ECE 699: Lecture 6

Using DMA

& AXI4-Stream

Required Reading

LogiCORE IP AXI DMA v7.1: Product Guide

- *Chapter 1: Overview*
- *Chapter 2: Product Specification*
- *Chapter 3: Designing with the Core*
- *Chapter 4: Design Flow Steps*
- *Chapter 5: Example Design*

The ZYNQ Book

- *Chapter 2.3: Processing System – Programmable Logic Interfaces*
- *Chapter 9.3: Buses*
- *Chapter 10.1: Interfacing and Signals*

Recommended Reading

M.S. Sadri, ZYNQ Training (presentations and videos)

- *Lesson 1 : What is AXI?*
- *Lesson 2 : What is an AXI Interconnect?*
- *Lesson 3 : AXI Stream Interface*

AXI Interfaces and Interconnects

Interface

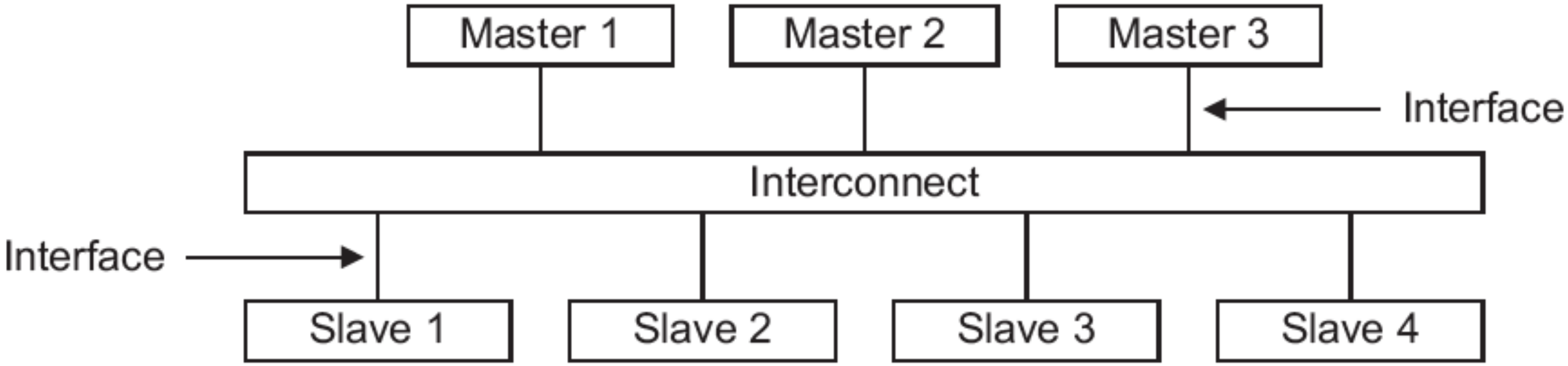
A point-to-point connection for passing data, addresses, and hand-shaking signals between master and slave clients within the system

Interconnect

A switch which manages and directs traffic between attached AXI interfaces

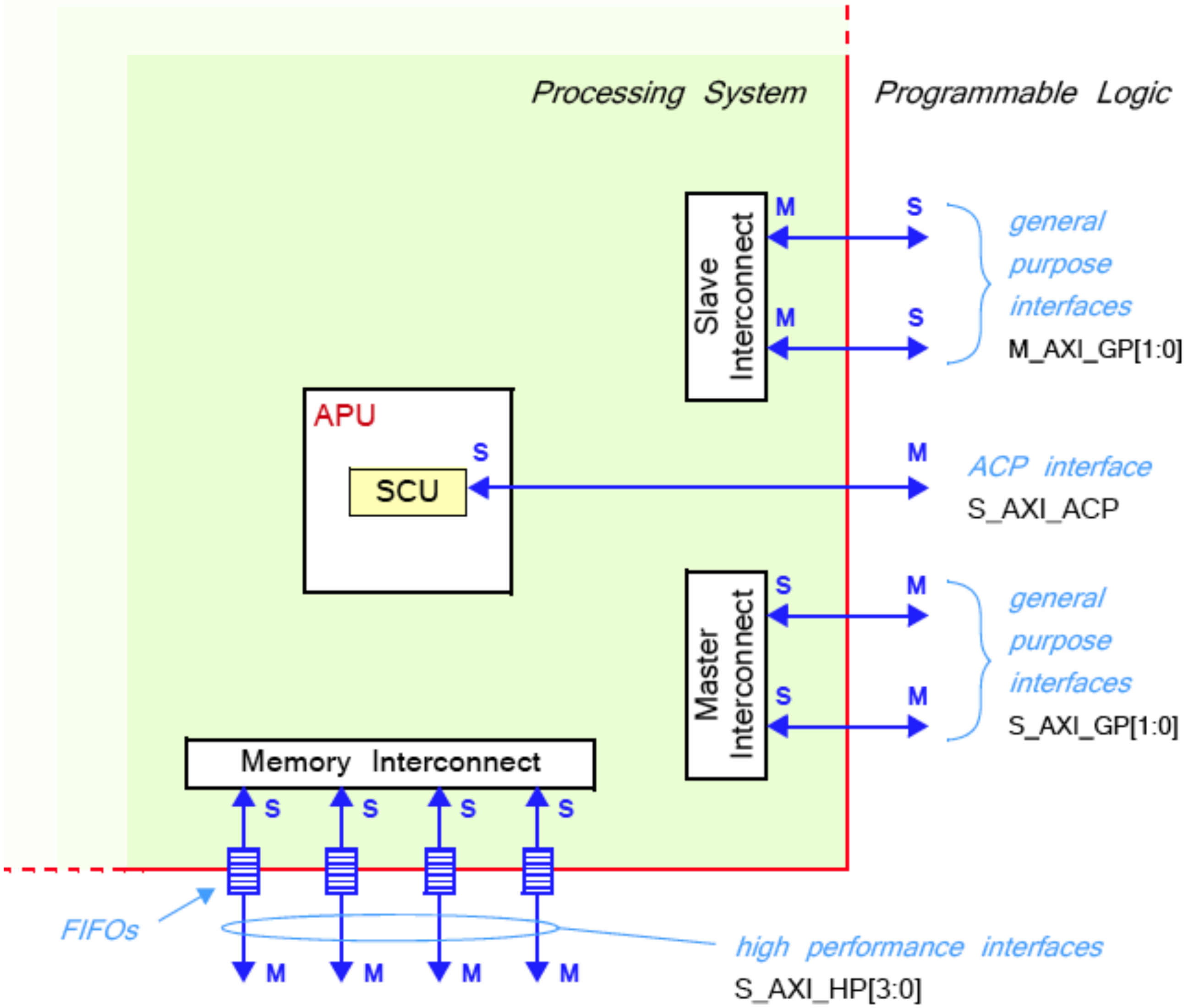
Source: The Zynq Book

Interconnect vs. Interface



Source: ARM AMBA AXI Protocol v1.0: Specification

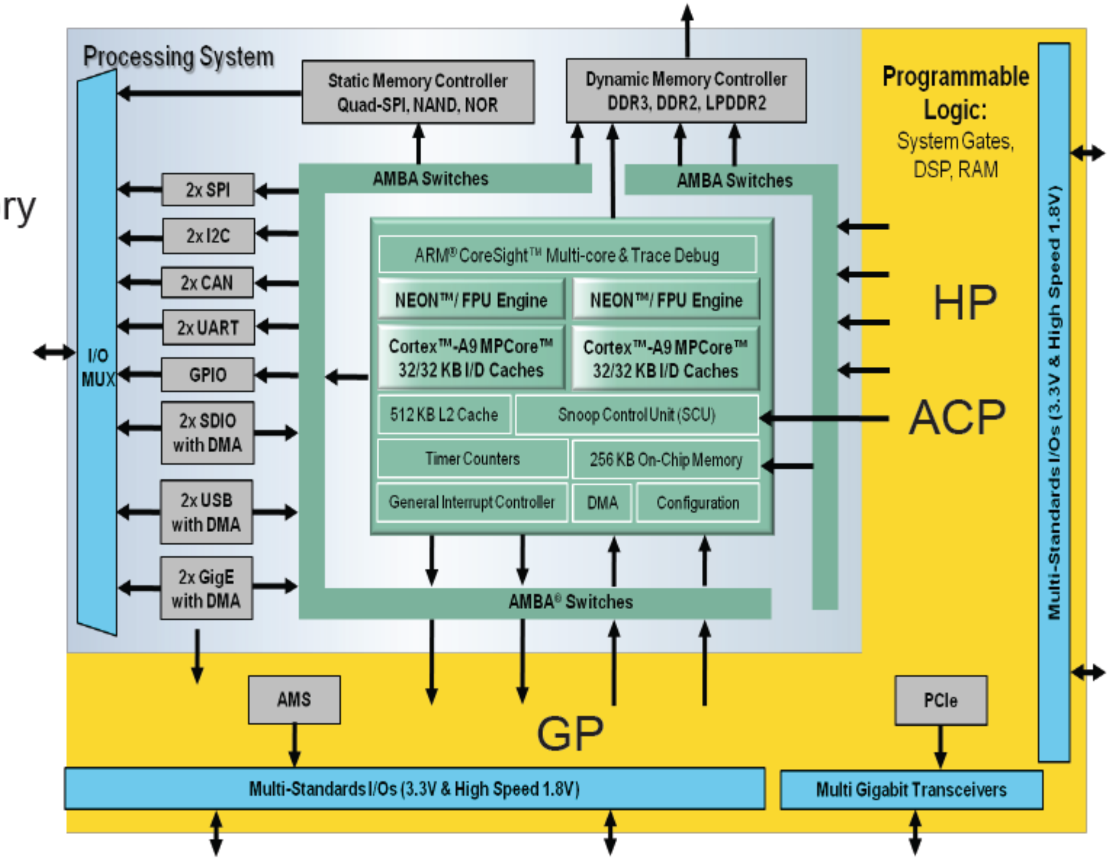
PS-PL Interfaces and Interconnects



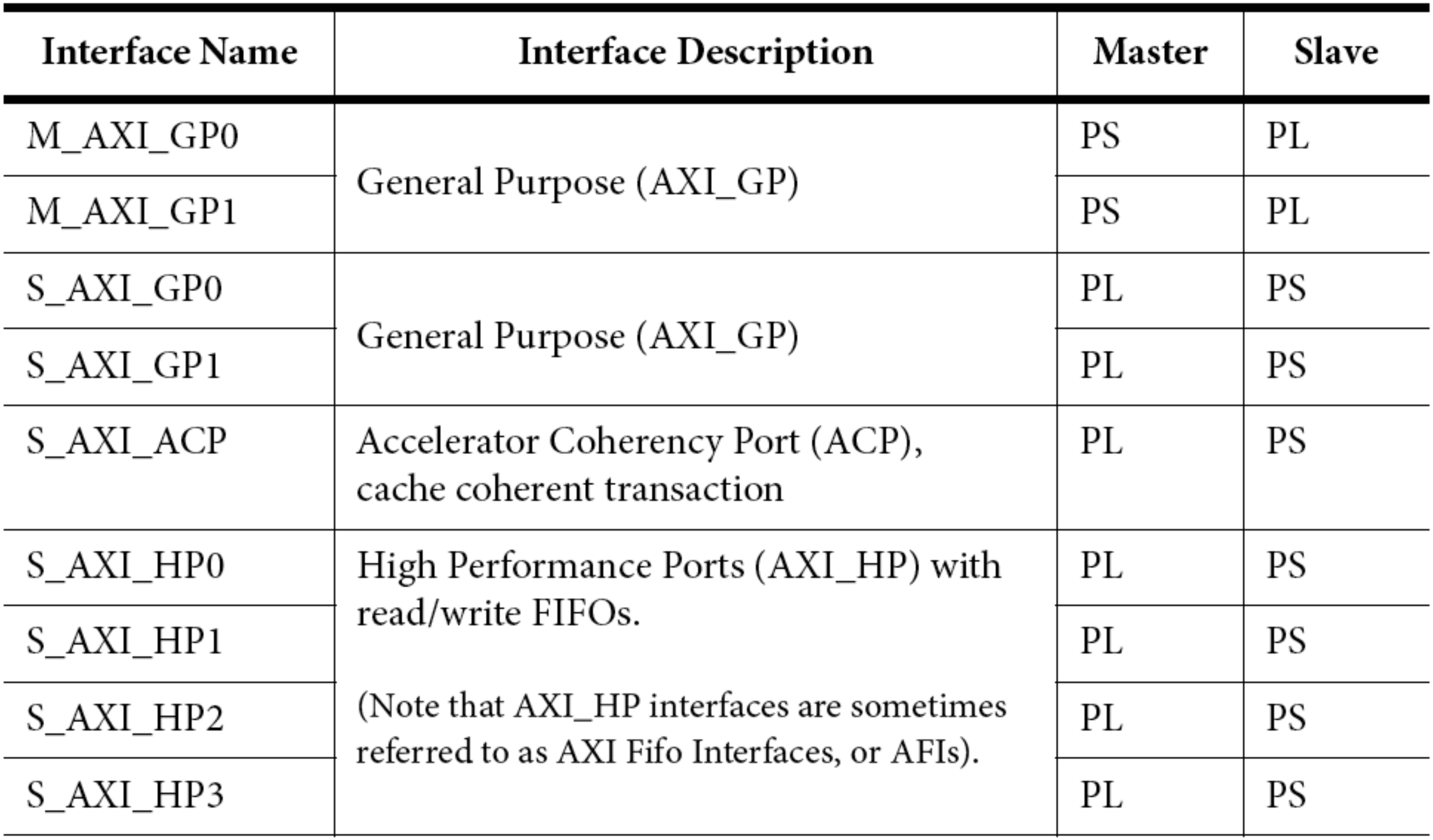
Source: The Zynq Book

Zynq AXI PS-PL Interfaces

- **HP**
  - 4 x 64 bit Slave interfaces
    - Optimized for high bandwidth access from PL to external memory
- **GP**
  - 2 x 32 bit Slave interfaces
    - Optimized for access from PL to PS peripherals
  - 2 x 32 bit Master interfaces
    - Optimized for access from processors to PL registers
- **ACP**
  - 1 x 64 bit Slave interface
    - Optimized for access from PL to processor caches



AXI Interfaces between PS and PL



Interface Name	Interface Description	Master	Slave
M_AXI_GP0	General Purpose (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	General Purpose (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), cache coherent transaction	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs).	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Source: The Zynq Book

General-Purpose Port Summary

- GP ports are designed for maximum flexibility
- Allow register access from PS to PL or PL to PS
- Good for Synchronization
- Prefer ACP or HP port for data transport

High-Performance Port Summary

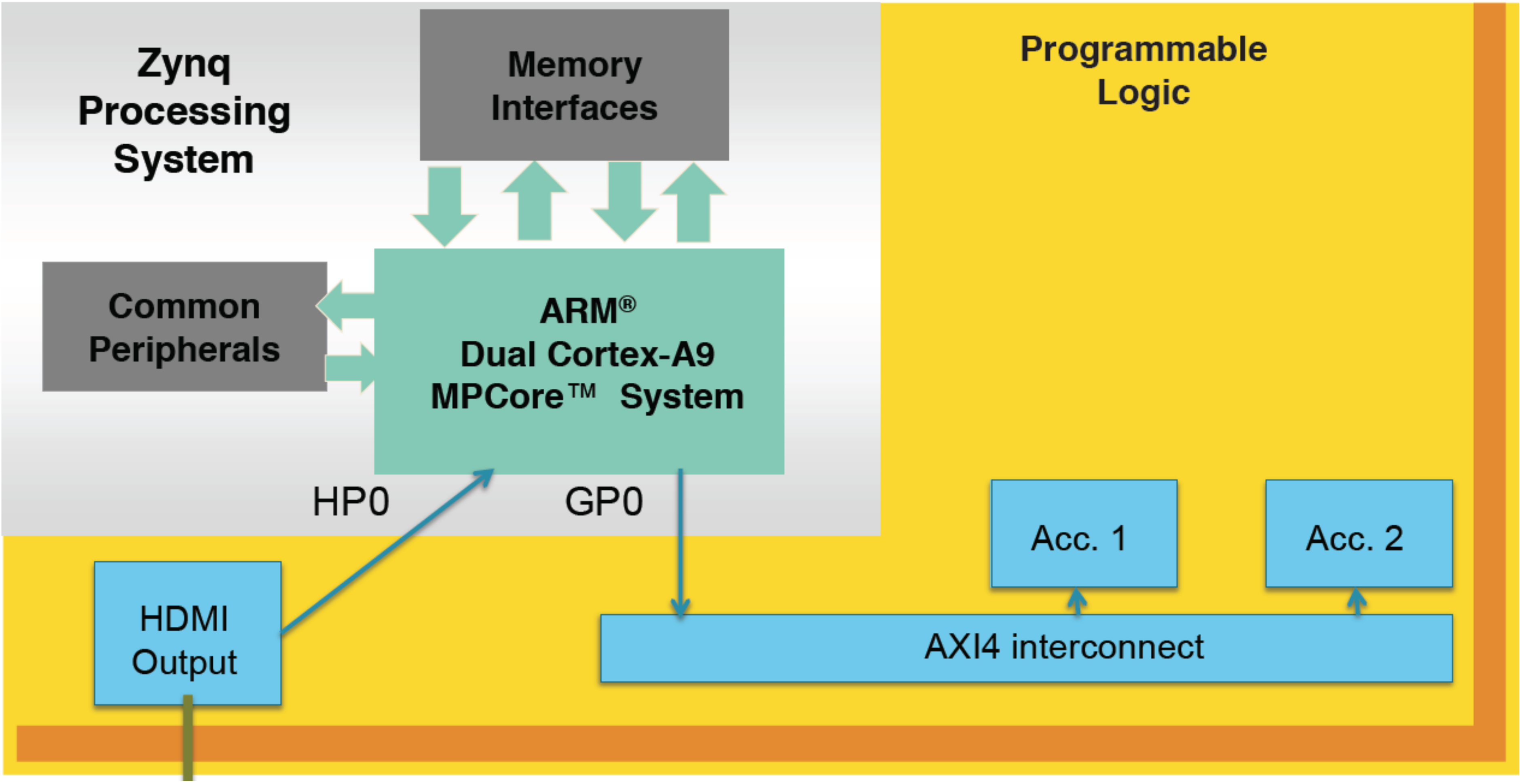
- HP ports are designed for maximum bandwidth access to external memory and OCM
- When combined can saturate external memory and OCM bandwidth
  - HP Ports : 4 * 64 bits * 150 MHz * 2 = 9.6 GByte/sec
  - external DDR: 1 * 32 bits * 1066 MHz * 2 = 4.3 GByte/sec
  - OCM : 64 bits * 222 MHz * 2 = 3.5 GByte/sec
- Optimized for large burst lengths and many outstanding transactions
- Large data buffers to amortize access latency
- Efficient upsizing/downsizing for 32 bit accesses

Accelerator Coherency Port (ACP) Summary

- ACP allows limited support for Hardware Coherency
  - Allows a PL accelerator to access cache of the Cortex-A9 processors
  - PL has access through the same path as CPUs including caches, OCM, DDR, and peripherals
  - Access is low latency (assuming data is in processor cache) no switches in path
- ACP does not allow full coherency
  - PL is not notified of changes in processor caches
  - Use write to PL register for synchronization
- ACP is compromise between bandwidth and latency
  - Optimized for cache line length transfers
  - Low latency for L1/L2 hits
  - Minimal buffering to hide external memory latency
  - One shared 64 bit interface, limit of 8 masters

Accelerator Architecture with Bus Slave

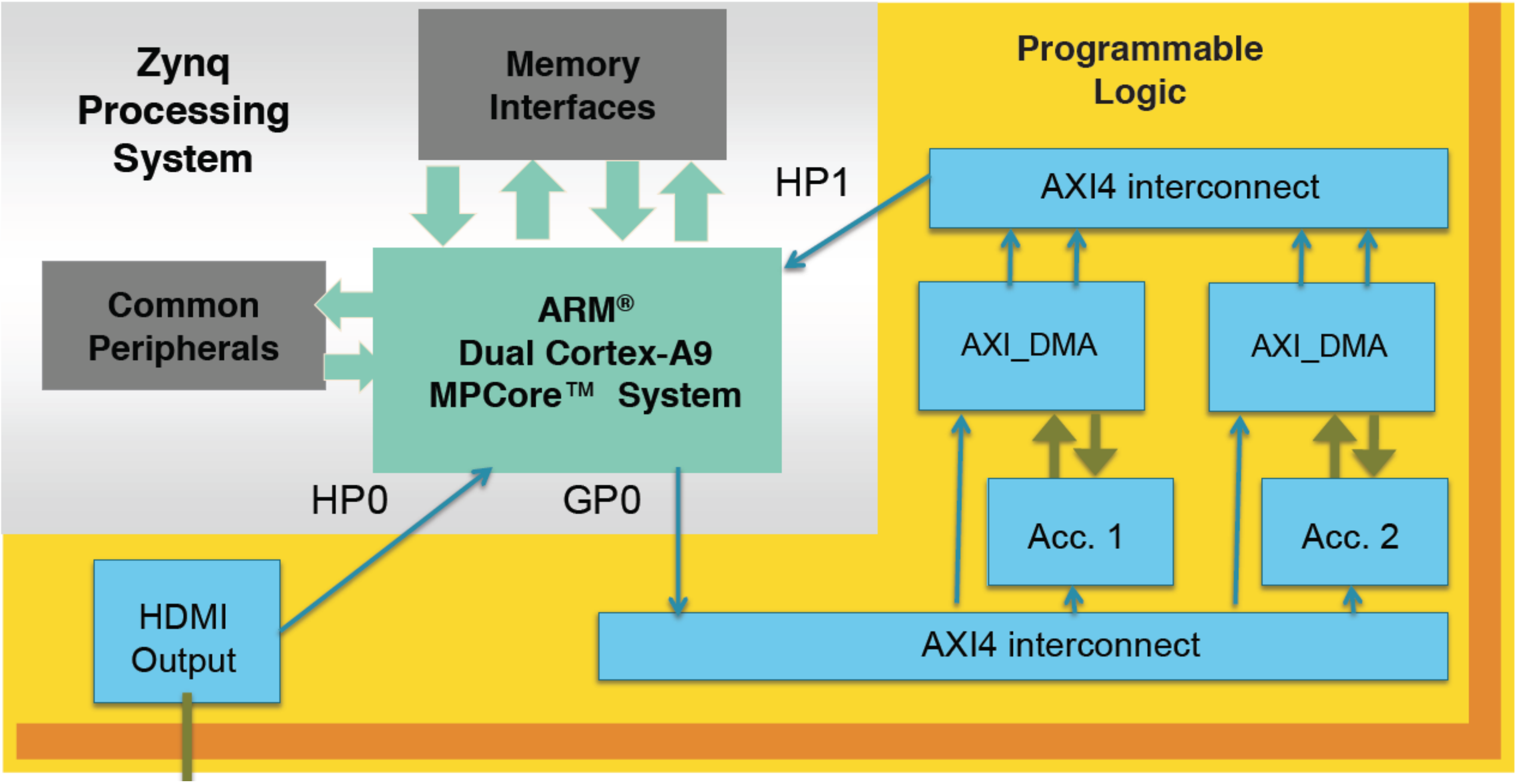
Pro: Simple System Architecture
Con: Limited communication bandwidth



Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

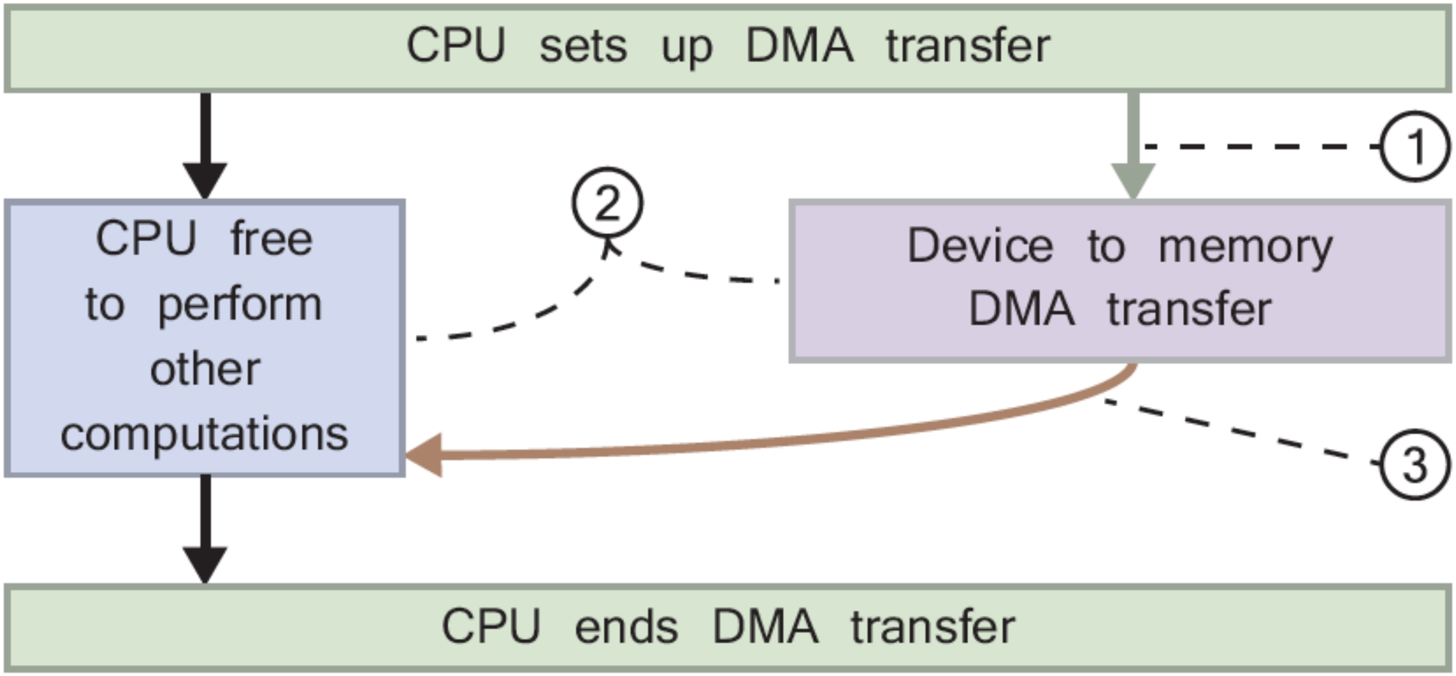
Accelerator Architecture with DMA

Pro: High Bandwidth Communication
Con: Complicated System Architecture, High Latency



Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

DMA Memory Transfer Operation


CPU sets up DMA transfer
1
2
CPU free to perform other computations
Device to memory DMA transfer
3
CPU ends DMA transfer

Source: The Zynq Book

AXI DMA-based Accelerator Communication

Write to Accelerator

- processor allocates buffer
- processor writes data into buffer
- processor flushes cache for buffer
- processor initiates DMA transfer

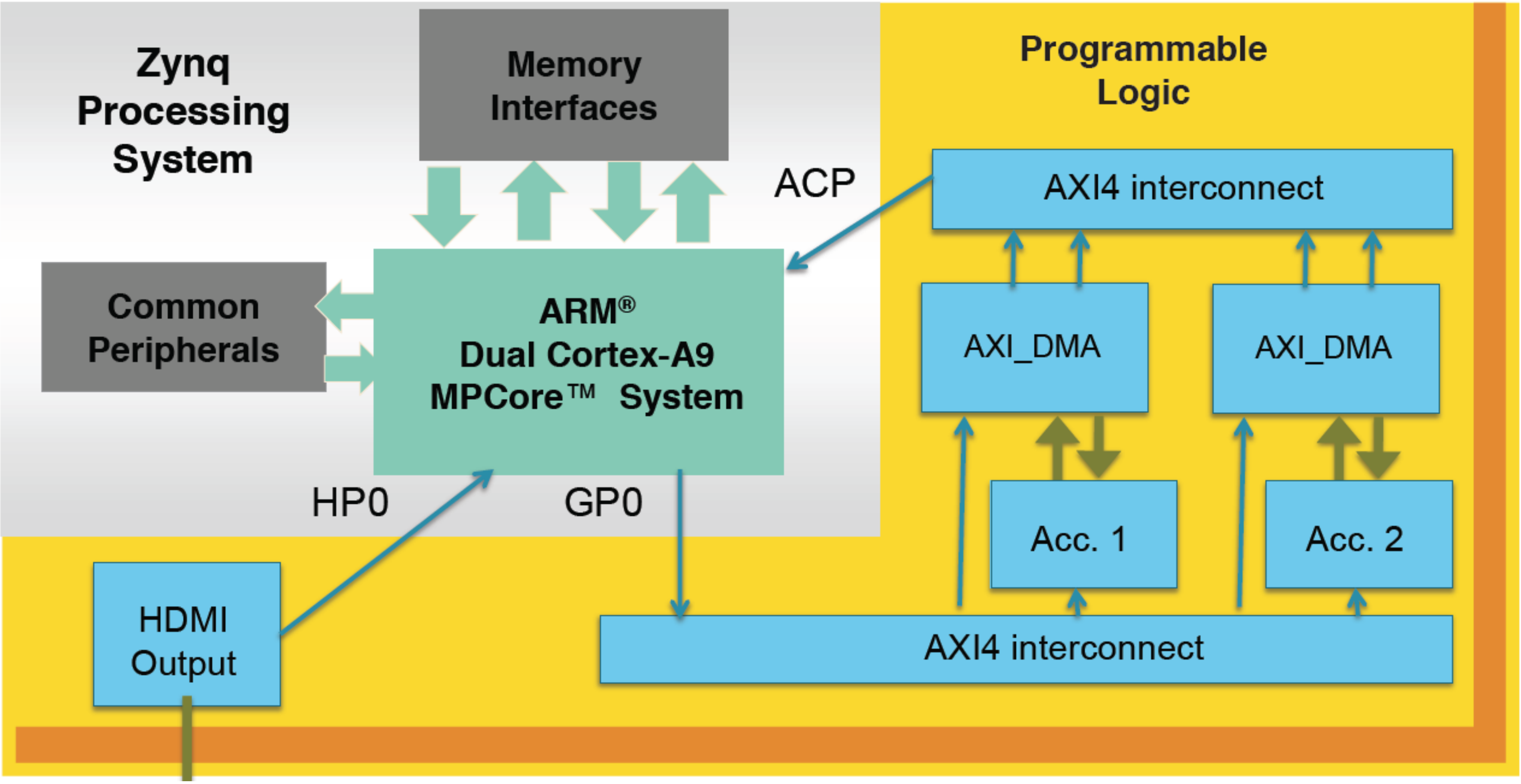
Read from Accelerator

- processor allocates buffer
- processor initiates DMA transfer
- processor waits for DMA to complete
- processor invalidates cache for buffer
- processor reads data from buffer

Accelerator Architecture with Coherent DMA

Pro: Low latency, high-bandwidth communication

Con: Complicated system architecture, Limited to data that fits in caches



Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

Coherent AXI DMA-based Accelerator Communication

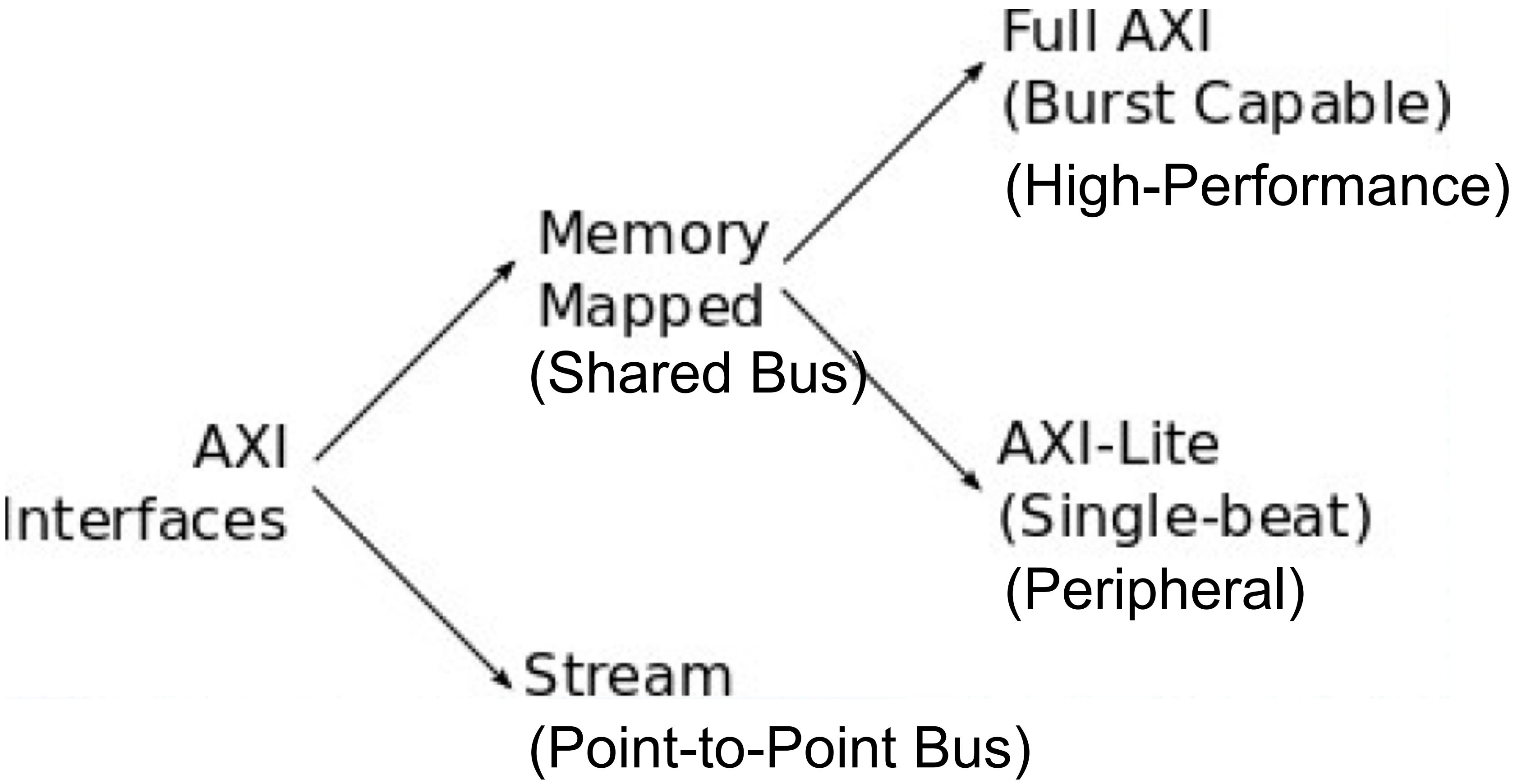
Write to Accelerator

- processor allocates buffer
- processor writes data into buffer
- ~~processor flushes cache for buffer~~
- processor initiates DMA transfer

Read from Accelerator

- processor allocates buffer
- processor initiates DMA transfer
- processor waits for DMA to complete
- ~~processor invalidates cache for buffer~~
- processor reads data from buffer

AXI Interfaces



AXI Interfaces

Memory Mapped
(Shared Bus)

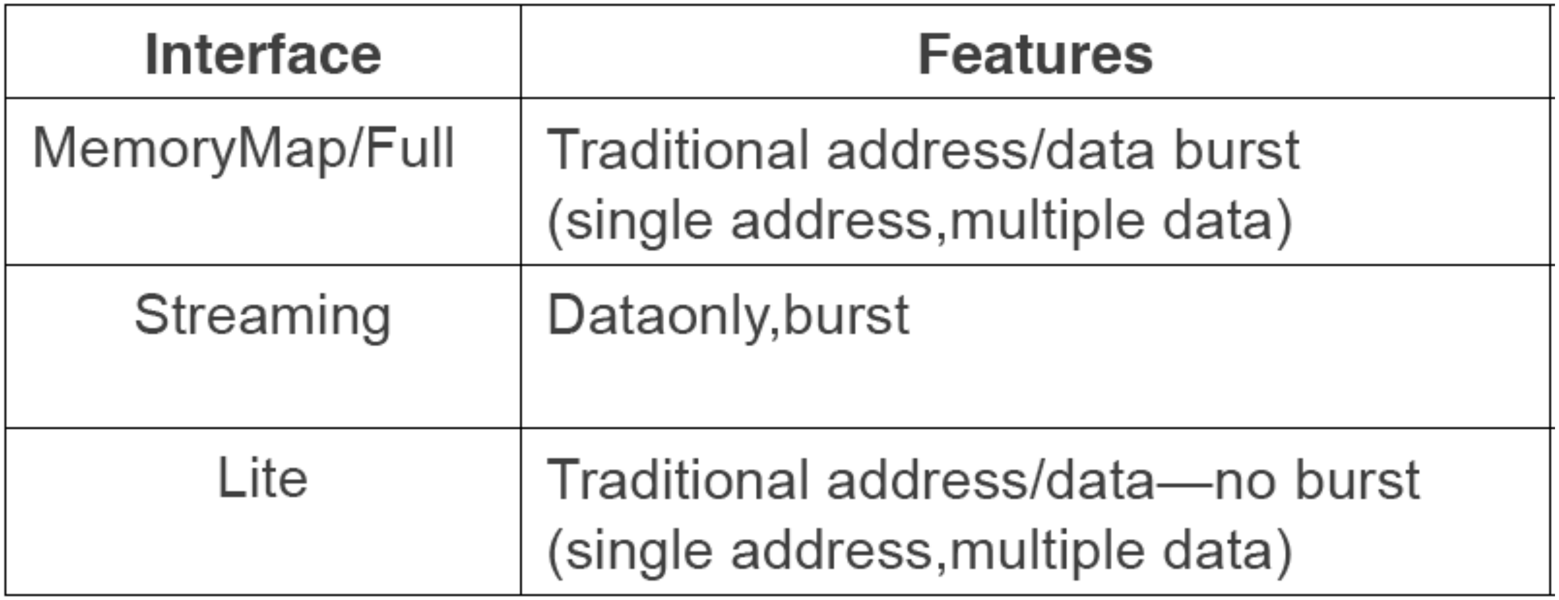
Full AXI
(Burst Capable)
(High-Performance)

AXI-Lite
(Single-beat)
(Peripheral)

Stream
(Point-to-Point Bus)

Source: M.S. Sadri, Zynq Training

Features of AXI Interfaces

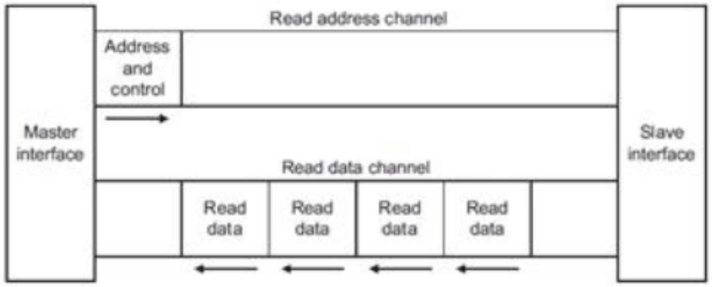


Interface	Features
MemoryMap/Full	Traditional address/data burst (single address,multiple data)
Streaming	Dataonly,burst
Lite	Traditional address/data—no burst (single address,multiple data)

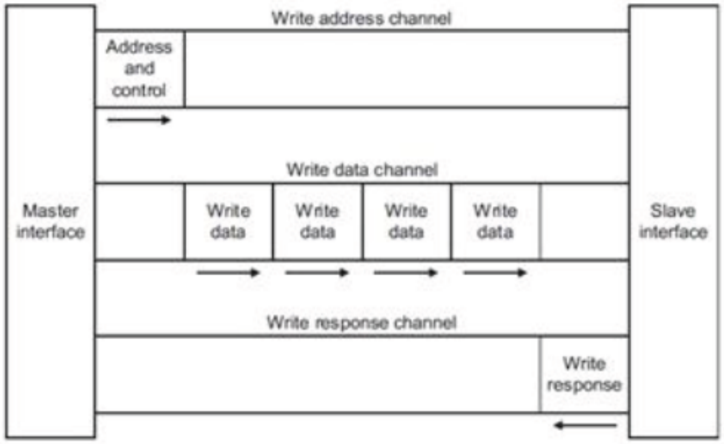
Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

Summary of AXI Full and AXI Lite Interfaces

- Memory mapped interfaces consist of 5 streams
  - Read Address
  - Read Data
  - Write Address
  - Write Data
  - Write Acknowledge
- Burst length limited to 256
- Data width limited to 256 bits for Xilinx IP
- AXI Lite is a subset
  - no bursts
  - 32 bit data width only



AXI4 READ

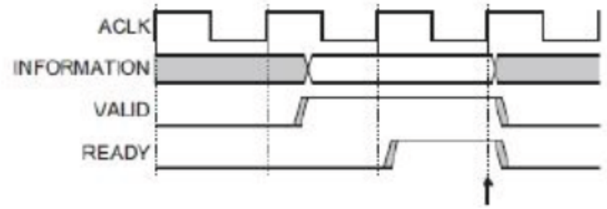


AXI4 Write

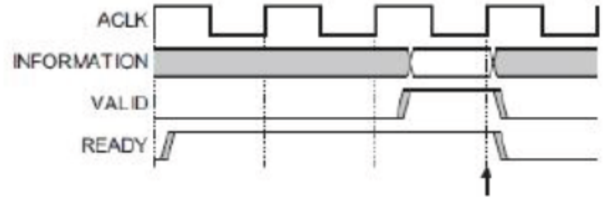
Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

Summary of AXI Stream Interface

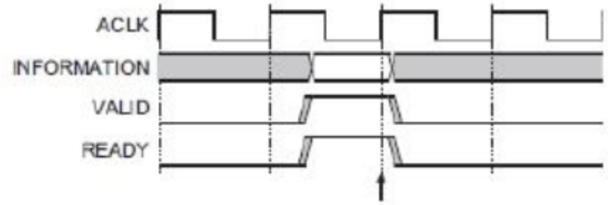
- AXI Streams are fully handshaked
  - Data is transferred when source asserts VALID and destination asserts READY
- 'Information' includes DATA and other side channel signals
  - STRB
  - KEEP
  - LAST
  - ID
  - DEST
  - USER
- Most of these are optional



Inserting Wait States



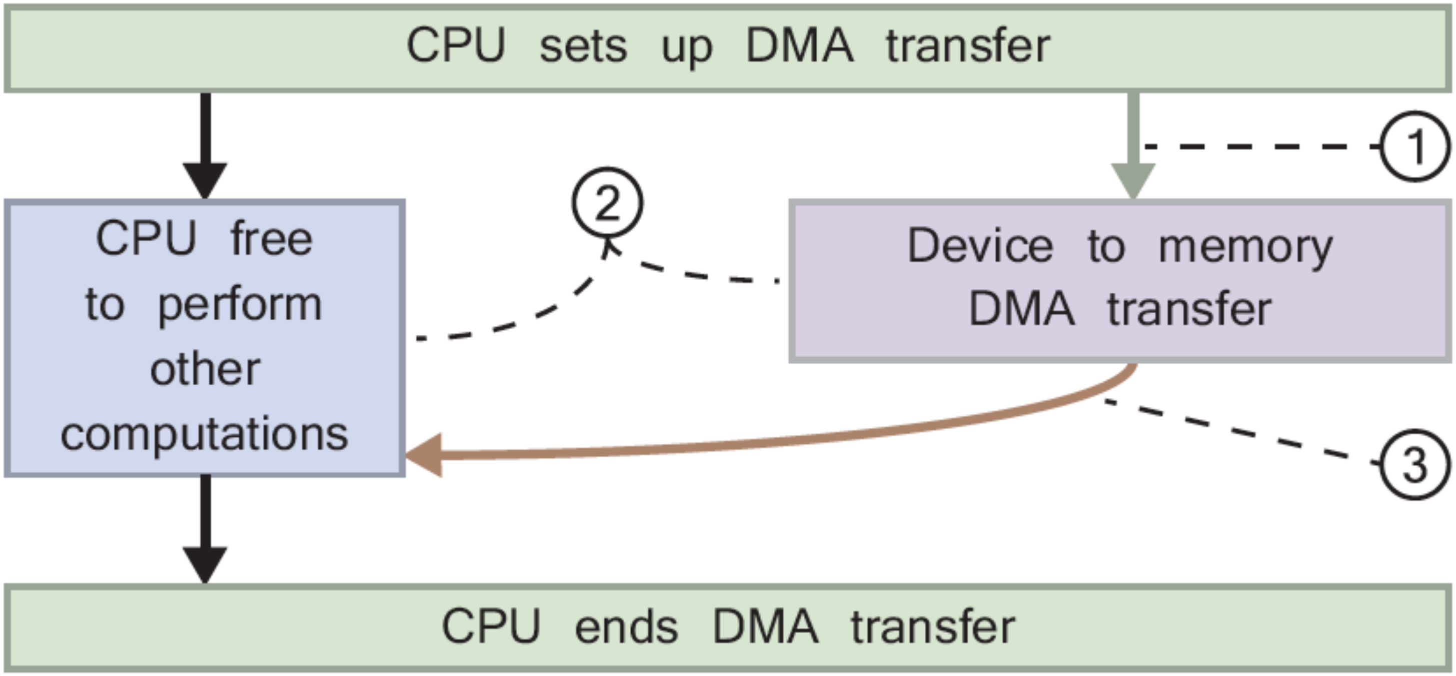
Always Ready



Same Cycle Acknowledge

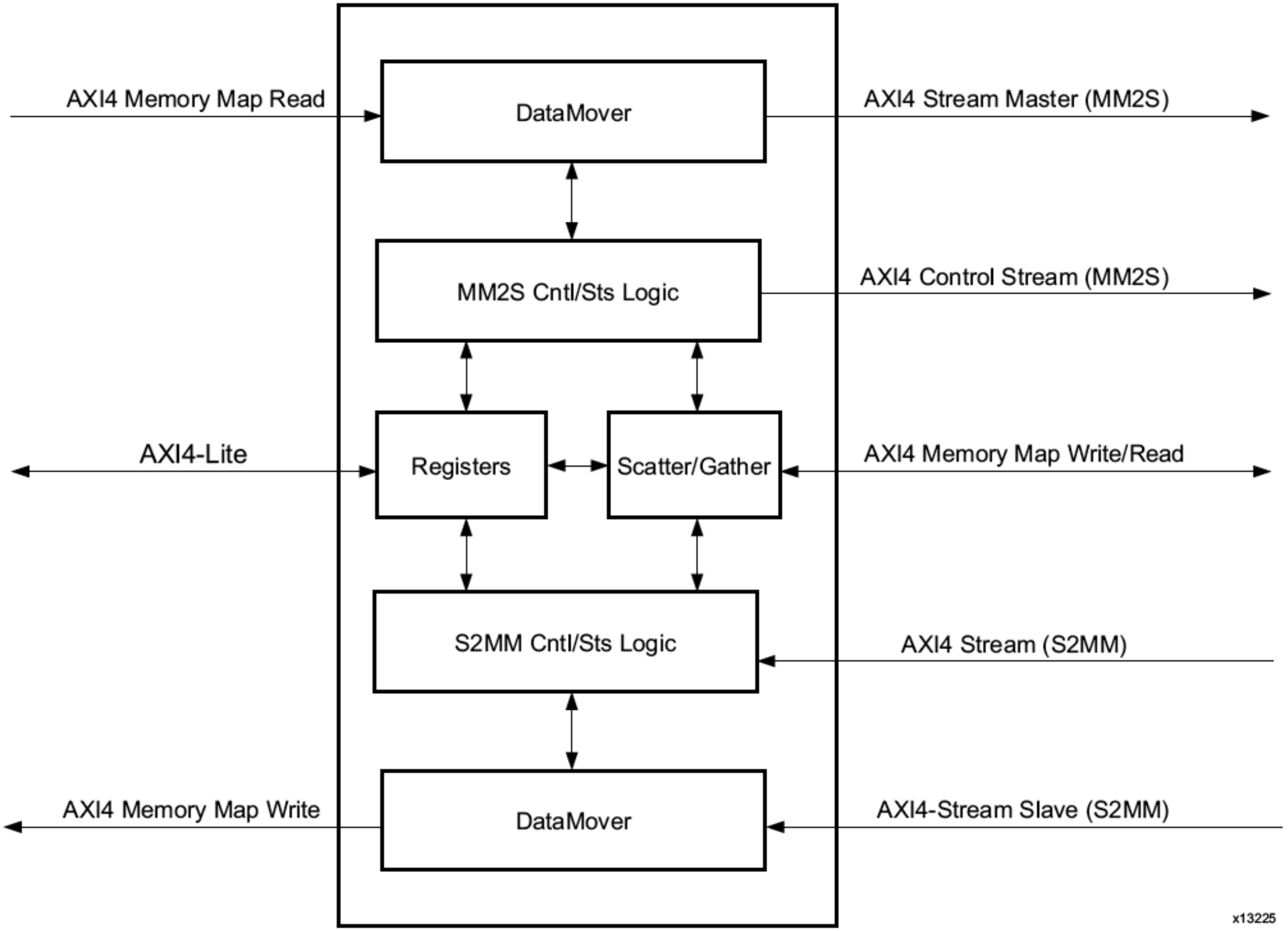
Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

DMA Memory Transfer Operation


CPU sets up DMA transfer
1
2
CPU free to perform other computations
Device to memory DMA transfer
3
CPU ends DMA transfer

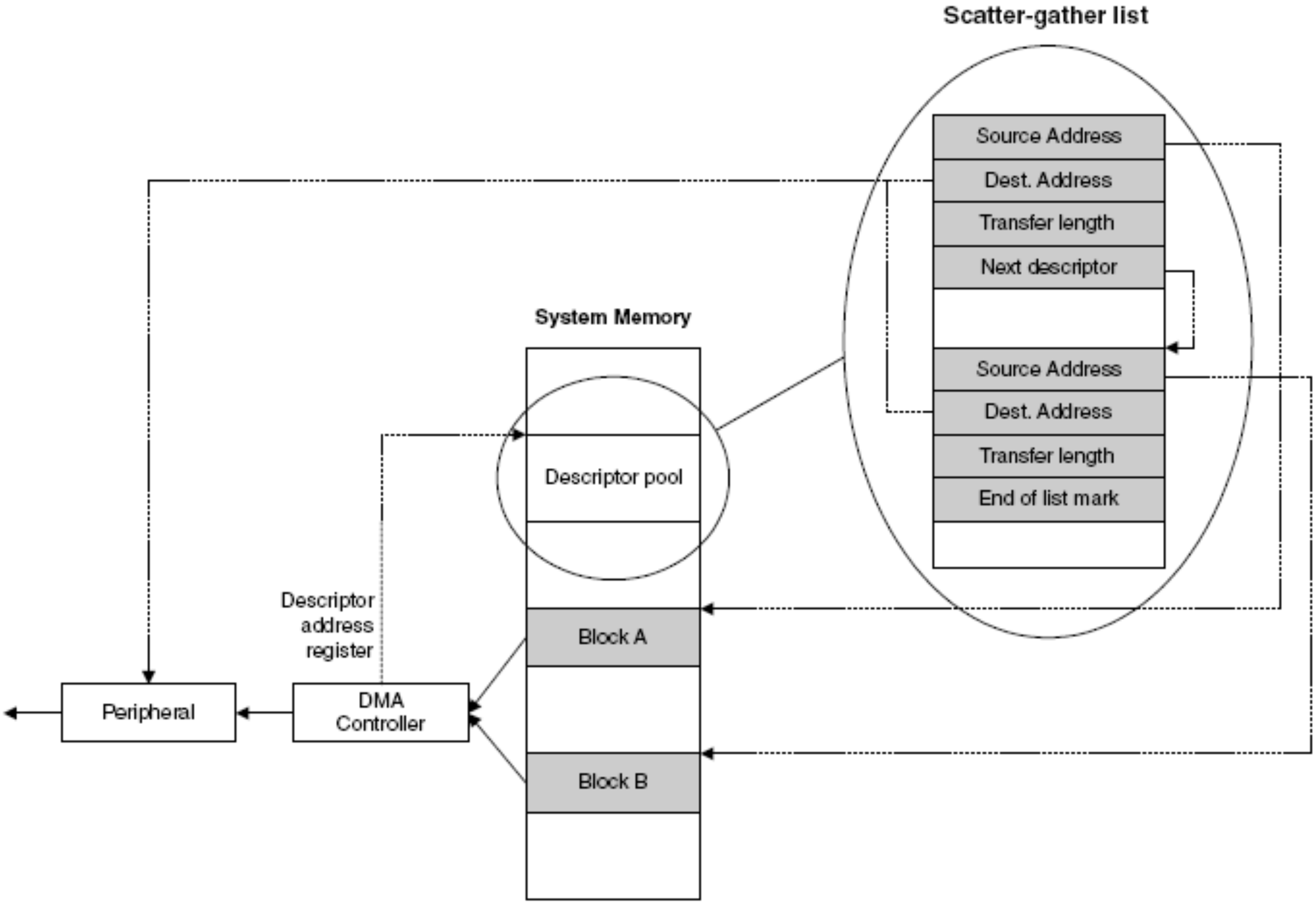
Source: The Zynq Book

Block Diagram of AXI DMA Core



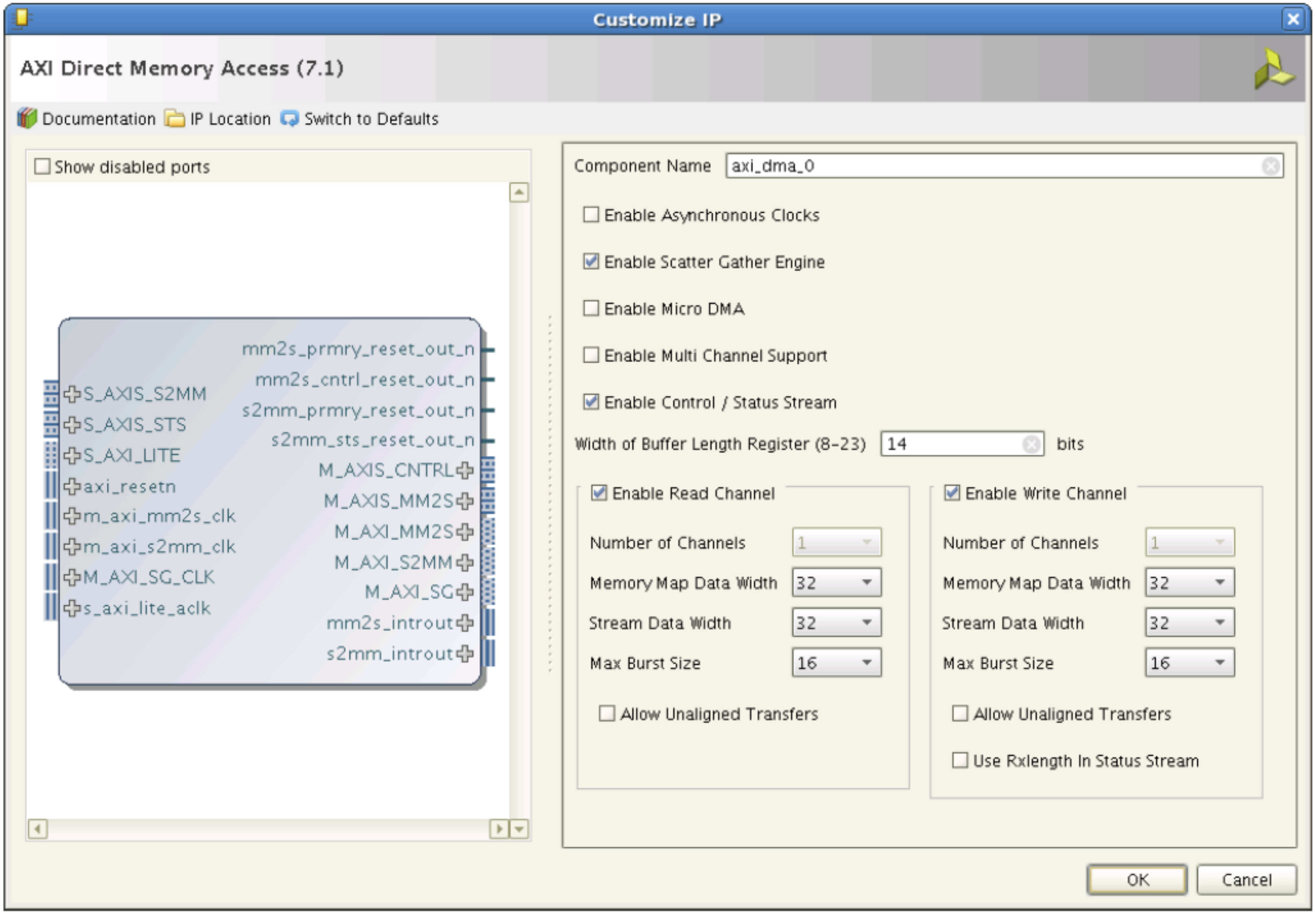
Source: LogiCORE IP AXI DMA v7.1: Product Guide

Scatter Gather DMA Mode



Source: Symbian OS Internals/13. Peripheral Support

Customizing AXI DMA Core



□ Show disabled ports

Component Name: axi_dma_0

□ Enable Asynchronous Clocks

☑ Enable Scatter Gather Engine

□ Enable Micro DMA

□ Enable Multi Channel Support

☑ Enable Control / Status Stream

Width of Buffer Length Register (8-23): 14 bits

☑ Enable Read Channel

- Number of Channels: 1
- Memory Map Data Width: 32
- Stream Data Width: 32
- Max Burst Size: 16
- □ Allow Unaligned Transfers

☑ Enable Write Channel

- Number of Channels: 1
- Memory Map Data Width: 32
- Stream Data Width: 32
- Max Burst Size: 16
- □ Allow Unaligned Transfers
- □ Use Rxlength In Status Stream

Source: LogiCORE IP AXI DMA v7.1: Product Guide

Parameters of AXI DMA Core (1)

Number of Channels

the number of channels: 1..16

Memory Map Data Width

data width in bits of the AXI MM2S Memory Map Read data bus: 32, 64, 128, 256, 512 or 1,024

Stream Data Width

data width in bits of the AXI MM2S AXI4-Stream Data bus:
8, 16, 32, 64, 128, 512 or 1,024;
Stream Data Width ≤ Memory Map Data Width;

Max Burst Size

maximum size of burst on the AXI4-Memory Map side of MM2S: 2, 4, 8,16, 32, 64, 128, or 256

Options of AXI DMA Core (1)

Enable Asynchronous Clocks

0 – all clocks inputs should be connected to the same clock signal
1 – separate asynchronous clocks for MM2S interface, S2MM interface, AXI4-Lite control interface, and the Scatter Gather Interface

Enable Scatter Gather Engine

0 – Simple DMA Mode operation
1 – Scatter Gather Mode operation; the Scatter Gather Engine included in AXI DMA

Options of AXI DMA Core (2)

Enable Micro DMA

0 – regular DMA
1 – area-optimized DMA; the maximum number of bytes per transaction = MMap_Data_width * Burst_length/8; addressing restricted to burst boundries

Enable Multi Channel Support

0 – the number of channels fixed at 1 for both directions
1 – the number of channels can be greater than 1

Options of AXI DMA Core (3)

Enable Control/Status Stream

0 – no AXI4 Control/Status Streams
1 – The AXI4 Control stream allows user application metadata associated with the MM2S channel to be transmitted to a target IP. The AXI4 Status stream allows user application metadata associated with the S2MM channel to be received from a target IP.

Width of Buffer Length Register (8-23)

For Simple DMA mode:
the number of valid bits in the MM2S_LENGTH and S2MM_LENGTH registers
For Scatter Gather mode:
the number of valid bits used for the Control field *buffer length* and Status field *bytes transferred* in the Scatter/Gather descriptors

Options of AXI DMA Core (4)

Allow Unaligned Transfers

Enables or disables the MM2S Data Realignment Engine (DRE). If the DRE is enabled, data reads can start from any Buffer Address byte offset, and the read data is aligned such that the first byte read is the first valid byte out on the AXI4-Stream.

Use RxLength In Status Stream

Allows AXI DMA to use a receive length field that is supplied by the S2MM target IP in the App4 field of the status packet. This gives AXI DMA a pre-determined receive byte count, allowing AXI DMA to command the exact number of bytes to be transferred.

Simple DMA Transfer
Programming Sequence for MM2S channel (1)

1. **Start the MM2S channel running** by setting the run/stop bit to 1, MM2S_DMACR.RS = 1.
2. If desired, **enable interrupts** by writing a 1 to MM2S_DMACR.IOC_IrqEn and MM2S_DMACR.Err_IrqEn.
3. Write a **valid source address** to the MM2S_SA register.
4. Write the **number of bytes to transfer** in the MM2S_LENGTH register.

   The MM2S_LENGTH register must be written last.

   All other MM2S registers can be written in any order.

Simple DMA Transfer

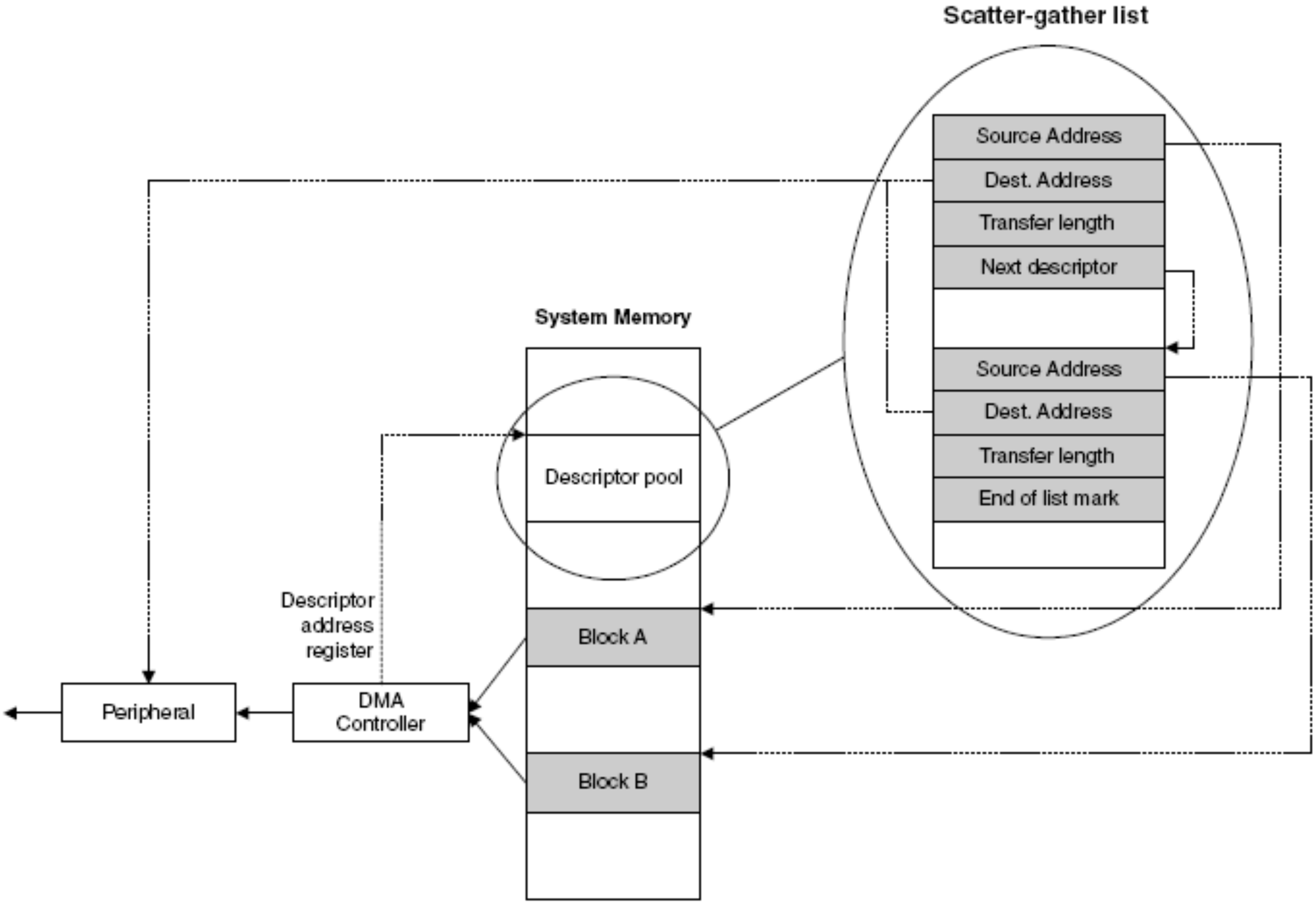
Programming Sequence for S2MM channel (1)

1. **Start the S2MM channel running** by setting the run/stop bit to 1, S2MM_DMACR.RS = 1.
2. If desired, **enable interrupts** by by writing a 1 to S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn.
3. Write a **valid destination address** to the S2MM_DA register.
4. Write the **length in bytes of the receive buffer** in the S2MM_LENGTH register.

   The S2MM_LENGTH register must be written last.

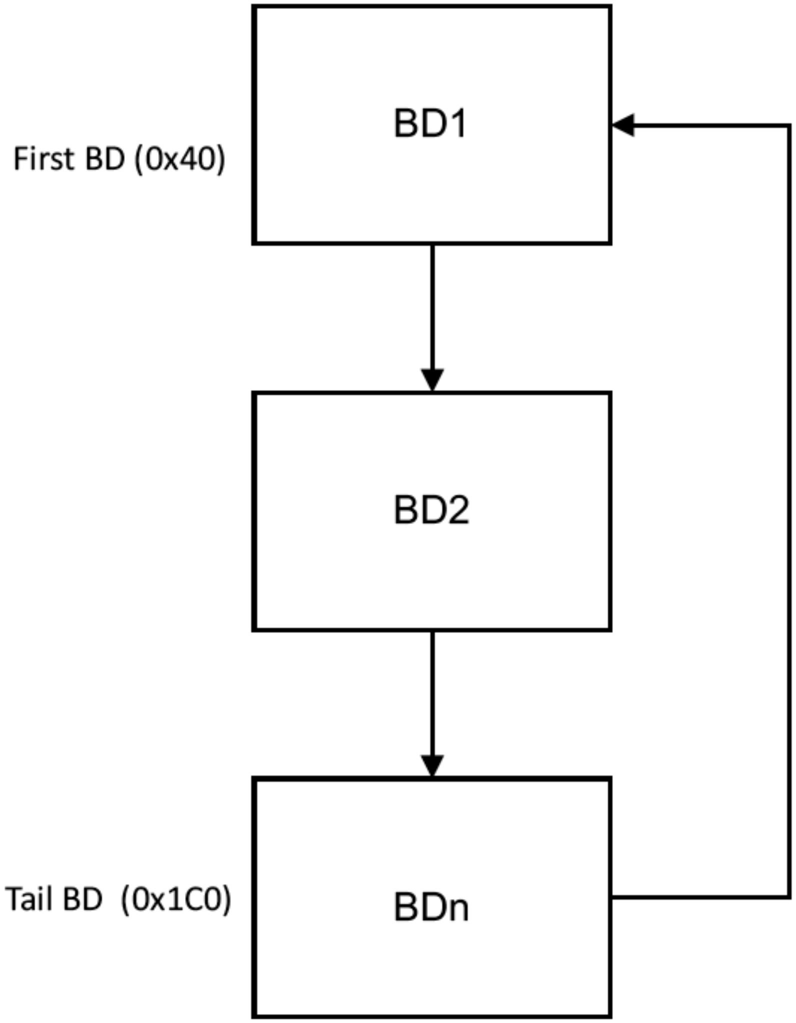
   All other S2MM registers can be written in any order.

Scatter Gather DMA Mode



Source: Symbian OS Internals/13. Peripheral Support

Chain of Buffer Descriptors (BDs)



Scatter Gather DMA Transfer
Programming Sequence for MM2S channel (1)

1. **Write the address of the starting descriptor** to **the Current Descriptor register**
2. **Start the MM2S channel running** by setting the run/stop bit to 1, MM2S_DMACR.RS = 1.
3. If desired, **enable interrupts** by writing a 1 to MM2S_DMACR.IOC_IrqEn and MM2S_DMACR.Err_IrqEn.
4. **Write a valid address to the Tail Descriptor register**.

   Writing to the Tail Descriptor register triggers the DMA to start fetching the descriptors from the memory.

Simple DMA Transfer
Programming Sequence for S2MM channel (1)

1. **Write the address of the starting descriptor** to **the Current Descriptor register**
2. **Start the S2MM channel running** by setting the run/stop bit to 1, S2MM_DMACR.RS = 1.
3. If desired, **enable interrupts** by by writing a 1 to S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn.
4. **Write a valid address to the Tail Descriptor register**.

   Writing to the Tail Descriptor register triggers the DMA to start fetching the descriptors from the memory.

Main Program

Main Program (1)

```
//============= Include Files =============/

#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"
#include "dma_lite_n_stream.h"
```

Main Program (2)

```
//============ Constant Definitions ============/
/* Device hardware build related constants */

#define DMA_DEV_ID   XPAR_AXIDMA_0_DEVICE_ID

// use RAM0 to store the data to be transferred to
// DMA --> FPGA coprocessor
#define MEM_BASE_ADDR      XPAR_PS7_RAM_0_S_AXI_BASEADDR

#define TX_BUFFER_BASE     (MEM_BASE_ADDR)
#define RX_BUFFER_BASE     (MEM_BASE_ADDR + 0x00001000)

#define M_REG_ADDR
        XPAR_DMA_LITE_N_STREAM_1_S00_AXI_BASEADDR
#define M_REG_0_OFFSET
        DMA_LITE_N_STREAM_S00_AXI_SLV_REG0_OFFSET
```

Main Program (3)

```
#define RX_INTR_ID
        XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID
        XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define INTC_DEVICE_ID     XPAR_SCUGIC_SINGLE_DEVICE_ID
#define INTC               XScuGic
#define INTC_HANDLER       XScuGic_InterruptHandler

/* Timeout loop counter for reset */
#define RESET_TIMEOUT_COUNTER  10000

/* Buffer and Buffer Descriptor related constant definitions */
#define MAX_PKT_LEN_WORDS        8     // 8 32-bit words
#define MAX_PKT_LEN   MAX_PKT_LEN_WORDS*4  // 8*4 = 32 bytes

#define YES 1
#define USE_TX_CHANNEL YES
#define USE_RX_CHANNEL YES
```

Main Program (4)

```
/************************** Function Prototypes ******************************/
static int CheckData(void);
static void TxIntrHandler(void *Callback);
static void RxIntrHandler(void *Callback);

static int SetupIntrSystem(INTC * IntcInstancePtr, XAxiDma * AxiDmaPtr,
                           u16 TxIntrId, u16 RxIntrId);
static void DisableIntrSystem(INTC * IntcInstancePtr, u16 TxIntrId,
                           u16 RxIntrId);

/************************** Variable Definitions *****************************/
/* Device instance definitions */

static XAxiDma AxiDma;          /* Instance of the XAxiDma */
static INTC Intc;               /* Instance of the Interrupt Controller */

/* Flags interrupt handlers use to notify the application */
volatile int TxDone;
volatile int RxDone;
volatile int Error;
```

Main Program (5)

```
int main(void)
{
   int Status;
   XAxiDma_Config *Config;

   xil_printf("\r\n--- Entering main() --- \r\n");

  // transfer packet length to PL through AXI-Lite interface
  DMA_LITE_N_STREAM_mWriteReg(MEM_BASE_ADDR,
                                  M_REG_0_OFFSET, MAX_PKT_LEN_WORDS);
   xil_printf("Number of words in Tx packet are %d",
                 Xil_In32(MEM_BASE_ADDR + M_REG_0_OFFSET) );

   // configure DMA engine
   Config = XAxiDma_LookupConfig(DMA_DEV_ID);
   if (!Config) {
            xil_printf("No config found for %d\r\n", DMA_DEV_ID);
            return XST_FAILURE;
   }
```

Main Program (6)

```
// Initialize DMA engine
Status = XAxiDma_CfgInitialize(&AxiDma, Config);
if (Status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE;
}

// check that scatter/ gather mode is disabled
if(XAxiDma_HasSg(&AxiDma)){
        xil_printf("Device configured as SG mode \r\n");
        return XST_FAILURE;
}

// Set up Interrupt system
Status = SetupIntrSystem(&Intc, &AxiDma, TX_INTR_ID, RX_INTR_ID);
if (Status != XST_SUCCESS) {

        xil_printf("Failed intr setup\r\n");
        return XST_FAILURE;
}
```

Main Program (7)

```
// Disable all interrupts before setup
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
          XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
          XAXIDMA_DEVICE_TO_DMA);

// Enable all interrupts
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
          XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
          XAXIDMA_DEVICE_TO_DMA);

// Initialize flags before start transfer test
TxDone = 0;
RxDone = 0;
Error = 0;
```

Main Program (8)

```
        u32 *TxBufferPtr;
        TxBufferPtr = (u32 *)TX_BUFFER_BASE ;
        *(TxBufferPtr)     = 0x11111111;
        *(TxBufferPtr + 1) = 0x089FF0BA;
        *(TxBufferPtr + 2) = 0xA3A777E0;
        *(TxBufferPtr + 3) = 0xFF2402DB;
        *(TxBufferPtr + 4) = 0xE4308049;
        *(TxBufferPtr + 5) = 0x78F59D1B;
        *(TxBufferPtr + 6) = 0x42F9B10F;
        *(TxBufferPtr + 7) = 0x00000000;

/* Flush the SrcBuffer before the DMA transfer */
        Xil_DCacheFlushRange((u32)TxBufferPtr, MAX_PKT_LEN);
/* Send a packet */
        Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) TxBufferPtr,
MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
        if (Status != XST_SUCCESS) { return XST_FAILURE; }
        while (!TxDone);
```

Main Program (9)

```
u32 *RxBufferPtr;
RxBufferPtr = (u32 *)RX_BUFFER_BASE;

/* Flush the DstBuffer before the DMA reception */
Xil_DCacheFlushRange((u32)RxBufferPtr, MAX_PKT_LEN);

/* Receive a packet */
Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) RxBufferPtr,
         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) { return XST_FAILURE; }
```

Main Program (10)

```
// Wait TX done, RX done, or Error
while (!TxDone && !RxDone && !Error) { /* NOP */ }
xil_printf("======================= \n");
Status = CheckData();

if (TxDone)         xil_printf("Transmission done \n");
else                xil_printf("Transmission not done \n");
if (RxDone)         xil_printf("Reception done \n");
else                xil_printf("Reception not done \n");

if (Error) {
xil_printf("Failed test transmit %s done, receive %s done \r\n",
           TxDone? "":" not", RxDone? "":" not");
}

if (Error) {
          goto Done;
}
```

Main Program (11)

```
	xil_printf("AXI DMA interrupt example test passed\r\n");

	/* Disable TX and RX Ring interrupts and return success */
	DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);

Done:
	xil_printf("--- Exiting main() --- \r\n");

	return XST_SUCCESS;
}
```