Data Motion Network and Optimization

SDx 2018.2

Objectives

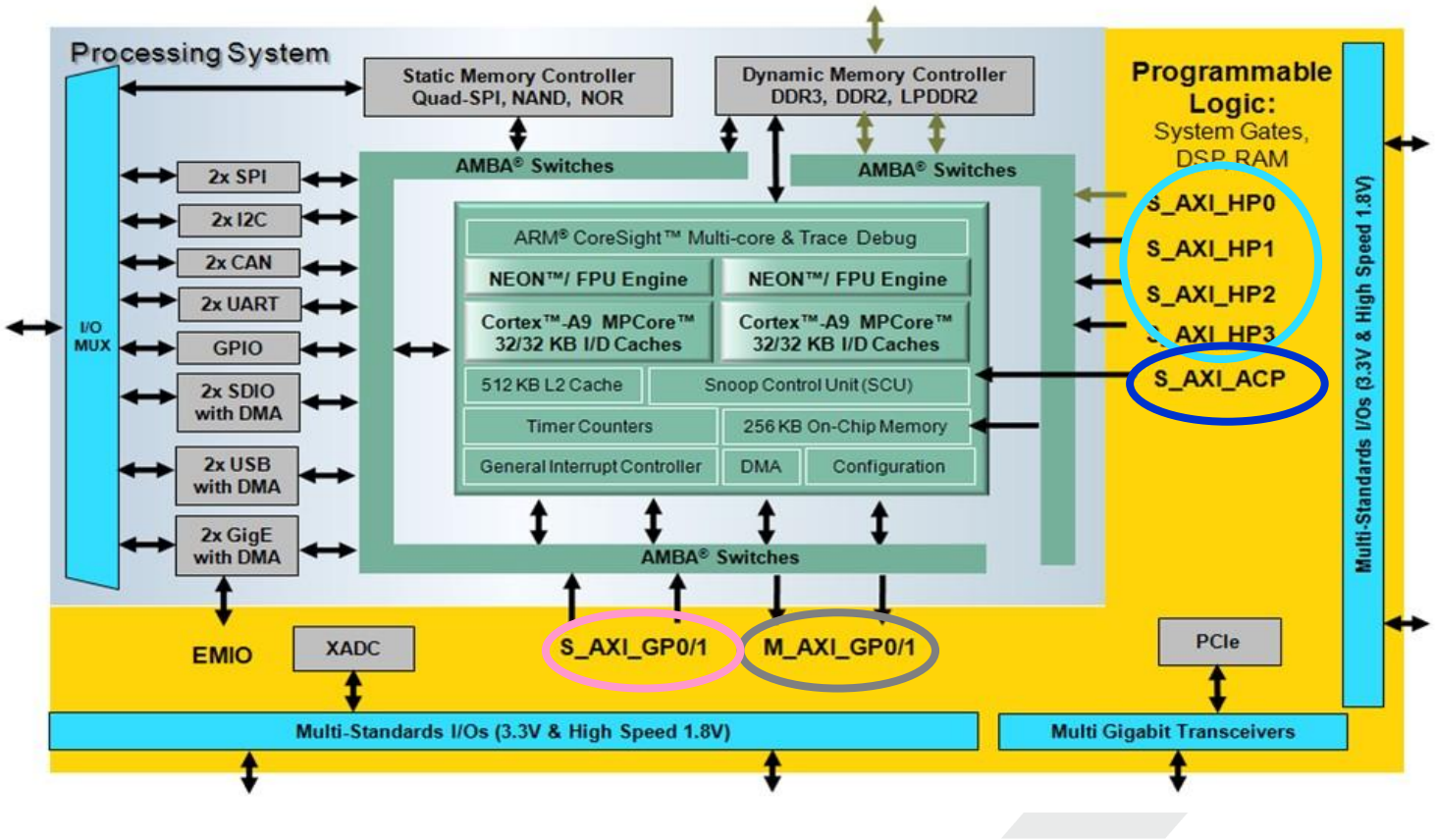
- After completing this module, you will be able to:
  - List various interface ports available to move data between the processor and user-generated IP in PL
  - Describe at the hardware level how caches interact with the accelerator coherency port (ACP)
  - List available functions to perform data exchanges and synchronize events
  - Identify some of the techniques used for hardware optimization

Outline

- Architectural Support for Data Motion
- Data Motion
- Optimization
- Summary
- Lab2 Intro

PS-PL Interface AXI Ports

- **The AMBA AXI ports of the PS-PL interface provide the primary mechanism for the flow of data between the PS and PL**
  - Two general-purpose master ports
  - Two general-purpose slave ports
  - Four high-performance slave ports
  - One accelerator coherency port (ACP) slave port



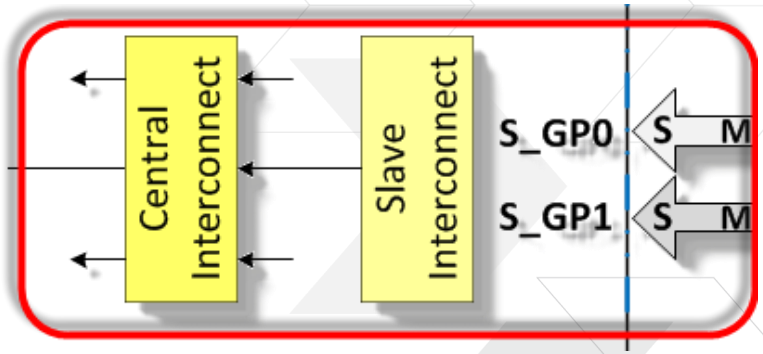
XILINX.

General-Purpose Master Ports

- **Two identical 32-bit AXI master ports**
  - M_AXI_GP0
  - M_AXI_GP1
- **Attaches to slave AXI ports in programmable logic via PL AXI interconnect**
  - PL slave
    - Your peripheral built in programmable logic
    - IP Integrator interface or other third-party-based IP
- **Mostly used for CPU and I/O peripheral block data movement to programmable logic**
- **Why two ports?**
  - Each port, having it's own 1 GB space, is capable of driving a number of peripherals using an AXI switch
  - Multiple ports enable the designer to distribute bandwidth
- **PS masters can be**
  - Cortex-A9 processors via the L2 cache controller
  - USB, Ethernet, SD/SDIO controllers
  - DMAC
  - Debug access port

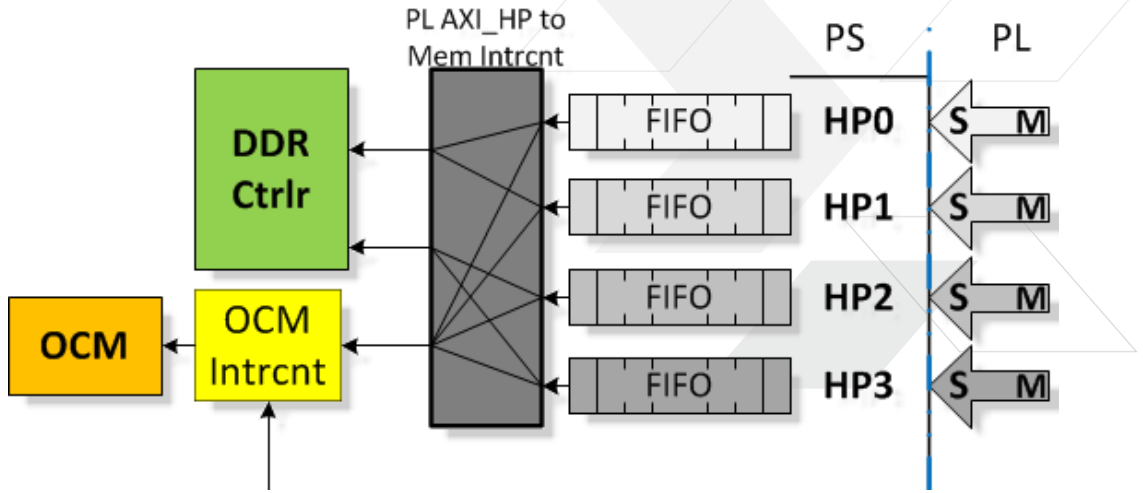
General-Purpose Slave Ports

- **Two identical 32-bit AXI slave ports**
  - S_AXI_GP0
  - S_AXI_GP1
- **Provides initiator access from PL master to PS target**
- **Attaches to master AXI port in PL via AXI interconnect**
  - PL master
    - MicroBlaze processor
    - Your master built in programmable logic
    - Other third-party-based IP
- **PS slaves can be**
  - DDRx controller
  - On-chip memory (OCM)
  - IOP peripheral



High-Performance Slave Ports

- **Asynchronous crossing between the PL clock domain and PS clock domain**
  - S_AXI_HP0, S_AXI_HP1
  - S_AXI_HP2, S_AXI_HP3
- **Provides initiator access from PL master to PS memory**
- **Provides low latency access to DDR and OCM**
- **Attaches to master AXI port in PL**
  - DMA controller
  - MicroBlaze processor
  - Custom master built in programmable logic
  - Other third-party based IP
- **QoS supported from the programmable logic ports**
- **FIFOs smooth out of *long-latency* events**
  - 1 KB (128 by 64 bit) data FIFOs
  - Both read and write paths



XILINX.

Accelerator Coherency Port (ACP) Slave Port

- **One accelerator coherence port (ACP)**
  - S_AXI_ACP
  - High-performance, coprocessor interface to PL
    - Accelerators gain access to the CPU cache hierarchy
      - Enables PL to participate in the coherency among the caches and DDRx memory
    - Supported in hardware; no software needed
  - No drivers required for the ACP; however, the IP connected to the ACP may require drivers
- **Provides initiator access from PL master to PS target**
  - Via L1 and L2 cache
- **Attaches to master AXI port in PL**
  - Typically a PL-based co-processor/accelerator
  - Also could be
    - MicroBlaze processor
    - Custom master built in programmable logic
    - Other third-party-based IP

Why Connection to the SCU is Important for Accelerators

- **ACP bolts directly into the snoop control unit (SCU)**
  - ACP is a PS slave AXI port (that is, accelerator is a master)
  - Enables the accelerator on the ACP to write directly into the L1 and L2 caches
    - And indirectly to the DDR/OCM
  - Data movement is limited by the size of the cache target
    - Typically L2 is the target due to its larger size
    - Cache misses result in moving data to the DDR
    - Frequent misses indicate that the HP port (not the ACP port) should have been used
- **When the ACP is not used, there is no non-standard use of the caches**

System-Level Considerations that Determine Coding Techniques

- **Which PS interfaces to use for the accelerator?**
  - AXI_HP
  - AXI_ACP
  - AXI_GP (AXI_lite for control)
- **Where does the input to the accelerator connect from?**
  - From PS/PS DDR interface
  - MIG
  - From an external interface (like video)
- **Where does the output from the accelerator connect to?**
  - To PS
  - To an external interface
- **What is the optimal data movement mechanism between the PS and accelerator?**
  - DMA driven
    - Simple
    - Scatter gather
  - Software driven
  - How to manage cache coherency?
    - Software
    - SCU
- **What is the optimal signaling between PS and the accelerator?**
  - Interrupt driven
  - Event interface
  - Polled

Preferred Interfaces for Hardware Accelerators

- **AXI_HP DMA or AXI_ACP DMA give the highest performance**
  - Both DDR and OCM (4x64K) accessible, lower latency through HP ports
- **AXI_ACP offers further performance when cache coherent data transfers are implemented to/from the accelerator**
- **When is ACP more suited?**
  - Not suited for instruction-level acceleration; tightly coupled processor registers perform best
  - Not suited for large data sets; this can cause cache thrashing and degrade performance
  - Suited for small to moderate size data sets that can fit into L1 (<32K) or L2 (<256K) caches

Comparing Data Movement Methods (1)

Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
CPU Programmed I/O	• Simple software • Fewest PL resources • Simple PL slaves	• Lowest throughput	• Control functions	<25 MB/s
PS DMAC	• Fewest PL resources • Medium throughput • Multiple channels • Simple PL slaves	• Somewhat complex DMA programming	• Limited PL resource DMAs	600 MB/s
PL AXI_HP DMA	• Highest throughput • Multiple interfaces • Command/data FIFOs	• OCM/DDR access only • More complex PL master design	• High-performance DMA for large datasets	1200 MB/s (per interface)

Comparing Data Movement Methods (2)

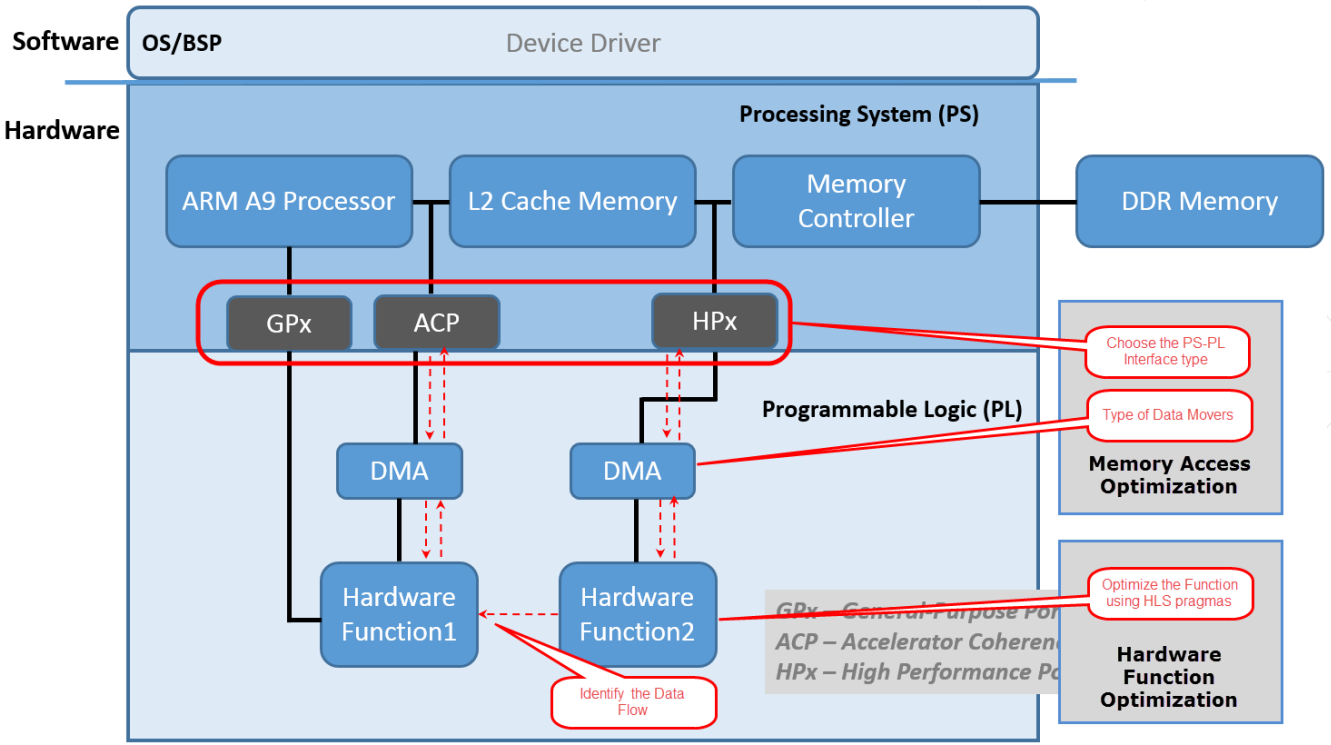
Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
PL AXI_ACP DMA	• Highest throughput • Lowest latency • Optional cache coherency	• Large burst might cause cache thrashing • Shared CPU interconnect bandwidth • More complex PL master design	• High-performance DMA for smaller, Coherent datasets • Medium granularity CPU offload	1200 MB/s
PL AXI_GP DMA	• Medium throughput	• More complex PL master design	• PL to PS control functions • PS I/O peripheral access	600 MB/s

Data Motion

XILINX

Data Movement in Zynq

- Data may move between master/slave IP located in PL and memories located internal to the processor or external to the chip using the nine ports described in the previous section
- The decision is made by the SDSoC compiler after analyzing the software or following user's directives



XILINX.

How SDSoC Compiler Maps Programs to HW/SW?

- **User specifies a program, a platform and a set of functions to map to hardware**
  - This defines the SDSoC-generated hardware / software interface
  - Program may also call other software functions provided by the platform, e.g., “write LED”, which are not targeted to hardware
- **The SDSoC system compiler analyzes the program and maps it into a hardware / software system**
  - Preserves program semantics, including calls to hardware functions
  - Hardware functions can run concurrently with well-defined synchronization
- **Hardware function calls define “connections” implemented using…**
  - Program properties: data flow between calls to hardware functions, memory allocation,…
  - Function argument properties: payload size, hardware interfaces,…
  - Function properties: memory access patterns, implementation latency,…

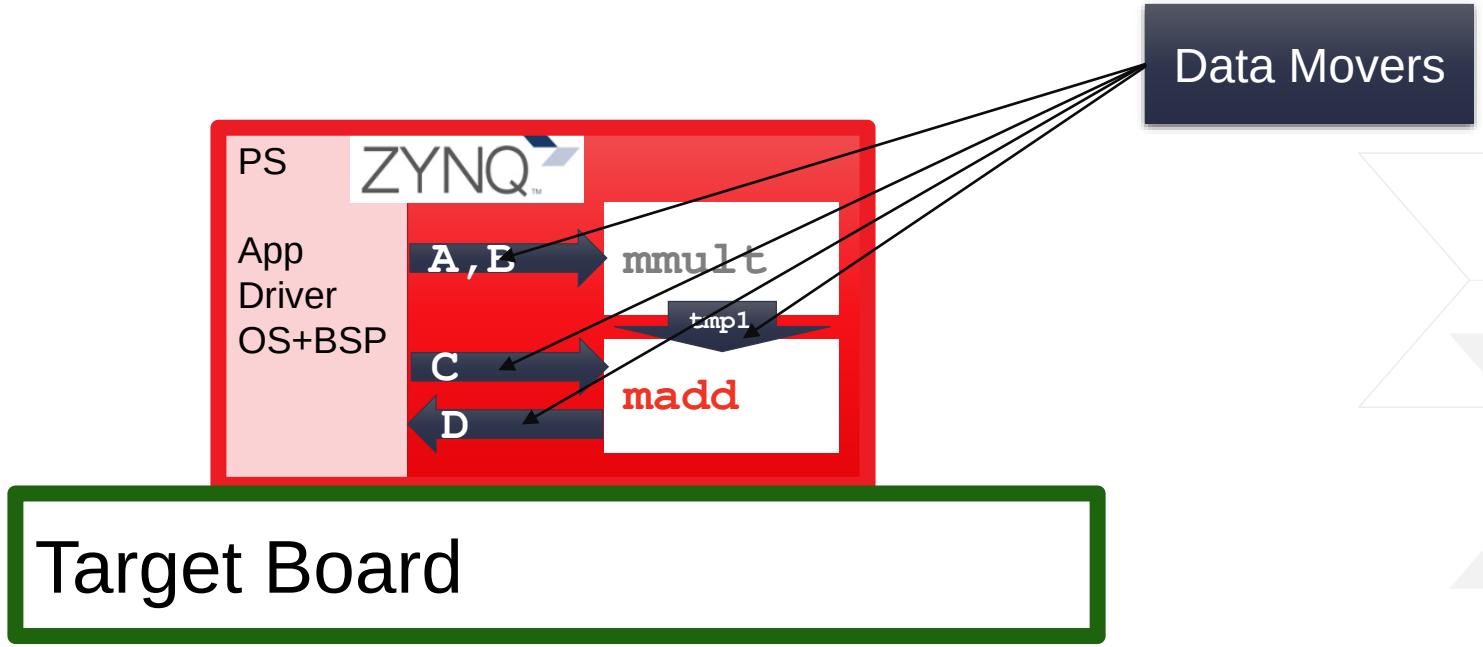
How SDSoC Compiler Maps Programs to HW/SW? (2)

> Program source code

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          madd(tmp1, C, D);

          mmult_sw(A, B, tmp2);
          madd_sw(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```

How SDSoC Compiler Maps Programs to HW/SW? (3)

- **Structure of generated hardware system**



XILINX.

How SDSoC Compiler Maps Programs to HW/SW? (4)

- Structure of generated software

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     std::cout << "Testing mmult .." << std::endl;

     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          _p0_mmult_0(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          _p0_madd_0(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }

     return true;
}
```

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
            start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);

mult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
mult_0.in_B), in_B, 1024 * 4, &_p0_request_3);
```

Control transfer

Data transfers

```
void _p0_madd_0(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_madd_0_cmd;
  cf_send_i(&(_p0_swinst_madd_0.cmd_madd), start_seq, 3*sizeof(int),
            &_p0_swinst_madd_0_cmd);
  cf_wait(_p0_swinst_madd_0_cmd);

  cf_send_i(&(_p0_swinst_madd_0.B_PORTA), B, 1024 * 4, &_p0_request_0);
  cf_receive_i(&(_p0_swinst_madd_0.C_PORTA), C, 1024 * 4,
               &_p0_madd_0_num_C_PORTA, &_p0_request_1);
  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

Control transfer

Data transfers

XILINX.

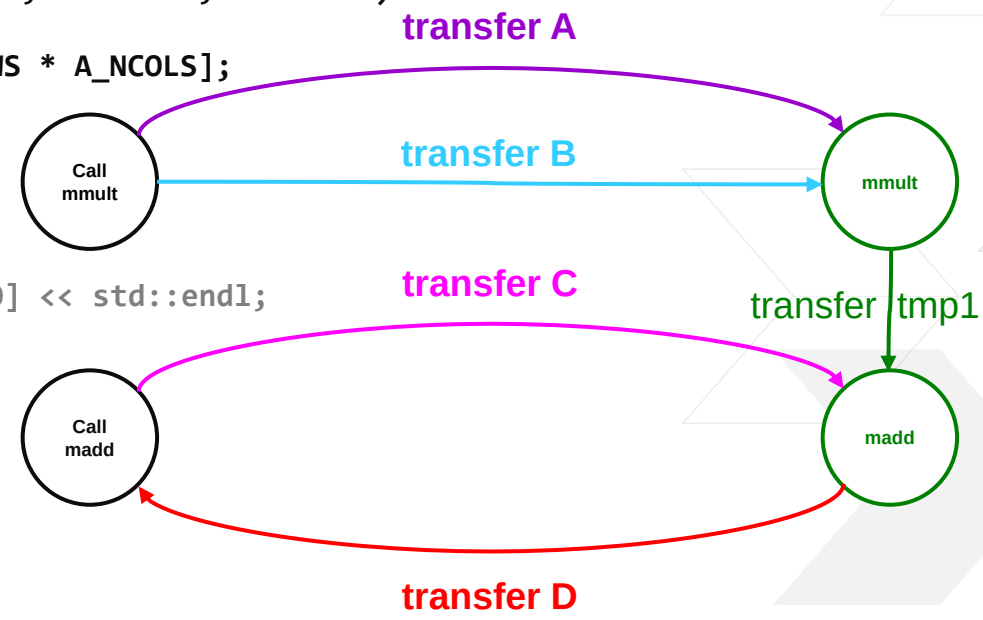
Optimization

System Level Optimization Using SDSoC

- Program connectivity and implementation
- Datamovers
- Memory models
- Platform port selection
- Direct connections
- Hardware function interface

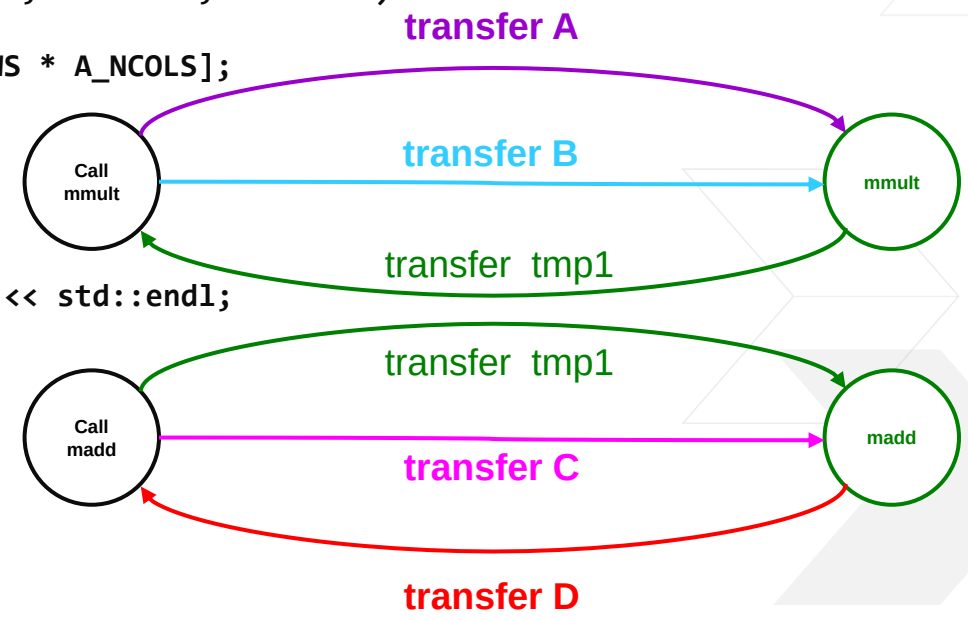
Program Connectivity

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



Program Connectivity (2)

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



> **Simple program changes can cause significant system changes!**

Program Connectivity Implementation

- **Every connection with data movement between the software program and a hardware function requires a *datamover***
- **Program connections can have multiple possible implementations**
  - SDSoC compiler infers datamovers and system ports
  - Datamover and system ports can be overridden with pragma
- **Datamover inference is based on payload size, payload memory attributes, and hardware function argument access pattern**
  - `#pragma SDS data data_mover(arg:<DM_TYPE>) // to override`
- **Note: `#pragma SDS` is always treated as a rule, not a hint**
  - Not all errors will be caught, so care is required

Datamovers

- **Every datamover has two components**
  - Hardware IP, e.g., axi_dma, a hardware function or a CPU
  - OS-specific software library function (userspace for Linux)

Datamover	IP	IP port types	Payload (bytes)	Phys. memory contiguity
axilite	ps7	register, axilite		either
axi_dma_simple	axi_dma	bram,ap_fifo, axis	< 8M	contiguous
axi_dma_sg	axi_dma	bram,ap_fifo, axis		either
axi_dma_2d	axi_dma	bram		contiguous
axi_fifo	axi_fifo_mm_s	bram,ap_fifo, axis	≤ 300	non-contiguous
zero_copy	HW fcn IP	m_aximm		contiguous

Memory Models

- **sds_lib.h functions to “declare” memory properties**
  - `sds_alloc(size_t size);` `// guaranteed contiguity`
  - `sds_mmap(void *paddr, size_t size, void *vaddr);` `// assumed contiguity`
  - `sds_register_dmabuf(void *vaddr, int fd);` `// assumed contiguity`
- **`#pragma SDS data copy (A[offset:array_size])`**
  - Copy in, copy out semantics (e.g., axi_dma, axi_fifo)
  - Also used to specify variable and non-default transfer sizes
- **`#pragma SDS zero_copy(A[offset:array_size])`**
  - Shared memory semantics (hardware function m_aximm only)
- **`#pragma SDS data access_pattern (A:SEQUENTIAL|RANDOM)`**
  - Only valid for “copy” semantics
  - SDSoC generates FIFO or BRAM hardware interface accordingly

Platform Port Selection

- **Compiler infers ACP or HP (AFI) ports based on memory attributes and other criteria**
  - `#pragma SDS data sys_port (A:ACP|AFI|MIG)` to override inference rules
    - `A` must be one of the formal arguments of the function

- **Memory is by default assumed CACHEABLE, i.e., compiler maintains cache coherency between CPU and hardware function**
  - Cache flush before transferring data to hardware function
  - Cache invalidate before transferring data from hardware function

Direct Hardware Connections

- **Compiler maps data movement between hardware functions onto direct connections when possible**
- **Direct connections implemented entirely in hardware via AXI streams**

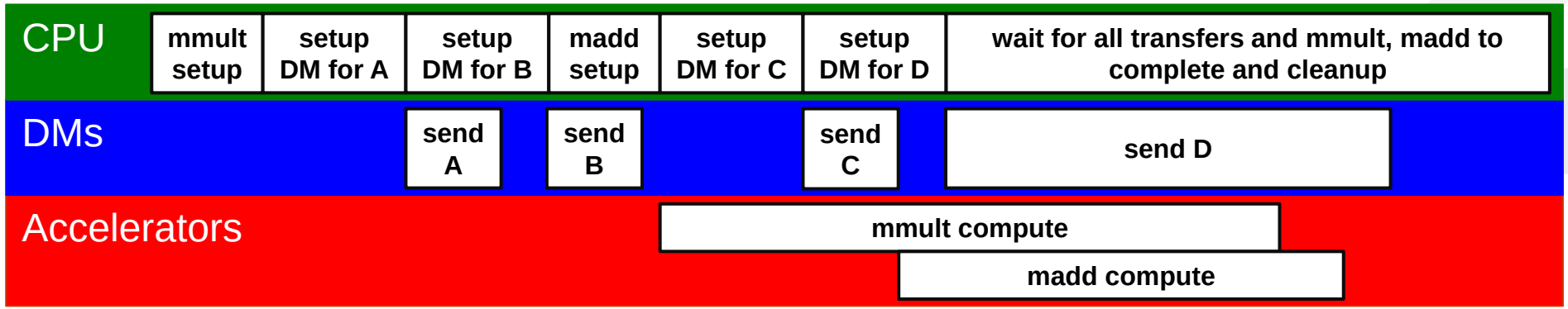
Port IF	BRAM	FIFO	AXIS
BRAM	Y	Y	64-bit TDATA
FIFO	Y	Y	64-bit TDATA
AXIS	64-bit TDATA	64-bit TDATA	Equal TDATA width

- **User must guarantee sufficient buffering**
  - `#pragma SDS data buffer_depth(A:<buffer_depth>)`

Interface	Type	Depth	Array Size	Data width	Default
BRAM	multi-buffer	1:4	2:16384	8:16:32:64	1
FIFO	buffer	2:(16384/array_size)	2:16384	8:16:32:64	1024

Hardware Function Call Sequencing

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



Hardware Function Call Interface

- **SDSoC rewrites calls to hardware functions and replaces function by a stub to transfer data and control**

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     std::cout << "Testing mmult .." << std::endl;

     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          _p0_mmult_0(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          _p0_madd_0(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }

     return true;
}
```

In _sds/swstubs directory

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0_cmd;
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
              start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);

  cf_send_i(&(_p0_swinst_mmult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
  cf_send_i(&(_p0_swinst_mmult_0.in_B), in_B, 1024 * 4, &_p0_request_3);


}
```

Control transfer

Data transfers

Hardware Function Call Interface (2)

- `cf_wait()` synchronization barriers automatically inserted at the end of hardware function pipeline

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     std::cout << "Testing mmult .." << std::endl;

     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          _p0_mmult_0(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std
          _p0_madd_0(tmp1, C, D);
```

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0_cmd;
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
              start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);

  cf_send_i(&(_p0_swinst_mmult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
  cf_send_i(&(_p0_swinst_mmult_0.in_B), in_B, 1024 * 4, &_p0_request_3);


}
```

```
void _p0_madd_0(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_madd_0_cmd;
  cf_send_i(&(_p0_swinst_madd_0.cmd_madd), start_seq, 3*sizeof(int),
              &_p0_swinst_madd_0_cmd);
  cf_wait(_p0_swinst_madd_0_cmd);

  cf_send_i(&(_p0_swinst_madd_0.B_PORTA), B, 1024 * 4, &_p0_request_0);
  cf_receive_i(&(_p0_swinst_madd_0.C_PORTA), C, 1024 * 4,
               &_p0_madd_0_num_C_PORTA, &_p0_request_1);
  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

`cf_wait()` for madd

`cf_wait()` for mmult

XILINX.

Summary

Summary

- **High-performance AXI port connects the accelerator to DDR/OCM or SCU for data moving**
  - Internal routing and port select can provide low-latency paths between the accelerator and memory
- **Caches provide both processors and accelerator with coherent data**
  - Caches can be locked and flushed as needed for the accelerator
- **Processor provides clock sources to accelerators**
- **Compiler rewrites calls to hardware functions and replaces function by a stub to transfer data and control**
- **There are three functions available to exchange data and synchronize events**
  - `cf_send`
  - `cf_receive`
  - `cf_wait`

Lab2 Intro

Lab2 Intro

- **Introduction**
  - This lab guides you through the process of handling data movements between the software and hardware accelerators using various pragmas and functions
- **Objectives**
  - Use pragmas to select ACP or AFI ports for data transfer
  - Use pragmas to select different data movers for your hardware function arguments
  - Understand the use of sds_alloc() and sds_free() calls
  - Understand the use of malloc() and free() calls
  - Analyze built hardware

Adaptable.
Intelligent.

XILINX