action.sa_flags=0;
if(sigaction(SIGIO, &action, NULL)) err(1, "Setting signal handler");

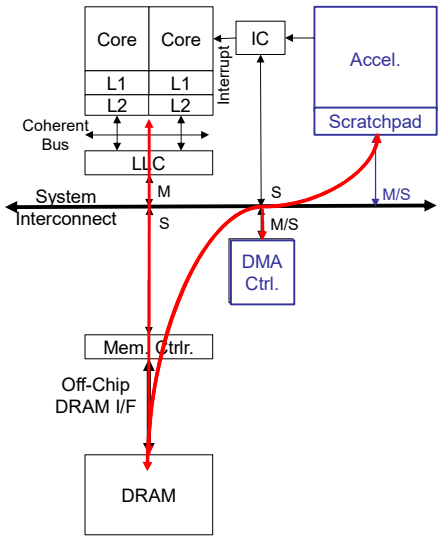
…

while(!det_int) continue;
```

Install signal handler
(for SIGIO from driver)

Wait for interrupt

DMA or Bus-Mastering Accelerator



- **Data sharing through DRAM**
  - DRAM address space
- **Virtual memory in CPU**
  - Address translation
  - Non-contiguous in DRAM

➢ **Accelerator/DMA is virtual**
  ➢ IO-MMU (SMMU in Zynq)
➢ **Reserved DRAM space**
  ➢ `mem=x` kernel command line
  ➢ *mmap()* / *ioremap*()
➢ **Contiguous memory allocator**
  ➢ Reserved CMA pool (`cma=x`)
  ➢ *dma_alloc_coherent()* / *dma_map_single()*
  ➢ *dma_mmap_coherent()*

https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

Lecture 10: Outline

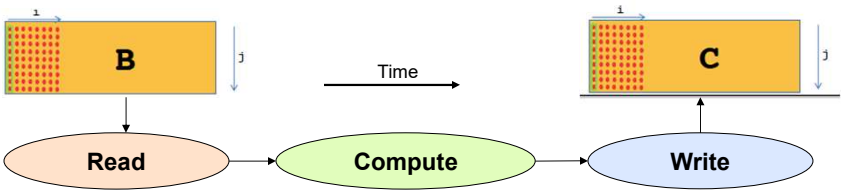
✓ **Hardware interfacing**
  ✓ Accelerator & SoC architecture
  ✓ Accelerator coupling
  ✓ Zynq architecture

✓ **Software interfacing**
  ✓ Application mapping
  ✓ Hardware abstraction layer (HAL)
  ✓ Drivers

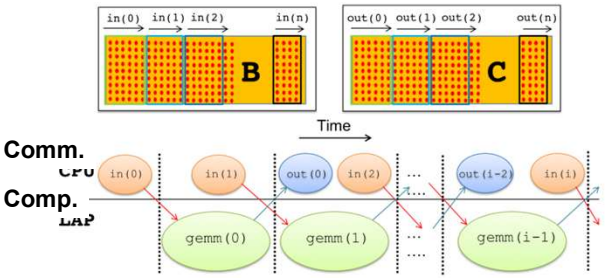
- **System optimization**
  - Data movement

Communication Hiding

- **Accelerator communication and computation**

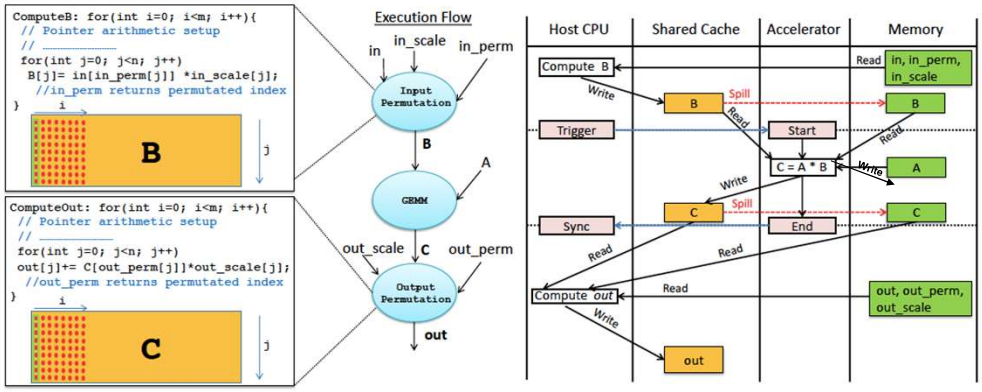


- **Blocking, pipelining, overlapping (prefetching/poststoring)**
  - Extreme case: streaming (item-by-item)



Application Patterns

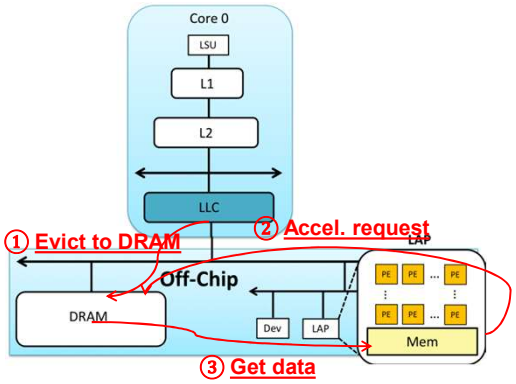
- **Dataflow patterns**
  - Producer/consumer
  - Pre-/post-processing on CPU

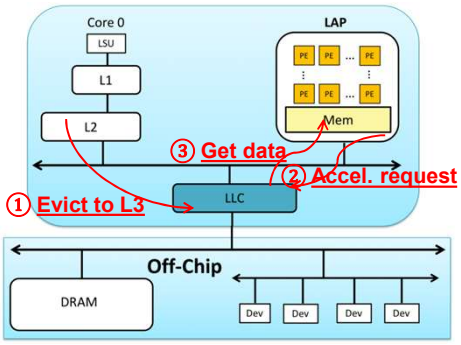


N-body problem example [TPDS'21]

Accelerator Placement

- **Off-chip at DRAM level**
- **On-chip at shared LLC**





Processing in/near memory (PIM)
(data in DRAM)

On-chip dataflow & staging
(cache-contained vs. DRAM spills)