ECE 699: Lecture 5

AXI Interfacing

IP Creation
Required Reading

The ZYNQ Book Tutorials
- Tutorial 4: IP Creation
  - Exercise 4A: Creating IP in HDL

The ZYNQ Book
- Chapter 19: AXI Interfacing

ARM AMBA AXI Protocol v1.0: Specification
- Chapter 1: Introduction
- Chapter 2: Signal Descriptions
- Chapter 3: Channel Handshake
- Chapter 4: Addressing Options
- Chapter 9: Data Buses
Recommended Reading

P. Schaumont, A Practical Introduction to Hardware/Software Codesign, 2nd Ed.

- Chapter 10: On-Chip Buses

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
Components of Today’s Systems-on-Chip

- Digital Signal Processor (Green)
- UART (Purple)
- USB (Purple)
- DRAM Controller (Blue)
- Shared Memory (Yellow)
- High Performance CPU (Gray)
- Low Performance CPU (Pink)
- Ethernet (Purple)
- Video Controller (Dark Gray)
- ADC/DAC (Pink)

Source: M.S. Sadri, Zynq Training
Connectivity Requirements

- A standard
  - All units talk based on that standard
  - All units can talk easily to each other

- Maintenance
  - Design is easily maintained/updated, debugged

- Re-use
  - Units can be easily re-used in other designs

Source: M.S. Sadri, Zynq Training
SoC Buses

A Standard Way of Communication between The Module and the Bus!

Source: M.S. Sadri, Zynq Training
Solution Adopted in ZYNQ

Advanced Microcontroller Bus Architecture (AMBA): an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. First version introduced by ARM in 1996.

AMBA Advanced eXtensible Interface 4 (AXI4): the fourth generation of AMBA interface defined in the AMBA 4 specification, targeted at high performance, high clock frequency systems. Introduced by ARM in 2010.

Source: M.S. Sadri, Zynq Training
Basic Concepts

• Transaction:
  - Transfer of data from one point in the hardware to another point

• Master: Initiates the transaction

• Slave: Responds to the initiated transaction

Source: M.S. Sadri, Zynq Training
Communication Between AXI Master and AXI Slave

Source: M.S. Sadri, Zynq Training
Additional Information Exchanged Between AXI Master and AXI Slave

Source: M.S. Sadri, Zynq Training
Five Channels of AXI Interface

Source: M.S. Sadri, Zynq Training
Connecting Masters and Slaves

AXI Master 1 (CPU)

AXI Master 2 (DMA)

AXI Slave 1 (UART)

AXI Slave 2 (GPIO)

AXI Slave 3 (RAM)

Source: M.S. Sadri, Zynq Training
AXI Interconnect

AXI Master 1 (CPU)

AXI Master 2 (DMA)

AXI Slave 1 (UART)

AXI Slave 2 (GPIO)

AXI Slave 3 (RAM)
Interconnect vs. Interface
Addressing of Slaves

Source: M.S. Sadri, Zynq Training
AXI Interconnect Address Decoding

Address Decoding Table
UART: 0x40000000 - 0x40000FFF
GPIO: 0x40001000 - 0x40001FFF
RAM: 0x40010000 - 0x4001FFFF

AXI Master 1 (CPU)
AXI Master 2 (DMA)
AXI Interconnect
AXI Slave 1 (UART)
AXI Slave 2 (GPIO)
AXI Slave 3 (RAM)

Address Range: 4K
Address Offset: 0x40000000
Address: 0x40000000 - 0x40000FFF
Address Range: 4K
Address Offset: 0x40001000
Address: 0x40001000 - 0x40001FFF
Address Range: 64K
Address Offset: 0x40010000
Address: 0x40010000 - 0x4001FFFF

Source: M.S. Sadri, Zynq Training
Clock Domain and Width Conversion

Source: M.S. Sadri, Zynq Training
Hierarchical AXI Interconnects

Source: M.S. Sadri, Zynq Training
Simple Address Definition Rules
No Overlaps

Source: M.S. Sadri, Zynq Training
Simple Address Definition Rules

Address Alignment

- **AXI Master 1 (CPU)**
- **AXI Master 2 (DMA)**
- **AXI Interconnect**
- **AXI Slave 1 (UART)**
- **AXI Slave 2 (GPIO)**

**AXI Slave 1 (UART)**
- Address Range: 4K
- Address Offset: 0x40000000
- Address: 0x40000000 - 0x40000FFF

**AXI Slave 2 (GPIO)**
- Address Range: 2G
- Address Offset: 0x40000000
- Address: 0x40000000 - .... (Wrong!)
Point-to-Point Data Flows

Signal Processing

A/D → FIR → DFT

Video Processing

Camera → Filter → Transform

Source: M.S. Sadri, Zynq Training
AXI Memory-Mapped vs. AXI Stream

Source: M.S. Sadri, Zynq Training
Selected AXI Stream Ports

Source: M.S. Sadri, Zynq Training
AXI Port Naming Conventions

- Memory Mapped
  - M_AXI...
  - AXI Unit (MM)
  - S_AXI...

- Stream
  - M_AXIS...
  - AXI Unit (S)
  - S_AXIS...

- Signal names:
  - e.g. S_AXI_awid, M_AXI_arready ....

Source: M.S. Sadri, Zynq Training
AXI Interfaces

- Memory Mapped (Shared Bus)
- AXI-Lite (Single-beat) (Peripheral)
- Full AXI (Burst Capable) (High-Performance)
- Stream (Point-to-Point Bus)

Source: M.S. Sadri, Zynq Training
Concept of a Burst

Source: M.S. Sadri, Zynq Training
### Competing System-on-Chip Bus Standards

<table>
<thead>
<tr>
<th>Bus</th>
<th>Developed by</th>
<th>High-Performance Shared Bus</th>
<th>Peripheral Shared Bus</th>
<th>Point-to-Point Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA v3</td>
<td>ARM</td>
<td>AHB</td>
<td>APB</td>
<td></td>
</tr>
<tr>
<td>AMBA v4</td>
<td>ARM</td>
<td>AXI4</td>
<td>AXI4-Lite</td>
<td>AXI4-Stream</td>
</tr>
<tr>
<td>Coreconnect</td>
<td>IBM</td>
<td>PLB</td>
<td>OPB</td>
<td></td>
</tr>
<tr>
<td>Wishbone</td>
<td>SiliCore Corp.</td>
<td>Crossbar Topology</td>
<td>Shared Topology</td>
<td>Point to Point Topology</td>
</tr>
<tr>
<td>Avalon</td>
<td>Altera</td>
<td>Avalon-MM</td>
<td>Avalon-MM</td>
<td>Avalon-ST</td>
</tr>
</tbody>
</table>

AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced eXtensible Interface
AHB: AMBA High-speed Bus
APB: AMBA Peripheral Bus
PLB: Processor Local Bus
OPB: On-chip Peripheral Bus
MM: Memory Mapped
ST: Streaming

Source: A Practical Introduction to Hardware/Software Codesign
AXI4 Write

Source: The Zynq Book
AXI4 Read

Source: The Zynq Book
AXI4 Interface

Write Address Channel

Write Data Channel

Write Response Channel

Read Address Channel

Read Data Channel

Source: The Zynq Book
Prefixes of Ports from Particular Channels

- **Write Address Channel** — the signals contained within this channel are named in the format `s_axi_aw...

- **Write Data Channel** — the signals contained within this channel are named in the format `s_axi_w...

- **Write Response Channel** — the signals contained within this channel are named in the format `s_axi_b...

- **Read Address Channel** — the signals contained within this channel are named in the format `s_axi_ar...

- **Read Data Channel** — the signals contained within this channel are named in the format `s_axi_r...

Source: The Zynq Book
Timing Diagram Conventions

- Clock
- HIGH to LOW
- Transient
- HIGH/LOW to HIGH
- Bus stable
- Bus to high impedance
- Bus change
- High impedance to stable bus

Source: ARM AMBA AXI Protocol v1.0: Specification
VALID before READY Handshake

Source: ARM AMBA AXI Protocol v1.0: Specification
READY before VALID Handshake

Source: ARM AMBA AXI Protocol v1.0: Specification
VALID with READY Handshake
Channel Architecture of Reads

Source: ARM AMBA AXI Protocol v1.0: Specification
Read Burst

Source: ARM AMBA AXI Protocol v1.0: Specification
Overlapping Read Bursts

Source: ARM AMBA AXI Protocol v1.0: Specification
Read Transaction
Handshake Dependencies

```
ARVALID -> RVALID
  |
  v
ARREADY  RREADY
```
Channel Architecture of Writes
Write Burst

Source: ARM AMBA AXI Protocol v1.0: Specification
Write Transaction Handshake Dependencies

Source: ARM AMBA AXI Protocol v1.0: Specification
<table>
<thead>
<tr>
<th>ARBURST[1:0]</th>
<th>AWBURST[1:0]</th>
<th>Burst type</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>b00</td>
<td></td>
<td>FIXED</td>
<td>Fixed-address burst</td>
<td>FIFO-type</td>
</tr>
<tr>
<td>b01</td>
<td></td>
<td>INCR</td>
<td>Incrementing-address burst</td>
<td>Normal sequential memory</td>
</tr>
<tr>
<td>b10</td>
<td></td>
<td>WRAP</td>
<td>Incrementing-address burst that wraps to a lower address at the wrap boundary</td>
<td>Cache line</td>
</tr>
<tr>
<td>b11</td>
<td></td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Source: ARM AMBA AXI Protocol v1.0: Specification
Role of Write Strobe

WSTRB

<table>
<thead>
<tr>
<th>63</th>
<th>56</th>
<th>55</th>
<th>48</th>
<th>47</th>
<th>40</th>
<th>39</th>
<th>32</th>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WSTRB[n] corresponds to
WDATA[8*n+7 downto 8*n]
# Narrow Transfer Example with 8-bit Transfers

<table>
<thead>
<tr>
<th>Byte lane used</th>
<th>DATA[7:0]</th>
<th>1st transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DATA[15:8]</td>
<td>2nd transfer</td>
</tr>
<tr>
<td></td>
<td>DATA[23:16]</td>
<td>3rd transfer</td>
</tr>
<tr>
<td>DATA[31:24]</td>
<td></td>
<td>4th transfer</td>
</tr>
<tr>
<td></td>
<td>DATA[7:0]</td>
<td>5th transfer</td>
</tr>
</tbody>
</table>

Source: ARM AMBA AXI Protocol v1.0: Specification
### Narrow Transfer Example with 32-bit Transfers

<table>
<thead>
<tr>
<th>Byte lane used</th>
<th>1st transfer</th>
<th>2nd transfer</th>
<th>3rd transfer</th>
</tr>
</thead>
</table>

Source: ARM AMBA AXI Protocol v1.0: Specification
Aligned and Unaligned Word Transfers on a 32-bit Bus

Address: 0x00
Transfer size: 32 bits
Burst type: incrementing
Burst length: 4 transfers

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st transfer
2nd transfer
3rd transfer
4th transfer

Address: 0x01
Transfer size: 32 bits
Burst type: incrementing
Burst length: 4 transfers

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st transfer
2nd transfer
3rd transfer
4th transfer

Source: ARM AMBA AXI Protocol v1.0: Specification
### Aligned and Unaligned Word Transfers on a 64-bit Bus

<table>
<thead>
<tr>
<th>Address: 0x00</th>
<th>Transfer size: 32 bits</th>
<th>Burst type: incrementing</th>
<th>Burst length: 4 transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0</td>
<td>1st transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>2nd transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>3rd transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F E D C B A 9 8</td>
<td>4th transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F E D C B A 9 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address: 0x07</th>
<th>Transfer size: 32 bits</th>
<th>Burst type: incrementing</th>
<th>Burst length: 4 transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0</td>
<td>1st transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>2nd transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F E D C B A 9 8</td>
<td>3rd transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F E D C B A 9 8</td>
<td>4th transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 16 15 14 13 12 11 10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: ARM AMBA AXI Protocol v1.0: Specification
Example of IP Core with AXI Interface

Source: The Zynq Book
Exit from Reset

Source: ARM AMBA AXI Protocol v1.0: Specification
Custom IP Core Used in Class Exercise

Source: The Zynq Book Tutorials
Custom IP Core Used in Class Exercise

Source: The Zynq Book Tutorials
Class Exercise:
Modifying a Counter

Source: The Zynq Book Tutorials
Displaying Consecutive LED Values

```
LED value: 0
LED value: 1
LED value: 2
LED value: 3
LED value: 4
LED value: 5
LED value: 6
LED value: 7
LED value: 8
```

Source: The Zynq Book Tutorials
Creating and Packaging Custom IP Core
Configuring AXI Interface

Source: The Zynq Book Tutorials
Adding a Software Driver

The table below lists all the components found in your hardware system. You can modify the driver (or its version) assigned for each component. If you do not want to assign a driver to a component or peripheral, please choose 'none'.

<table>
<thead>
<tr>
<th>Component</th>
<th>Component Type</th>
<th>Driver</th>
<th>Driver Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps7_cortexa9_0</td>
<td>ps7_cortexa9</td>
<td>cpu_cortexa9</td>
<td>1.01.a</td>
</tr>
<tr>
<td>led_controller_0</td>
<td>led_controller</td>
<td>led_controller</td>
<td>1.00.a</td>
</tr>
<tr>
<td>ps7_afi_0</td>
<td>ps7_afi</td>
<td>generic</td>
<td>1.00.a</td>
</tr>
<tr>
<td>ps7_afi_1</td>
<td>ps7_afi</td>
<td>generic</td>
<td>1.00.a</td>
</tr>
<tr>
<td>ps7_afi_2</td>
<td>ps7_afi</td>
<td>generic</td>
<td>1.00.a</td>
</tr>
<tr>
<td>ps7_afi_3</td>
<td>ps7_afi</td>
<td>generic</td>
<td>1.00.a</td>
</tr>
</tbody>
</table>

Source: The Zynq Book Tutorials
Mapping of an Embedded SoC Hardware Architecture to Zynq

Source: Xilinx White Paper: Extensible Processing Platform
Block Design for Class Exercise
Constraint File
entity design_int_wrapper is
    port ( 
        DDR_addr : inout STD_LOGIC_VECTOR ( 14 downto 0 );
        DDR_ba : inout STD_LOGIC_VECTOR ( 2 downto 0 );
        DDR_cas_n : inout STD_LOGIC;
        DDR_ck_n : inout STD_LOGIC;
        DDR_ck_p : inout STD_LOGIC;
        DDR_cke : inout STD_LOGIC;
        DDR_cs_n : inout STD_LOGIC;
        DDR_dm : inout STD_LOGIC_VECTOR ( 3 downto 0 );
        DDR_dq : inout STD_LOGIC_VECTOR ( 31 downto 0 );
        DDR_dqs_n : inout STD_LOGIC_VECTOR ( 3 downto 0 );
        DDR_dqs_p : inout STD_LOGIC_VECTOR ( 3 downto 0 );
        DDR_odt : inout STD_LOGIC;
        DDR_ras_n : inout STD_LOGIC;
        DDR_reset_n : inout STD_LOGIC;
        DDR_we_n : inout STD_LOGIC;
        FIXED_IO_ddr_vrn : inout STD_LOGIC;
        FIXED_IO_ddr_vrp : inout STD_LOGIC;
        FIXED_IO_mio : inout STD_LOGIC_VECTOR ( 53 downto 0 );
        FIXED_IO_ps_clk : inout STD_LOGIC;
        FIXED_IO_ps_porb : inout STD_LOGIC;
        FIXED_IO_ps_srstb : inout STD_LOGIC;
        LEDs_out : out STD_LOGIC_VECTOR ( 3 downto 0 );
    );
end design_int_wrapper;
design_1_i: component design_1
    port map (  
        DDR_addr(14 downto 0) => DDR_addr(14 downto 0),
        DDR_ba(2 downto 0) => DDR_ba(2 downto 0),
        DDR_cas_n => DDR_cas_n,
        DDR_ck_n => DDR_ck_n,
        DDR_ck_p => DDR_ck_p,
        DDR_cke => DDR_cke,
        DDR_cs_n => DDR_cs_n,
        DDR_dm(3 downto 0) => DDR_dm(3 downto 0),
        DDR_dq(31 downto 0) => DDR_dq(31 downto 0),
        DDR_dqs_n(3 downto 0) => DDR_dqs_n(3 downto 0),
        DDR_dqs_p(3 downto 0) => DDR_dqs_p(3 downto 0),
        DDR_odt => DDR_odt,
        DDR_ras_n => DDR_ras_n,
        DDR_reset_n => DDR_reset_n,
        DDR_we_n => DDR_we_n,
        FIXED_IO_ddr_vrn => FIXED_IO_ddr_vrn,
        FIXED_IO_ddr_vrp => FIXED_IO_ddr_vrp,
        FIXED_IO_mio(53 downto 0) => FIXED_IO_mio(53 downto 0),
        FIXED_IO_ps_clk => FIXED_IO_ps_clk,
        FIXED_IO_ps_porb => FIXED_IO_ps_porb,
        FIXED_IO_ps_srstb => FIXED_IO_ps_srstb,
        LEDs_out(3 downto 0) => LEDs_out (3 downto 0)
    );
ZYBO General Purpose Input Output (GPIO)
## LEDs

### IO_L23P_T3_35
set_property PACKAGE_PIN M14 [get_ports {LEDs_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs_out[0]}]

### IO_L23N_T3_35
set_property PACKAGE_PIN M15 [get_ports {LEDs_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs_out[1]}]

### IO_0_35
set_property PACKAGE_PIN G14 [get_ports {LEDs_out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs_out[2]}]

### IO_L3N_T0_DQS_AD1N_35
set_property PACKAGE_PIN D18 [get_ports {LEDs_out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs_out[3]}]
Main Program
Main Program (1)

/* Generated driver function for led_controller IP core */
#include "led_controller.h"
#include "xparameters.h"

// Define maximum LED value (2^4)-1 = 15
#define LED_LIMIT 15
// Define delay length
#define DELAY 10000000

/* Define the base memory address of the led_controller IP core */
#define LED_BASE XPAR_LED_CONTROLLER_0_S00_AXI_BASEADDR
int main(void) {

    /* unsigned 32-bit variables for storing current LED value */
    u32 led_val = 0;
    int i=0;

    xil_printf("led_controller IP test begin.\r\n");
xil_printf("----------------------------------------------\r\n\n");
Main Program (3)

/* Loop forever */
while(1) {
    while(led_val<=LED_LIMIT){
        /* Print value to terminal */
xil_printf("LED value: %d\r\n", led_val);
        /* Write value to led_controller IP core using generated 
          driver function */
        **LED_CONTROLLER_mWriteReg(LED_BASE, 0, led_val);**
        /* increment LED value */
        led_val++;
        /* run a simple delay to allow changes on LEDs to be visible */
        for(i=0;i<DELAY;i++);
    }

    /* Reset LED value to zero */
    led_val = 0;
}

return 1;
Device Driver
led_controller.h
#include "xil_types.h"
#include "xstatus.h"

#define LED_CONTROLLER_S00_AXI_SLV_REG0_OFFSET 0
#define LED_CONTROLLER_S00_AXI_SLV_REG1_OFFSET 4
#define LED_CONTROLLER_S00_AXI_SLV_REG2_OFFSET 8
#define LED_CONTROLLER_S00_AXI_SLV_REG3_OFFSET 12
/* Write a value to a LED_CONTROLLER register. A 32 bit write is performed. 
* If the component is implemented in a smaller width, only the least 
* significant data is written. 
* 
* @param BaseAddress is the base address of the LED_CONTROLLER 
*     device. 
* @param RegOffset is the register offset from the base to write to. 
* @param Data is the data written to the register. 
* 
* @return None. 
* 
* @note C-style signature: 
* void LED_CONTROLLER_mWriteReg(u32 BaseAddress, 
*     unsigned RegOffset, u32 Data) 
* /

#define LED_CONTROLLER_mWriteReg(BaseAddress, RegOffset, Data) 
    Xil_Out32(((BaseAddress) + (RegOffset)), (u32)(Data))
/* Read a value from a LED_CONTROLLER register.  
* A 32 bit read is performed.  
* If the component is implemented in a smaller width, only the least  
* significant data is read from the register. The most significant data  
* will be read as 0.  
*  
* @param BaseAddress is the base address of the LED_CONTROLLER  
*    device.  
* @param RegOffset is the register offset from the base to write to.  
*  
* @return Data is the data from the register.  
*  
* @note C-style signature:  
* u32 LED_CONTROLLER_mReadReg(u32 BaseAddress,  
*   unsigned RegOffset)  
* /

#define LED_CONTROLLER_mReadReg(BaseAddress, RegOffset)  
   Xil_In32((BaseAddress) + (RegOffset))

Standard IO Functions
xil_io.c
/* Contains I/O functions for memory-mapped or non-memory-mapped I/O architectures. These functions encapsulate Cortex A9 architecture-specific I/O requirements. */

/* Performs an input operation for a 32-bit memory location by reading from the specified address and returning the Value read from that address. *
* @param Addr contains the address to perform the input operation at.
* @return The Value read from the specified input address. */

u32 Xil_In32(u32 Addr)
{
    return *(volatile u32 *) Addr;
}
/* Performs an output operation for a 32-bit memory location by writing the
* specified Value to the specified address.
* 
* @param OutAddress contains the address to perform the output
* operation at.
* @param Value contains the Value to be output at the specified address.
* 
* @return None.
*/

void Xil_Out32(u32 OutAddress, u32 Value)
{
    *(volatile u32 *) OutAddress = Value;
}
VHDL Code
led_controller_v1_0_S00_AXI.vhd
Entity Declaration (1)

entity led_controller_v1_0_S00_AXI is
  generic (  
    -- Users to add parameters here

    -- User parameters ends
    -- Do not modify the parameters beyond this line

    -- Width of S_AXI data bus
    C_S_AXI_DATA_WIDTH       : integer := 32;

    -- Width of S_AXI address bus
    C_S_AXI_ADDR_WIDTH       : integer := 4
  );
Add AXI4 interfaces supported by your peripheral

- Enable Interrupt Support
- Interfaces:
  - led_controller_v1.0
  - S00_AXI

Name: S00_AXI
Interface Type: Lite
Interface Mode: Slave
Data Width (Bits): 32
Memory Size (Bytes): 64
Number of Registers: [4..512]

Source: The Zynq Book Tutorials
Entity Declaration (2)

port (
    -- Users to add ports here
    LEDs_out : out std_logic_vector(3 downto 0);

    -- User ports ends
    -- Do not modify the ports beyond this line

    -- Global Clock Signal
    S_AXI_ACLK    : in std_logic;
    -- Global Reset Signal. This Signal is Active LOW
    S_AXI_ARESETN: in std_logic;
    -- Write address (issued by master, accepted by Slave)
    S_AXI_AWADDR: in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);

    . . . . . . .
)
-- Read address valid. This signal indicates that the channel
-- is signaling valid read address and control information.
S_AXI_ARVALID : in std_logic;
-- Read address ready. This signal indicates that the slave is
-- ready to accept an address and associated control signals.
S_AXI_ARREADY : out std_logic;
-- Read data (issued by slave)
S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
-- Read response. This signal indicates the status of the
-- read transfer.
S_AXI_RRESP : out std_logic_vector(1 downto 0);
-- Read valid. This signal indicates that the channel is
-- signaling the required read data.
S_AXI_RVALID : out std_logic;
-- Read ready. This signal indicates that the master can
-- accept the read data and response information.
S_AXI_RREADY : in std_logic);
end led_controller_v1_0_S00_AXI;
process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_arready <= '0';
      axi_araddr  <= (others => '1');
    else
      if (axi_arready = '0' and S_AXI_ARVALID = '1') then
        -- indicates that the slave has acceped the valid read address
        axi_arready <= '1';
        -- Read Address latching
        axi_araddr  <= S_AXI_ARADDR;
      else
        axi_arready <= '0';
      end if;
    end if;
  else
    axi_arready <= '0';
  end if;
end if;
end process;
process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_rvalid <= '0';
      axi_rresp  <= "00";
    else
      if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
        -- Valid read data is available at the read data bus
        axi_rvalid <= '1';
        axi_rresp  <= "00"; -- 'OKAY' response
      elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
        -- Read data is accepted by the master
        axi_rvalid <= '0';
      end if;
    end if;
  end if;
end process;
memory mapped read logic

slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;
process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, 
       S_AXI_ARESETN, slv_reg_rden)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
        when b"00" =>
            reg_data_out <= slv_reg0;
        when b"01" =>
            reg_data_out <= slv_reg1;
        when b"10" =>
            reg_data_out <= slv_reg2;
        when b"11" =>
            reg_data_out <= slv_reg3;
        when others =>
            reg_data_out  <= (others => '0')
    end case;
end process;
process(S_AXI_ACLK) is
begin
  if (rising_edge(S_AXI_ACLK)) then
    if (S_AXI_ARESETN = '0') then
      axi_rdata <= (others => '0');
    else
      if (slv_reg_rden = '1') then
        -- When there is a valid read address (S_AXI_ARVALID) with
        -- acceptance of read address by the slave (axi_arready),
        -- output the read data
        -- Read address mux
        axi_rdata <= reg_data_out;  -- register read data
        end if;
      end if;
    end if;
  end if;
end process;
User Logic

-- Add user logic here
LEDs_out <= slv_reg0(3 downto 0);

-- User logic ends

end arch_imp;